Eighth Edition

LINUX ADMINISTRATION A BEGINNER'S GUIDE

- Set up, maintain, and secure the latest Linux distributions on site and in cloud environments
- Configure and manage file systems, as well as e-mail, FTP, DNS, Web, VoIP, LDAP, SSH, and print servers
- Build and deploy virtual machines, containers, and servers

WALE SOYINKA

Mc Graw Hill

Linux Administration
A Beginner's Guide

Eighth Edition

Wale Soyinka

New York Chicago San Francisco
Athens London Madrid Mexico City
Milan New Delhi Singapore Sydney Toronto

Copyright © 2020 by McGraw Hill LLC. All rights reserved. Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher, with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication.

ISBN: 978-1-26-044171-0
MHID: 1-26-044171-7

The material in this eBook also appears in the print version of this title: ISBN: 978-1-26-044170-3, MHID: 1-26-044170-9.

eBook conversion by codeMantra
Version 1.0

All trademarks are trademarks of their respective owners. Rather than put a trademark symbol after every occurrence of a trademarked name, we use names in an editorial fashion only, and to the benefit of the trademark owner, with no intention of infringement of the trademark. Where such designations appear in this book, they have been printed with initial caps.

McGraw-Hill Education eBooks are available at special quantity discounts to use as premiums and sales promotions or for use in corporate training programs. To contact a representative, please visit the Contact Us page at www.mhprofessional.com.

Information has been obtained by McGraw Hill LLC from sources believed to be reliable. However, because of the possibility of human or mechanical error by our sources, McGraw Hill LLC, or others, McGraw Hill LLC does not guarantee the accuracy, adequacy, or completeness of any information and is not responsible for any errors or omissions or the results obtained from the use of such information.

TERMS OF USE

This is a copyrighted work and McGraw-Hill Education and its licensors reserve all rights in and to the work. Use of this work is subject to these terms. Except as permitted under the Copyright Act of 1976 and the right to store and retrieve one copy of the work, you may not decompile, disassemble, reverse engineer, reproduce, modify, create derivative works based upon, transmit, distribute, disseminate, sell, publish or sublicense the work or any part of it without McGraw-Hill Education's prior consent. You may use the work for your own noncommercial and personal use; any other use of the work is strictly prohibited. Your right to use the work may be terminated if you fail to comply with these terms.

THE WORK IS PROVIDED "AS IS." McGRAW-HILL EDUCATION AND ITS LICENSORS MAKE NO GUARANTEES OR WARRANTIES AS TO THE ACCURACY, ADEQUACY OR COMPLETENESS OF OR RESULTS TO BE OBTAINED FROM USING THE WORK, INCLUDING ANY INFORMATION THAT CAN BE ACCESSED THROUGH THE WORK VIA HYPERLINK OR OTHERWISE, AND EXPRESSLY DISCLAIM ANY WARRANTY, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. McGraw-Hill Education and its licensors do not warrant or guarantee that the functions contained in the work will meet your requirements or that its operation will be uninterrupted or error free. Neither McGraw-Hill Education nor its licensors shall be liable to you or anyone else for any inaccuracy, error or omission, regardless of cause, in the work or for any damages resulting therefrom. McGraw-Hill Education has no responsibility for the content of any information accessed through the work. Under no circumstances shall McGraw-Hill Education and/or its licensors be liable for any indirect, incidental, special, punitive, consequential or similar damages that result from the use of or inability to use the work, even if any of them has been advised of the possibility of such damages. This limitation of liability shall apply to any claim or cause whatsoever whether such claim or cause arises in contract, tort or otherwise.

This book is dedicated to everyone who has contributed to open source technologies and ideals in one form or another. Without you, we would have nothing to write about.

About the Author

Wale Soyinka is a system administrator, a DevOps/SecOps aficionado, an open source evangelist, and hacker.

He is also the author of several Linux, network, and Windows administration books, including *Linux Administration: A Beginner's Guide, Wireless Network Administration: A Beginner's Guide,* and *Advanced Linux Administration*.

When he is not marveling at his own culinary creations, Wale participates in discussions, projects, and ventures, usually focused on promoting and showcasing open source technologies and culture.

About the Contributor

Buki Adeniji is an AWS cloud infrastructure consultant and seasoned telephony and telecomms professional who is extremely passionate about cloud computing, Linux, VoIP, and Asterisk, and has many global deployments under his belt. Buki has experience building and managing high-availability carrier-grade VoIP platforms and production infrastructure. He has led global teams that managed mission-critical cloud-based services, achieving 99.99 percent ("four nines") service availability. Being a lifetime learner with advanced degrees in Telecomms, Business, and Project Management, as well as a number of AWS certifications, Buki's goal is to continue to excel at the boundaries of technology and innovation by using principles of ethnography and design thinking to develop technical solutions that address current and next-generation business needs. Above all, he is passionate about converting complex concepts and procedures into simple, concise, and executable steps. Buki currently resides in the Bay Area, where he is a passionate family man, as well as an active maker who participates in building and racing DIY self-driving model cars.

About the Technical Editor

David Lane is a DevOps/SRE manager with 30 years of experience in open source solutions development and implementation for the government, corporations, and associations. He discovered Linux via the initial Slackware distribution in the late 1990s and has never looked back. Today, David helps companies transition to DevOps methodologies from traditional software development practices with open source tooling. When he is not helping companies, he is an amateur radio operator and emergency coordinator for ARES, where he espouses open source solutions for Software Defined Radio (SDR) and other methods of communication during a communications emergency.

Contents at a Glance

Part I	**Introduction and Installation**	
1	Introduction to Linux, Distributions, and FOSS	3
2	Installing a Linux Server	15
3	Deploying Linux Servers in the Cloud	39
Part II	**Single-Host Administration**	
4	The Command Line	47
5	Managing Software	83
6	Managing Users and Groups	111
7	Booting and Shutting Down	137
8	File Systems	155
9	Core System Services	175
10	The Linux Kernel	201
11	Knobs and Dials: API (Virtual) File Systems	221
Part III	**Networking and Security**	
12	TCP/IP for System Administrators	235
13	Network Configuration	271
14	Linux Firewall (Netfilter)	299
15	Local Security	325
16	Network Security	341
Part IV	**Internet Services**	
17	Domain Name System (DNS)	355
18	File Transfer Protocol (FTP)	389
19	Apache Web Server	403
20	Simple Mail Transfer Protocol (SMTP)	419
21	Post Office Protocol and Internet Mail Access Protocol (POP and IMAP)	429
22	Voice over Internet Protocol (VoIP)	447
23	Secure Shell (SSH)	479

Part V **Intranet Services**

24 Network File System (NFS) 497
25 Samba 515
26 Distributed File Systems (DFS) 531
27 Lightweight Directory Access Protocol (LDAP) 541
28 Printing 559
29 Dynamic Host Configuration Protocol (DHCP) 573
30 Virtualization 583
31 Backups 599

Part VI **Appendixes**

A Creating a Linux Installer on Flash/USB Devices 613
B Demo Virtual Machine and Container 621

Index 637

Contents

Acknowledgments xxiii
Introduction xxiv

Part I Introduction and Installation

Chapter 1 Introduction to Linux, Distributions, and FOSS 3
Linux: The Operating System 3
Open Source Software and GNU: Overview 5
The GNU Public License 6
Upstream and Downstream 6
The Advantages of Open Source Software 8
Understanding the Differences Between Windows and Linux 9
Single Users vs. Multiple Users vs. Network Users 9
The Monolithic Kernel and the Micro-Kernel 10
Separation of the GUI and the Kernel 10
My Network Places 11
The Registry vs. Text Files 12
Domains and Active Directory 12
Summary 13

Chapter 2 Installing a Linux Server 15
Hardware and Environmental Considerations 15
Server Design 16
Uptime 17
Methods of Installation 17
Installing Fedora 18
Project Prerequisites 19
The Installation 20
Installation Summary 21
Localization Section 21
Software Section 21
System Section 22
User Settings Section 30
The Installation 31

Installing Ubuntu Server 32
Start the Installation 33
Configure the Network 34
Configure Proxy 34
Configure Ubuntu Archive Mirror 34
File System Setup 34
Profile Setup 36
SSH Setup 36
Featured Server Snaps 37
Install Complete 37
Summary 37

Chapter 3 Deploying Linux Servers in the Cloud 39
Behind the Cloud 39
Obtaining and Spinning Up New Virtual Linux Servers 40
Free-to-Run Virtual Linux Servers 40
Commercial Cloud Service Providers 41
Summary 44

Part II Single-Host Administration

Chapter 4 The Command Line 47
An Introduction to Bash 48
Job Control 48
Environment Variables 49
Pipes 51
Redirection 52
Command-Line Shortcuts 52
Filename Expansion 52
Environment Variables as Parameters 53
Multiple Commands 53
Backticks 53
Documentation Tools 54
The man Command 55
The texinfo System 56
Files (Types, Ownership, and Permissions) 56
Normal Files 56
Directories 56
Hard Links 57
Symbolic Links 57
Block Devices 57
Character Devices 57
Listing Files: ls 58
Change Ownership: chown 58

Change Group: chgrp 59
Change Mode: chmod. 59
File Management and Manipulation 61
Copy Files: cp 62
Move Files: mv 62
Link Files: ln 63
Find a File: find. 63
File Compression: gzip 64
File Compression: bzip2 65
File Compression: xz. 65
Create a Directory: mkdir 65
Remove Files or Directories: rm 66
Show Present Working Directory: pwd. 66
Tape Archive: tar 66
Concatenate Files: cat 68
Display a File One Screen at a Time: more or less. 69
Show the Directory Location of a File: which. 69
Locate a Command: whereis 69
Editors. 70
vi 70
emacs 70
pico. 71
sed. 71
Miscellaneous Tools 72
Disk Utilization: du 72
Disk Free: df 73
List Processes: ps 73
Show an Interactive List of Processes: top 75
Send a Signal to a Process: kill. 75
Show System Information: uname 77
Who Is Logged In: who. 77
A Variation on who: w 77
Switch User: su. 77
Putting It All Together (Moving a User and Its Home Directory) 78
Summary. 81
Chapter 5 Managing Software 83
The Red Hat Package Manager. 84
Managing Software Using RPM 86
Querying for Information the RPM Way
(Getting to Know One Another) 86
Installing Software with RPM (Moving in Together) 89
Uninstalling Software with RPM (Ending the Relationship) 92
Other Things RPM Can Do 93

Yum 96
DNF 96
GUI RPM Package Managers 97
Fedora or Ubuntu 97
openSUSE and SLE 98
The Debian Package Management System 99
APT 99
Software Management in Ubuntu 99
Querying for Information 100
Installing Software in Ubuntu 100
Removing Software in Ubuntu 101
Compile and Install GNU Software 103
Getting and Unpacking the Source Package 103
Looking for Documentation 104
Configuring the Package 105
Compiling the Package 106
Installing the Package 106
Testing the Software 107
Cleanup 107
Common Problems when Building from Source Code 108
Problems with Libraries 108
Missing Configure Script 109
Broken Source Code 109
Summary 109
Chapter 6 Managing Users and Groups 111
What Exactly Constitutes a User? 111
Where User Information Is Kept 112
The /etc/passwd File 112
The /etc/shadow File 116
The /etc/group File 117
User Management Tools 118
Command-Line User Management 118
GUI User Managers 122
Users and Access Permissions 123
Understanding SetUID and SetGID Programs 124
Sticky Bit 125
Pluggable Authentication Modules 125
How PAM Works 126
PAM's Files and Their Locations 126
Configuring PAM 127
A Sample PAM Configuration File 128
The "Other" File 131
D'oh! I Can't Log In! 131
Debugging PAM 131

A Grand Tour 131
Creating Users with useradd 132
Creating Groups with groupadd 133
Modifying User Attributes with usermod 133
Modifying Group Attributes with groupmod 134
Deleting Users and Groups with userdel and groupdel 134
Summary 135
Chapter 7 Booting and Shutting Down 137
Boot Loaders 137
GRUB Legacy 138
GRUB 2 139
Bootstrapping 144
The init Process 145
Systemd Scripts 145
Writing Your Own rc Script 146
Enabling and Disabling Services 150
Enabling a Service 150
Disabling a Service 150
Odds and Ends of Booting and Shutting Down 151
fsck! 151
Booting into Single-User (“Recovery”) Mode 152
Summary 153
Chapter 8 File Systems 155
The Makeup of File Systems 155
i-Nodes 155
Blocks 156
Superblocks 157
ext4 157
Btrfs 158
XFS 159
Managing File Systems 159
Mounting and Unmounting Local Disks 159
Using fsck 163
Adding a New Disk 165
Overview of Partitions 165
Traditional Disk and Partition Naming Conventions 165
Volume Management 166
Creating Partitions and Logical Volumes 168
Creating File Systems 172
Summary 174
Chapter 9 Core System Services 175
systemd 175
systemd’s Role 176
How systemd Works 176

xinetd . . . 181
The /etc/xinetd.conf File . . . 182
Examples: A Simple (echo) Service Entry . . . 186
The Logging Daemon . . . 187
rsyslogd . . . 188
systemd-journald (journald) . . . 195
The cron Program . . . 197
The crontab File . . . 197
Editing the crontab File . . . 198
Summary . . . 199
Chapter 10 The Linux Kernel . . . 201
What Exactly Is a Kernel? . . . 201
Finding the Kernel Source Code . . . 203
Getting the Correct Kernel Version . . . 203
Unpacking the Kernel Source Code . . . 204
Building the Kernel . . . 204
Preparing to Configure the Kernel . . . 205
Kernel Configuration . . . 206
Compiling the Kernel . . . 211
Installing the Kernel . . . 212
Booting the Kernel . . . 214
The Author Lied! It Didn't Work! . . . 215
Patching the Kernel . . . 216
Downloading and Applying Patches . . . 216
If the Patch Worked . . . 219
If the Patch Didn't Work . . . 219
Summary . . . 220
Chapter 11 Knobs and Dials: API (Virtual) File Systems . . . 221
What's Inside the /proc Directory? . . . 221
Tweaking Files Inside of /proc . . . 222
Some Useful /proc Entries . . . 223
Enumerated /proc Entries . . . 223
Common proc Settings and Reports . . . 225
SYN Flood Protection . . . 226
Issues on High-Volume Servers . . . 227
SysFS . . . 227
cgroupfs . . . 229
tmpfs . . . 230
tmpfs Example . . . 231
Summary . . . 231

Part III Networking and Security

Chapter 12 TCP/IP for System Administrators 235
The Layers 235
Packets 236
TCP/IP Model and the OSI Model 238
Headers 241
Ethernet 241
IP (IPv4) 243
TCP 246
UDP 249
A Complete TCP Connection 250
Opening a Connection 250
Transferring Data 251
Closing the Connection 252
How ARP Works 253
The ARP Header: ARP Works with Other Protocols, Too! 254
Bringing IP Networks Together 255
Hosts and Networks 255
Subnetting 256
Netmasks 257
Static Routing 258
Dynamic Routing with RIP 260
tcpdump Bits and Bobs 265
Reading and Writing Dumpfiles 265
Capturing More or Less per Packet 265
Performance Impact 265
Don't Capture Your Own Network Traffic 266
Troubleshooting Slow Name Resolution (DNS) Issues 266
IPv6 268
IPv6 Address Format 268
IPv6 Address Types 269
IPv6 Backward Compatibility 269
Summary 270

Chapter 13 Network Configuration 271
Modules and Network Interfaces 271
Network Device Configuration Utilities (ip, ifconfig, and nmcli) 272
Sample Usage: ifconfig, ip, and nmcli 273
Setting Up NICs at Boot Time 278
Managing Routes 284
Sample Usage: Route Configuration 285
Displaying Routes 287

A Simple Linux Router . . . 290
Routing with Static Routes . . . 290
VPCs, Subnets, IPs, and Route Configuration (AWS Cloud Example) . . . 292
VPCs and Subnets (AWS) . . . 293
Internet Gateways and Routing (AWS) . . . 293
Security Groups (AWS) . . . 294
Launch a Linux Server in Its Own Subnet (AWS) . . . 295
Hostname Configuration . . . 295
Summary . . . 297
Chapter 14 Linux Firewall (Netfilter) . . . 299
How Netfilter Works . . . 300
A NAT Primer . . . 301
Chains . . . 303
Installing Netfilter . . . 306
Enabling Netfilter in the Kernel . . . 306
Configuring Netfilter . . . 308
Saving Your Netfilter Configuration . . . 308
The iptables Command . . . 310
firewalld . . . 317
Cookbook Solutions . . . 320
Simple NAT: iptables . . . 320
Simple NAT: nftables . . . 321
Simple Firewall: iptables . . . 322
Summary . . . 324
Chapter 15 Local Security . . . 325
Common Sources of Risk . . . 326
SetUID Programs . . . 326
Unnecessary Processes . . . 329
Picking the Right Runlevel . . . 330
Nonhuman User Accounts . . . 330
Limited Resources . . . 330
Mitigating Risk . . . 332
chroot . . . 333
SELinux . . . 335
AppArmor . . . 336
Monitoring Your System . . . 337
Logging . . . 337
Using ps and netstat . . . 337
Watch That Space (Using df) . . . 338
Automated Monitoring . . . 338
Staying in the Loop (Mailing Lists) . . . 338
Summary . . . 339

Chapter 16 Network Security 341
TCP/IP and Network Security 341
The Importance of Port Numbers 341
Tracking Services 342
Using the netstat Command 343
Security Implications of netstat's Output 343
Binding to an Interface 344
Shutting Down Services 345
Shutting Down xinetd and inetd Services 346
Shutting Down Non-xinetd Services 346
Monitoring Your System 347
Making the Best Use of syslog 347
Monitoring Bandwidth with MRTG 348
Handling Attacks 348
Trust Nothing (and No One) 349
Change Your Passwords 349
Pull the Plug 349
Network Security Tools 349
nmap 350
Snort 350
Nessus and OpenVAS 350
Wireshark/tcpdump 350
Summary 351

Part IV Internet Services

Chapter 17 Domain Name System (DNS) 355
The Hosts File 356
How DNS Works 356
Domain and Host Naming Conventions 356
The Root Domain 357
Subdomain 358
The in-addr.arpa Domain 360
Types of Servers 361
Installing a DNS Server 362
Understanding the BIND Configuration File 364
The Specifics 364
Configuring a DNS Server 367
Defining a Primary Zone in the named.conf File 367
Defining a Secondary Zone in the named.conf File 368
Defining a Caching Zone in the named.conf File 369
DNS Records Types 369
SOA: Start of Authority 370
NS: Name Server 371

A and AAAA: Address Records 371
PTR: Pointer Record 372
MX: Mail Exchanger 372
CNAME: Canonical Name 373
RP and TXT: The Documentation Entries 373
Setting Up BIND Database Files 374
DNS Server Setup Walkthrough 375
The DNS Toolbox 379
host 379
dig 380
resolvectl 381
nslookup 382
whois 382
nsupdate 382
The rndc Tool 383
Configuring DNS Clients 383
The Resolver 383
Configuring the Client (Traditional) 386
Summary 386
Chapter 18 File Transfer Protocol (FTP) 389
The Mechanics of FTP 389
Client/Server Interactions 389
Obtaining and Installing vsftpd 391
Configuring vsftpd 391
Starting and Testing the FTP Server 392
Customizing the FTP Server 395
Setting Up an Anonymous-Only FTP Server 396
Setting Up an FTP Server with Virtual Users 397
Summary 401
Chapter 19 Apache Web Server 403
Understanding HTTP 403
Headers 404
Ports 405
Process Ownership and Security 405
Installing the Apache HTTP Server 406
Apache Modules 406
Starting Up and Shutting Down Apache 407
Starting Apache at Boot Time 408
Testing Your Installation 408
Configuring Apache 409
Creating a Simple Root-Level Page 409
Apache Configuration Files 410
Common Configuration Options 410

Troubleshooting Apache 416
Summary. 417
Chapter 20 Simple Mail Transfer Protocol (SMTP) 419
Understanding SMTP. 419
Rudimentary SMTP Details. 420
Security Implications 421
E-mail Components 421
Installing the Postfix Server 422
Installing Postfix via DNF in Fedora, CentOS, or RHEL 422
Installing Postfix via APT in Ubuntu 423
Configuring the Postfix Server 423
The main.cf File 424
Checking Your Configuration 426
Running the Server 426
Checking the Mail Queue 427
Flushing the Mail Queue 427
The newaliases Command 427
Making Sure Everything Works 427
Summary. 428
Chapter 21 Post Office Protocol and Internet Mail Access Protocol (POP and IMAP) 429
POP3 and IMAP Protocol Basics 431
Dovecot (IMAP and POP3 Server) 431
Installing Dovecot 432
Dovecot Configuration Files and Options 434
Configuring Dovecot. 434
Running Dovecot 439
Checking Basic POP3 Functionality. 439
Checking Basic IMAP Functionality 440
Other Issues with Mail Services 442
SSL/TLS Security. 442
Availability 445
Log Files. 445
Summary. 445
Chapter 22 Voice over Internet Protocol (VoIP) 447
VoIP Overview. 447
VoIP Server 447
Analog Telephone Adapter (ATA) 448
IP Phones 448
VoIP Protocols 448
VoIP Implementations 450
Asterisk 451
How Asterisk Works 451

Asterisk Installation 451
Starting and Stopping Asterisk 452
Understanding Asterisk Configuration Files and Structure 453
The Dialplan: extensions.conf 457
Modules: modules.conf 458
Asterisk Network, Port, and Firewall Requirements 459
Configuring the Local Firewall for Asterisk 459
Configuring the PBX 460
Local Extensions 461
Outside Connection (VoIP Trunking) 467
Trunking Using Twilio Elastic SIP Trunks 467
Asterisk Maintenance and Troubleshooting 475
Asterisk CLI Commands 475
Helpful CLI Commands 475
Common Issues with VoIP 476
Summary 478

Chapter 23 Secure Shell (SSH) 479
Understanding Public Key Cryptography 479
Key Characteristics 481
SSH Backstory (Versions) 481
OpenSSH and OpenBSD 482
Alternative Vendors for SSH Clients 482
Installing OpenSSH on RPM-Based Systems 483
Installing OpenSSH via APT in Ubuntu 484
Server Startup and Shutdown 486
SSHD Configuration File 487
Using OpenSSH 488
Secure Shell (ssh) Client Program 488
Secure Copy (scp) Program 492
Secure FTP (sftp) Program 493
Files Used by the OpenSSH Client 493
Summary 494

Part V Intranet Services

Chapter 24 Network File System (NFS) 497
The Mechanics of NFS 497
Versions of NFS 498
Security Considerations for NFS 499
Mount and Access a Partition 499
Enabling NFS in Fedora, RHEL, and CentOS 500
Enabling NFS in Ubuntu and Debian 501
The Components of NFS 501
Kernel Support for NFS 502

Configuring an NFS Server 503
The /etc/exports Configuration File 503
Configuring NFS Clients 506
The mount Command 506
Soft vs. Hard Mounts 508
Cross-Mounting Disks 508
The Importance of the intr Option 508
Performance Tuning 508
Troubleshooting Client-Side NFS Issues 509
Stale File Handles 509
Permission Denied 509
Sample NFS Client and NFS Server Configuration 510
Common Uses for NFS 512
Summary 513
Chapter 25 Samba 515
The Mechanics of SMB 516
Usernames and Passwords 516
Encrypted Passwords 516
Samba Daemons 517
Installing Samba via RPM 518
Installing Samba via APT 518
Samba Administration 518
Starting and Stopping Samba 519
Creating a Share 519
Using smbclient 521
Mounting Remote Samba Shares 524
Samba Users 524
Creating Samba Users 525
Allowing Null Passwords 525
Changing Passwords with smbpasswd 526
Using Samba to Authenticate Against a Windows Server 526
winbindd Daemon 526
Troubleshooting Samba 529
Summary 529
Chapter 26 Distributed File Systems (DFS) 531
DFS Overview 531
DFS Implementations 534
GlusterFS 536
Summary 540
Chapter 27 Lightweight Directory Access Protocol (LDAP) 541
LDAP Basics 541
LDAP Directory 542
Client/Server Model 543

Uses of LDAP 543
LDAP Terminology 543
OpenLDAP 544
Server-Side Daemons 544
OpenLDAP Utilities 544
Installing OpenLDAP 545
Configuring OpenLDAP 546
Configuring slapd 548
Starting and Stopping slapd 551
Configuring OpenLDAP Clients 552
Creating Directory Entries 553
Searching, Querying, and Modifying the Directory 554
Using OpenLDAP for User Authentication 555
Configuring the Server 555
Configuring the Client 557
Summary 558
Chapter 28 Printing 559
Printing Terminologies 559
The CUPS System 560
Running CUPS 560
Installing CUPS 561
Configuring CUPS 562
Adding Printers 563
Local Printers and Remote Printers 564
Using the Web Interface to Add a Printer 565
Using Command-Line Tools to Add a Printer 567
Routine CUPS Administration 568
Setting the Default Printer 568
Enabling, Disabling, and Deleting Printers 568
Accepting and Rejecting Print Jobs 568
Managing Printing Privileges 569
Managing Printers via the Web Interface 569
Using Client-Side Printing Tools 570
lpr 570
lpq 571
lprm 571
Summary 572
Chapter 29 Dynamic Host Configuration Protocol (DHCP) 573
The Mechanics of DHCP 573
The DHCP Server 574
Installing DHCP Software via RPM 574
Installing DHCP Software via APT in Ubuntu 575

Configuring the DHCP Server 575
A Sample dhcpd.conf File. 579
The DHCP Client Daemon 581
Summary. 582
Chapter 30 Virtualization 583
Why Virtualize? 583
Virtualization Concepts 584
Virtualization Implementations 585
Hyper-V 585
Kernel-Based Virtual Machine (KVM) 585
QEMU 585
VirtualBox 586
VMware 586
Xen 586
KVM 586
KVM Example 587
Managing KVM Virtual Machines 590
Setting Up KVM in Ubuntu/Debian 591
Containers 594
Containers vs. Virtual Machines. 594
Docker 594
Summary. 598
Chapter 31 Backups 599
Evaluating Your Backup Needs. 599
Amount of Data 600
Backup Hardware and Backup Medium 600
Network Throughput 601
Speed and Ease of Data Recovery. 601
Data Deduplication 602
Tape Management. 603
Command-Line Backup Tools 605
dump and restore. 605
tar 608
rsync. 609
Miscellaneous Backup Solutions 609
Summary. 609

Part VI Appendixes

A Creating a Linux Installer on Flash/USB Devices 613
Overview 613
Native Solutions. 613
Distro-Specific Solutions 616
Universal Solutions 616

B Demo Virtual Machine and Container 621
Basic Host System Requirements 622
Installing the Virtualization Applications and Utilities 623
Download and Prep the Demo VM Image File 623
Import the Demo VM Image and Create a New VM Instance 625
Managing the Demo Virtual Machine 625
Connecting to the Demo VM 628
Virtual Network Computing (VNC) 628
Virtual Serial TTY Console 630
Connecting via SSH 631
Cockpit Application 632
Just Use It! 633
Demo Containers (Docker, podman, buildah, and kubectl) 634
Feedback 636

Index 637

Acknowledgments

I would like to thank Wendy Rinaldi, my editor, David Lane, Emily Walters, Patty Mon, Bart Reed, Tricia Lawrence, Claire Yee, and the McGraw-Hill crew for the magic that they do. Hats off to the folks at Cenveo for carrying us across the finish line.

I would never have gone far on this journey without the genius/work of thousands of strangers who contribute to FOSS as well the groundwork of individuals like Linus Torvalds and Richard Stallman. Thank you! Many thanks to Daniel A. for that first RedHat CD, the Berkeley UNIX Crew, Gene Schultz, Buki Adeniji, and Steve Shah for his foundational work.

I would like to thank Elédùmarè for my many adventures. I thank my mom and dad, my uncles, aunts, brothers, sisters, cousins, and my Bimpe. I am grateful to my friends for the gift of knowing them. I am forever thankful to Indian Baby and Hiromi who do little things that mean everything to me.

Introduction

On October 5, 1991, Linus Torvalds introduced the first cut of the Linux operating system to the world by posting a simple message to an online newsgroup. Unbeknownst to him (and anybody else), he had unleashed what was to become one of the world's most popular and disruptive operating systems. Today, entire industries and communities have been built and thrive around Linux. And, chances are, you've probably already used it (or benefited from it) in one form or another! This book is our eighth (and we hope successful) stab at continuing this tradition of openness and exploring and learning. Success for us is when you, the reader, can confidently take something learned from this book and apply it to solve a real-world problem.

Who Should Read This Book

The last part of the title of this book reads "A Beginner's Guide," and this is mostly apt. But what the title should say is, "A Beginner to Linux Server Administration Guide," because we do make a few assumptions about you, the reader. (However, we jolly well couldn't use that title because it's such a mouthful and isn't sexy enough.) We assume that you are already familiar with Microsoft Windows servers (or other operating systems) at a "power user" level or better. We assume that you are familiar with the terms (and some concepts) used in relation to small- to medium-sized networks of computers. Any experience with bigger networks or advanced Windows technologies will allow you to get more from the book but isn't required.

We make these assumptions because we did not want to write a guide for dummies. There are already enough books on the market that tell you what to click without telling you why; this book is not meant to be among those ranks.

In addition to your Windows background, we assume you're interested in having more information about the topics we discuss than just the material we have written. After all, we spend only 15 to 35 pages on some topics that have entire books devoted to them! For this reason, we have scattered pointers to other resources throughout the chapters. We urge you to take advantage of these recommendations.

We believe that seasoned Linux system administrators can also benefit from this book because it can serve as a quick how-to cookbook on various topics that might not be some seasoned readers' strong points. We understand that system administrators generally have aspects of system administration that they like or loath a lot. For example, making backups is not one of our favorite aspects of system administration, and this is reflected in the half a page we've dedicated to backups. (Just kidding. There's an entire chapter on the topic.)

What's in This Book?

Linux Administration: A Beginner's Guide, Eighth Edition is composed of six parts.

Part I: Introduction and Installation

Part I includes three chapters. Chapter 1, "Introduction to Linux, Distributions, and FOSS," provides a nice overview of what GNU/Linux is and how it compares to Windows in several key areas. Chapter 2, "Installing Linux in a Server," details how to install server-grade Fedora and Ubuntu Linux distributions. And Chapter 3, "Deploying Linux Servers in the Cloud," gives an overview of how to spin up Linux servers on some popular cloud platforms.

Ideally, the information in Part I should be enough to get you started and help you draw parallels to how Linux works based on your existing knowledge of other operating systems. Some of the server installation tasks and design choices made in Part I help to serve as a reference and jumping-off point for some other parts of the book.

Part II: Single-Host Administration

Part II covers the material necessary to manage what we call a "stand-alone" system—a system that does not require or provide any services to other systems on the network. Although the notion of a server *not* serving anything might seem counterintuitive at first, it is the foundation on which many other concepts are built, and it will come in handy for your understanding of network-based services later on. Part II contains eight chapters.

Chapter 4, "The Command Line," covers the basics of working with the Linux command-line interface (CLI) so that you can become comfortable working without a GUI. Although it is possible to administer a Linux system from within the graphical (GUI) desktop, you'll probably be best served by being comfortable with both the CLI and the GUI—and this is true for administering Windows systems too. Don't believe it? Open a command prompt, run **netsh**, and try to do what **netsh** does in the GUI.

Chapter 5, "Managing Software," covers how to install software from prepackaged binaries and source code, as well as how to perform standard software management tasks. Some of the software installation techniques you learn and use in this chapter are used throughout most of the book.

Chapter 6, "Managing Users and Groups," covers the underlying basics of user and group concepts on Linux platforms, as well as the day-to-day management tasks of adding and removing users and groups, multiuser operations, the Linux permissions model, and more.

Chapter 7, "Booting and Shutting Down," documents the entire booting and shutting-down processes. This chapter includes details on how to properly start up, shut down, and add, enable, and disable services. These basic skills will come in handy later on in the book.

Chapter 8, "File Systems," continues with the basics of file systems—their organization, their creation, and, most important, their management.

The discussion of operation basics continues in Chapter 9, "Core System Services," with coverage of basic tools such as **xinetd**, **rsyslog**, **cron**, **systemd**, **journald**, and more. **xinetd** is the Linux equivalent of Windows' **svchost**, and **rsyslog** manages logging for all applications in a unified framework. You might think of **rsyslog** and **journald** as more flexible versions of the Windows Event Viewer.

Chapter 10, "The Linux Kernel," documents the process of configuring, compiling, and installing your own custom kernel in Linux. This capability is one that gives Linux administrators an extraordinary amount of fine-grained control over how their systems operate.

Chapter 11, "Knobs and Dials: API (Virtual) File Systems," covers some kernel-level tweaking through the **/proc** and **/sys** file systems. The ability to view and modify certain kernel-level configuration and runtime variables through **/proc** and **/sys**, as shown in this chapter, gives administrators almost infinite system fine-tuning possibilities. When these are applied properly, it amounts to an arguably better and easier way to tweak the kernel than is afforded by Windows platforms.

Part III: Networking and Security

Part III begins our journey into the world of networking and security. We are not quite sure why we combined these two topics together under the same part, but if forced to justify this, we would say, "The network is the root of most evils and therefore it needs to be secured." And then we would shrug and walk away.

Moving right along: with the ongoing importance of security and privacy on the Internet, as well as the compliance requirements of Sarbanes-Oxley (SOX), the Health Insurance Portability and Accountability Act (HIPAA), and so on, the use of Linux in scenarios that require high security is unprecedented. We deliberately decided to cover security before introducing network-based services (Parts IV and V) so that we could touch on some essential security best practices that can help in protecting your network-based services from attacks.

This part of the book kicks off with Chapter 12, "TCP/IP for System Administrators," which provides a detailed overview of TCP/IP in the context of what system (and/or network) administrators need to know. The chapter provides a lot of detail on how to use troubleshooting tools such as **tcpdump** to capture packets and read them back, as well as a step-by-step analysis of how TCP connections work. These tools should enable you to troubleshoot network peculiarities effectively.

Chapter 13, "Network Configuration," returns to administration issues by focusing on basic network configuration (for both IPv4 and IPv6). This includes setting up IP addresses, routing entries, and so on. We extend past the basics in Chapter 14, "Linux Firewall (Netfilter)," by delving into advanced networking concepts and showing you how to build a Linux-based firewall and router. We touch on the new **nftables** project, which is slated to replace the existing and popular **iptables** framework.

Chapter 15, "Local Security," and Chapter 16, "Network Security," discuss aspects of system and network security in detail. They include Linux-specific issues as well as general security tips and tricks and commonsensical stuff so that you can better configure your system and protect it against attacks.

Part IV: Internet Services

The remainder of the book is divided into two distinct parts: "Internet Services" and "Intranet Services." Although they sound similar, they are different—In*TER*net and in*TRA*net. We define Internet services as those running on a Linux system exposed directly to the Internet. Examples include web and Domain Name System (DNS) services. We define intranet services as those that are typically run behind a firewall for internal users and are mostly for internal consumption only.

This section starts off with Chapter 17, "Domain Name System (DNS)," which covers the information you need to know to install, configure, and manage a DNS server. In addition to the actual details of running a DNS server, we provide a detailed background on how DNS works and several troubleshooting tips, tricks, and tools.

From DNS, we move on to Chapter 18, "File Transfer Protocol (FTP)," which covers the installation and care of FTP servers. This chapter also includes a background on FTP and some notes on its evolution.

Chapter 19, "Apache Web Server," moves on to what may be considered one of the most popular uses of Linux today: running a web server with the popular Apache software. This chapter covers installing, configuring, and managing the Apache web server.

Chapter 20, "Simple Mail Transfer Protocol (SMTP)," and Chapter 21, "Post Office Protocol and Internet Mail Access Protocol (POP and IMAP)," dive into e-mail through the setup and configuration of SMTP, POP, and IMAP servers. We cover the information needed to configure all three, as well as show how they interact with one another. We have chosen to cover the Postfix SMTP server instead of the classic Sendmail server, because Postfix provides a more flexible server with a better security record.

Chapter 22, "Voice over Internet Protocol (VoIP)," is an all-new revised contribution to this edition. A lot of work and effort went into brewing and distilling the very extensive topic of VoIP into just over 32 pages of this chapter. We think it was well worth it, and at the end of this chapter we build a simple VoIP-based PBX based on Asterisk software that can easily be extended to replace commercial-grade PBX solutions, make phone calls among local extensions, and interface with third-party VoIP providers to communicate with the rest of the world.

Part IV ends with Chapter 23, "Secure Shell (SSH)." Knowing how to set up, use, and manage the SSH service is essential and paramount in almost any GNU/Linux environment.

Part V: Intranet Services

Again, we define intranet services as those that are typically run behind a firewall for internal users and mostly for internal consumption only. Even in this environment, Linux has a lot to offer. Part V starts off with Chapter 24, "Network File System (NFS)." NFS has been around for almost 30 years now and has evolved, grown, and adapted over the years. This chapter covers NFS capabilities, including how to set up both clients and servers as well as how to troubleshoot.

Chapter 25, "Samba," continues the idea of sharing disks and resources with coverage of the Samba service. Using Samba, administrators can share disks and printing facilities and provide authentication for Windows (and Linux) users without having to install any special client software. Thus, Linux can become an effective server, able to support and share resources

between UNIX/Linux systems as well as Windows systems. If you are into that *sort* of thing, Samba can even be configured to serve as a drop-in replacement for full-fledged Active Directory Microsoft Windows servers! Including Chapter 26, "Distributed File Systems (DFS)," in Part V instead of Part IV came down to a coin toss, because DFS can be used/deployed in both Internet- and intranet-facing scenarios. DFS solutions are especially important and relevant in today's cloud-centric world. Among the many DFS implementations available, we have selected to cover GlusterFS because of its ease of configuration and cross-distribution support.

We discuss directory services in Chapter 27, "Lightweight Directory Access Protocol (LDAP)," with coverage of LDAP and how administrators can use this standard service for providing a centralized user database (directory) for use among heterogeneous operating systems and also for managing tons of users.

Chapter 28, "Printing," takes a tour of the Linux printing subsystem. The printing subsystem, when combined with Samba, allows administrators to support seamless printing from Windows-based clients. The result is a powerful way of centralizing printing options for Linux, Windows, and even macOS users on a single server!

Chapter 29, "Dynamic Host Configuration Protocol (DHCP)," covers another common use of Linux systems: DHCP servers. This chapter discusses how to deploy the Internet Systems Consortium (ISC) DHCP server, which offers a powerful array of features and options.

Then comes Chapter 30, "Virtualization." Over several months, we agonized and struggled over what title to use for this chapter, because we cover both virtualization and containers (containerization). We ended up naming the chapter simply "Virtualization" because we are not even sure if "containerization" is a word. (According to a copy editor, it is a word in the *shipping* industry!) Virtualization is everywhere. It allows companies to consolidate services and hardware that previously required several dedicated bare-metal machines into much fewer bare-metal machines. We discuss the basic virtualization concepts and briefly cover some of the popular virtualization technologies in Linux. The chapter also covers the Kernel-Based Virtual Machine (KVM) implementation in detail, with examples. The KVM concepts that we discuss will help prepare you for the brand-new goodies that we have in store for you in Appendix B. The tail end of Chapter 30 features containers. The concepts behind containers are *old* but *new* again. Like virtualization, containers are everywhere … and here to stay. We use the popular Docker platform for our container implementation and walk you through the process of how to deploy a web server, container-style!

They say that all good things must come to an end, and that even applies to this book! The final chapter is Chapter 31, "Backups." Backups are arguably one of the most critical pieces of administration. Linux-based systems support several methods of providing backups that are easy to use and readily usable by tape drives and other media. The chapter discusses some of the methods and explains how they can be used as part of a backup schedule. In addition to the mechanics of backups, we discuss general backup considerations, design, and optimization.

Part VI: Appendixes

At the end of the book, we include some useful reference materials and real-world resources that you can use today and every day at work, at home, in the classroom, or in the lab.

Appendix A, "Creating a Linux Installer on Flash/USB Devices," details alternate and generic methods for creating an installation media on nonoptical media, such as a USB flash drive, SD card, and so on.

Appendix B, "Using the Demo Virtual Machine Images and Containers," is another brand-new addition to this book. It covers one of the new features we've added to this eighth edition: how to obtain and use the purpose-built container(s) and virtual machine (VM) image file that we created especially for you as an accompaniment to this book. Once you power up the VM or container, you'll see most of the commands, scripts, software packages, hacks, and servers/daemons that we discuss throughout this book. If you look carefully, you might even find some of our very own sysadmin *mistakes* enshrined in these resources.

Typographic Conventions

We were very ambitious in aiming for this book to be the one-stop Linux administration resource for many of the mainstream Linux distros. But we didn't stop there! We also wanted to reflect the subtle nuances and idiosyncrasies that sometimes exist between mainstream distros. To help with this, we had to adopt some unique conventions throughout this book. Some of these conventions appear in code listings, commands, command output, and command prompts.

Commands that are meant to be typed out by the reader are in **bold Courier font**. For example:

```
command --options foobar
```

The output of commands is in plain Courier font:

```
This is a sample output
```

When we intend to reference or explain the output of commands that span multiple lines, we may sometimes number the output of such commands (also in Courier font). For example:

```
1. This is line 1 of a sample command output
2. This is line 2 of a sample command output
```

Some Linux commands and their options and output can be rather long. For these situations we place the backslash symbol (\) at the end of each line (except the final line) to indicate that the command spans multiple lines. For example, for a command with multiple options that spans multiple lines, we would show this as follows:

```
command --option1 foobar1 --option2 foobar2 \
--option3 foobar3 --option4 foobar4 --option4 foobar4 \
--option5 foobar5
```

For distribution-independent commands, the command prompt will not have any particular distribution name. For example:

```
[root@server  ~]#
```

For commands that target a specific Linux distribution, the command prompt will incorporate the distribution name. For example, for a Fedora-based server, the command prompt will resemble this:

```
[root@fedora-server  ~]#
```

And for an Ubuntu-based server, the command prompt will resemble the following:

```
root@ubuntu-server: ~#
```

Debian-based GNU/Linux distributions and other modern distros (such as Fedora, RHEL, and CentOS) often do not directly use the superuser (*root*) or administrator account for administering the system. Regular user accounts with elevated privileges are used for this purpose. To achieve this, most commands are often prefixed with the **sudo** command on such platforms. Consider a regular user named *master* with **sudo** privileges who wants to run a restricted command. In this case, the command prompt and command prefixed with **sudo** will resemble this:

```
master@ubuntu-server:~$   sudo   command  --options
```

For projects or exercises that are meant to be run from a client system, the command prompt might appear as follows:

```
user@client  ~$
```

For projects or exercises that are best suited for multiserver scenarios, when you absolutely need more than one server, the command prompt will change to reflect the steps or commands to be executed on the particular server. For example, the prompt for the first server (Server-A) might resemble

```
[root@server-A  ~]#
```

and the prompt for the second server (Server-B) might resemble

```
[root@server-B  ~]#
```

Updates and Feedback

Although we hope we've published a book with no errors, we have set up an errata list for this book at www.linuxserverexperts.com. If you find any errors, we welcome your submissions for errata updates. We also welcome your feedback and comments. Unfortunately, our day jobs prevent us from answering detailed questions, so if you're looking for help on a specific issue, you may find one of the many online communities a useful resource. However, if you have two cents to share about the book, we welcome your thoughts. You can send us an e-mail to feedback@linuxserverexperts.com.

PART

I

Introduction and Installation

CHAPTER 1
Introduction to Linux, Distributions, and FOSS

CHAPTER 2
Installing a Linux Server

CHAPTER 3
Deploying Linux Servers in the Cloud

This page intentionally left blank

CHAPTER

1 Introduction to Linux, Distributions, and FOSS

In this chapter we'll look at some of the core server-side technologies as they are implemented in the Linux (open source) world. And where applicable, we compare and contrast this to the Microsoft Windows Server world (possibly the platform you are more familiar with). But before delving into any technicalities, we briefly discuss some important underlying concepts, ideas, and ideologies that form the genetic makeup of Linux and Free and Open Source Software (FOSS).

Linux: The Operating System

Some people (mis)understand Linux to be an entire software suite of developer tools, editors, graphical user interfaces (GUIs), networking tools, and so forth. More formally and correctly, such software *collectively* is called a *distribution,* or *distro.* The distro is the entire software suite, including Linux, that forms the whole.

So if we consider a distribution everything plus Linux, what then *is* Linux exactly? Linux itself is the core of the operating system: the *kernel* or, more colloquially, *the heart.* The kernel is the program acting as chief of operations. It is responsible for starting and stopping other programs (such as text editors, web browsers, services, and so on), handling requests for memory, accessing disks, and managing network connections. The complete list of kernel activities could easily fill a book in itself, and, in fact, several books documenting the kernel's internal functions have been written.

The kernel is a nontrivial program. It is also what puts the Linux badge on all the numerous Linux distributions. All distributions use essentially a version of the Linux kernel, so the fundamental behavior of all Linux distributions is the same.

You've probably heard of the Linux distributions named Red Hat Enterprise Linux (RHEL), Fedora, Debian, Amazon Linux, Ubuntu, Mint, openSUSE, CentOS, Arch, Chrome OS, Slackware, Oracle Linux, and so on, which have received a great deal of press.

Depending on whom you are speaking with, Linux distributions can be categorized along a variety of different lines, including software management style, cultural, commercial,

noncommercial, philosophical, and function. One popular taxonomy for categorizing Linux distros is commercial versus noncommercial.

The vendors of commercial distros generally offer support for their distributions—at a cost. The commercial distros also tend to have a longer release life cycle. To meet certain regulatory requirements, some commercial distros may incorporate/implement more specific security requirements that the FOSS community might not care about but that some institutions/corporations do care about. Examples of commercial flavors of Linux-based distros are RHEL and SUSE Linux Enterprise (SLE).

The noncommercial distros, on the other hand, are free. These distros try to adhere to the original spirit of open source software. They are mostly community supported and maintained—the community consists of the users and developers. The community support and enthusiasm can sometimes supersede that provided by the commercial offerings!

Several of the so-called noncommercial distros also have the backing and support of their commercial counterparts. Very often, the companies that offer the purely commercial flavors have vested interests in making sure that free distros exist. Some of the companies use the free distros as the proving and testing ground for software that ends up in the commercial spins. This is a sort of *freemium* model. Examples of noncommercial flavors of Linux-based distros are Fedora, openSUSE, Ubuntu, Linux Mint, Gentoo, Raspbian, and Debian.

An interesting thing to note about the commercial Linux distributions is that most of the programs with which they ship were not written by the companies themselves! Rather, other people have (freely) released their programs with licenses, allowing their redistribution with source code. The distribution vendors simply bundle them into one convenient and cohesive package that's easy to install. In addition to bundling existing software, several distribution vendors also develop value-added tools that make their distributions easier to administer or use, but the software that they ship is generally written by others.

Kernel Differences

Each company that sells a Linux distribution of its own will be quick to tell you that its kernel is better than the others. How can a company make this claim? The answer comes from the fact that each company maintains its own patch set. To make sure that the kernels largely stay in sync, most companies do adopt patches that are posted on www.kernel.org, the "Linux Kernel Archives." Vendors, however, typically do not track the release of every single kernel version that is released onto www.kernel.org. Instead, they take a foundation, apply their custom patches to it, run the kernel through their quality assurance (QA) process, and then mark it production ready. This helps other commercial customers have confidence that these kernels have been sufficiently baked, thus mitigating any perceived risk of running pure open source–based operating systems.

One exception to this rule revolves around security issues. If a security issue is found with a version of the Linux kernel, vendors are quick to adopt the necessary patches to fix the problem immediately. Thankfully, exploits against the kernel itself are few and far between.

(*continued*)

So if each vendor maintains its own patch set, what exactly is it patching? This answer varies from vendor to vendor, depending on each vendor's target market. Red Hat, for instance, is largely focused on providing enterprise-grade reliability and solid efficiency for application servers. This might be different from the mission of the Fedora team, which is more interested in trying new technologies quickly. This, in turn, is different from the approach of a vendor that is trying to put together a desktop-oriented or multimedia-focused Linux system, and even much more different from a vendor using Linux for an embedded application.

What separates one distribution from the next is the value-added tools that come with each one. Asking which distribution is better is much like asking whether hot or cold water is better. Pure water almost always has the same basic elements—hydrogen and oxygen—and has the universal ability to quench thirst. So the answer to the question about whether hot or cold water is better is, "It depends." Are you using it to make hot tea or iced tea? Are you using it to bathe in? In the end, choosing a distribution comes down to your requirements: Do you need commercial support? Does your application vendor recommend one distribution over another? Do you have more in-house expertise for one distro over the others? When you review your requirements, you'll find that there is likely a distribution that is geared toward your exact needs.

Open Source Software and GNU: Overview

In the early 1980s, Richard Matthew Stallman began a movement within the software industry. He preaches that software should be free. Note that by *free* he doesn't mean in terms of price but rather in the same sense as *freedom* or *libre.* This means shipping not just a product but the entire source code as well. To clarify the meaning of free software, Stallman was once famously quoted as saying:

> *"Free software" is a matter of liberty, not price. To understand the concept, you should think of "free" as in "free speech," not as in "free beer."*

Stallman's position was, somewhat ironically, a return to classic computing, when software was freely shared among hobbyists!

The premise behind giving away source code is simple: A user of the software should never be forced to deal with a developer who might not support that user's intentions for the software. The user should never have to wait for bug fixes to be published. More important, code developed under the scrutiny of other programmers is typically of higher quality than code written behind locked doors. One of the great benefits of open source software comes from the users themselves: If they have the know-how, they can add new features to the original program and then contribute these features to the source so that everyone else can benefit from them.

This basic desire lead to the release of a complete UNIX-like system (aka Linux) to the public, free of license restrictions. Of course, before you can build any operating system, you need to build tools, and this is how the GNU project and its namesake license were born. The tight symbiotic relationship between the GNU project and the Linux Kernel project is one of the reasons why you would often see the complete stack written as ***GNU/Linux***.

NOTE GNU stands for GNU's Not UNIX. This is an example of a recursive acronym, which is a type of hacker humor. If you don't understand why it's funny, don't worry. You're still in the majority.

The GNU Public License

One important thing to emerge from the GNU project is the GNU Public License (GPL). This license explicitly states that the software being released is free and that no one can ever take away these freedoms. It is acceptable to take the software and resell it, even for a profit; however, in this resale, the seller must release the full source code, including any changes. Because the resold package remains under the GPL, the package can be distributed for free and resold yet again by anyone else for a profit. Of primary importance is the liability clause: The programmers are not liable for any damages caused by their software.

It should be noted that the GPL is not the only license used by open source software developers (although it is arguably the most popular). Other licenses, such as BSD and Apache, have similar liability clauses but differ in terms of their redistribution. For instance, the BSD license allows people to make changes to the code and ship those changes without having to disclose the added code (whereas the GPL requires that the added code is shipped). For more information about other open source licenses, check out www.opensource.org.

Historical Footnote

Many, many moons ago, Red Hat started a commercial offering of its erstwhile free product (Red Hat Linux). The commercial release gained steam with the Red Hat Enterprise Linux (RHEL) series. Because the foundation for RHEL is GPL, individuals interested in maintaining a free version of Red Hat's distribution have been able to do so. Furthermore, as an outreach to the community, Red Hat created the Fedora Project, which is considered the testing grounds for new software before it is adopted by RHEL. The Fedora Project is freely distributed and can be downloaded from http://fedoraproject.org or https://getfedora.org.

Upstream and Downstream

Upstream [developer, code, project] and downstream [developer, code, project] are terms you might come across frequently in the FOSS world. To help you understand the concept of upstream and downstream components, let's start with an analogy. Picture, if you will, a pizza with all your favorite toppings.

The pizza is put together and baked by a local *pizza shop.* Several things go into making a great pizza—cheeses, vegetables, flour (dough), herbs, meats (or meat substitutes), and sauces,

to mention a few. The pizza shop will often make some of these ingredients in-house and rely on other businesses to supply other ingredients. The pizza shop is also tasked with assembling the ingredients into a complete, finished pizza.

Let's consider one of the most common pizza ingredients—cheese. The cheese is made by a *cheesemaker* who makes her cheese for many other industries or applications, including the pizza shop. The cheesemaker is pretty set in her ways and has very strong opinions about how her product should be paired with other food stuffs (wine, crackers, bread, vegetables, and so on). The pizza shop owners, on the other hand, do not care about other food stuffs—they care only about making a great pizza. Sometimes the cheesemaker and the pizza shop owners will bump heads due to differences in opinion and objectives. And at other times they will be in agreement and cooperate beautifully. Ultimately (and sometimes unbeknownst to them), the pizza shop owners and the cheesemaker care about the same thing: producing the best product they can.

The pizza shop in our analogy here represents the Linux distributions' vendors/projects (Fedora, Debian, RHEL, openSUSE, and so on). The cheesemaker represents the different software project maintainers that provide the important programs and tools, such as the Bourne Again Shell (Bash), GNU Image Manipulation Program (GIMP), GNOME, KDE, Nmap, LibreOffice, and GNU Compiler Collection (GCC), that are packaged together to make a complete distribution (the pizza). The Linux distribution vendors are referred to as the *downstream* component of the open source food chain; the maintainers of the accompanying different software projects are referred to as the *upstream* component.

Standards

Very early on, the FOSS and Linux community understood that it needed a formal method and standardization process for how certain things should be done among the numerous Linux spins. As a result, two major standards are actively being worked on.

The *Filesystem Hierarchy Standard (FHS)* is an attempt by many of the Linux distributions to standardize on a directory layout so that developers have an easy time making sure their applications work across multiple distributions without difficulty. As of this writing, several major Linux distributions have become completely compliant with this standard.

The *Linux Standard Base (LSB)* specification is *written by* a standards group that specifies what a Linux distribution should have in terms of libraries and tools.

A developer who assumes that a Linux machine complies only with LSB and FHS is almost guaranteed to have an application that will work with all compliant Linux installations. All of the major distributors have joined these standards groups. This should ensure that all desktop distributions will have a certain amount of commonality on which a developer can rely.

From a system administrator's point of view, these standards are interesting but not crucial to administering a Linux environment. However, it never hurts to learn more about both. For more information on the FHS, go to http://refspecs.linuxfoundation.org/fhs.shtml. To find out more about LSB, check out www.linuxfoundation.org/collaborate/workgroups/lsb.

The Advantages of Open Source Software

If the GPL seems like a bad idea from a commercial standpoint, consider the surge in adoption of successful open source software projects—this is indicative of a system that does indeed work! This success has evolved for two reasons. First, as mentioned earlier, errors in the code itself are far more likely to be caught and quickly fixed under the watchful eyes of peers. Second, under the GPL system, programmers can release code without the fear of being sued. Without that protection, people might not feel as comfortable releasing their code for public consumption.

NOTE The concept of free software, of course, often begs the question of why anyone would release his or her work for free. As hard as it might be to believe, some people do it purely for altruistic reasons and the love of it.

Most projects don't start out as full-featured, polished pieces of work. They often begin life as a quick hack to solve a specific problem bothering the programmer at the time. As a quick-and-dirty hack, the code might not have a sales value. But when this code is shared and consequently improved upon by others who have similar problems and needs, it becomes a useful tool. Other program users begin to enhance the code with features they need, and these additions travel back to the original program. The project thus evolves as the result of a group effort and eventually reaches full refinement. This polished program can contain contributions from possibly hundreds, if not thousands, of programmers who have added little pieces here and there. In fact, there may be little evidence remaining of original author's code.

There's another reason for the success of generously licensed software. Any project manager who has worked on commercial software knows that the *real* cost of developing software isn't only in the development phase—it's also in the cost of selling, marketing, supporting, documenting, packaging, and shipping that software. A programmer carrying out a weekend hack to fix a problem with a tiny, kludged together program might lack the interest, time, and money to turn that hack into a profitable product.

When Linus Torvalds released Linux in 1991, he released it under the GPL. As a result of its open charter, Linux has had a notable number of contributors and analyzers. This participation has made Linux strong and rich in features. It is estimated that since the v.2.2.0 kernel, Torvalds's contributions represent less than 2 percent of the total code base!

NOTE This might sound strange, but it is true: Contributors to the Linux kernel code include companies with competing operating system platforms. For example, Microsoft was one of the top code contributors to the Linux version 3.0 kernel code base (as measured by the number of changes or patches relative to the previous kernel version). Even though this might have been for self-promoting reasons on Microsoft's part, the fact remains that the open source licensing model that Linux adopts permits this sort of thing to happen. Everyone and anyone who knows how, can contribute code. The code is subjected to a peer review process, which in turn helps the code to benefit from the "many eyeballs" axiom. In the end, everyone (end users, companies, developers, and so on) benefits.

Because Linux is free (as in speech), anyone can take the Linux kernel and other supporting programs, repackage them, and resell them. A lot of people and corporations have made money with Linux doing just this! As long as these folks release the kernel's full source code

along with their individual packages, and as long as the packages are protected under the GPL, everything is legal. Of course, this also means that packages released under the GPL can be resold by other people under other names for a profit.

In the end, what makes a package from one person more valuable than a package from another person is the value-added features, support channels, and documentation. The money isn't necessarily in the product alone; it can also be in the services that go with it.

The Disadvantages of Open Source Software

This section was included to provide a detailed, balanced, and unbiased contrast to the previous section, which discussed some of the advantages of open source software.

Unfortunately, we couldn't come up with any disadvantages at the time of this writing! Nothing to see here.

Understanding the Differences Between Windows and Linux

As you might imagine, the differences between Microsoft Windows and the Linux operating system cannot be completely discussed in the confines of this section. Throughout this book, topic by topic, you'll read about the specific contrasts between the two systems. In some chapters, you'll find no comparisons because a major difference doesn't really exist.

But before we attack the details, let's take a moment to discuss the primary architectural differences between the two operating systems.

Single Users vs. Multiple Users vs. Network Users

Windows was originally designed according to the "one computer, one desk, one user" vision of Microsoft co-founder Bill Gates. For the sake of discussion, we'll call this philosophy "single user." In this arrangement, two people cannot work in parallel running (for example) Microsoft Word on the same machine at the same time. You can buy Windows and run what is known as Terminal Services or thin clients, but this requires extra computing power/hardware and extra costs in licensing. Of course, with Linux, you don't run into the cost problem, and Linux will run fairly well on averagely specced hardware. Linux easily supports multiuser environments, where multiple users doing different things can be concurrently logged onto a central machine. The operating system (Linux) on the central machine takes care of the resource-"sharing" details.

"But, hey! Windows can allow people to offload computationally intensive work to a single machine!" you may argue. "Just look at SQL Server!" Well, that position is only half correct. Both Linux and Windows are indeed capable of providing services such as databases over the network. We can call users of this arrangement *network users,* since they are never actually logged into the server but rather send requests to the server. The server does the work and then sends the results back to the user via the network. The catch in this case is that an application must be specifically written to perform such server/client duties. Under Linux, a user can run any program allowed by the system administrator on the server

without having to redesign that program. Most users find the ability to run arbitrary programs on other machines to be of significant benefit.

The Monolithic Kernel and the Micro-Kernel

Three popular forms of kernels are used in operating systems. The first, a *monolithic kernel,* provides all the services the user applications need. The second, a *micro-kernel,* is much more minimal in scope and provides only the bare minimum core set of services needed to implement the operating system. And the third is a hybrid of the first two.

Linux, for the most part, adopts the monolithic kernel architecture: It handles everything dealing with the hardware and system calls. Windows, on the other hand, has traditionally worked off a micro-kernel design, with the latest Windows server versions using the hybrid kernel approach. The Windows kernel provides a small set of services and then interfaces with other executive services that provide process management, input/output (I/O) management, and so on. It has yet to be proved which methodology is truly the best way.

Separation of the GUI and the Kernel

Taking a cue from the original Macintosh design concept, Windows developers integrated the GUI with the core operating system. One simply does not exist without the other. The benefit with this tight coupling of the operating system and user interface is consistency in the appearance of the system.

Although Microsoft does not impose rules as strict as Apple's with respect to the appearance of applications, most developers tend to stick with a basic look and feel among applications. One reason why this is dangerous, however, is that the video card driver is now allowed to run at what is known as "Ring 0" on a typical *x*86 architecture. Ring 0 is a protection mechanism—only privileged processes can run at this level, and typically user processes run at Ring 3. Because the video card is allowed to run at Ring 0, it could misbehave (and it does!), and this can bring down the whole system.

On the other hand, Linux (like UNIX in general) has kept the two elements—user interface and operating system—separate. The windowing or graphical stack (X11, Xorg, Wayland, and so on) is run as a user-level application, which makes the overall system more stable. If the GUI (which is complex for both Windows and Linux) fails, Linux's core does not go down with it. The GUI process simply crashes, and you get a terminal window. The graphical stack also differs from the Windows GUI in that it isn't a complete user interface. It defines only how basic objects should be drawn and manipulated on the screen.

One of the most significant features of the X Window System is its ability to display windows across a network and onto another workstation's screen. This allows a user sitting on host A to log into host B, run an application on host B, and have all of the output routed back to host A. It is possible, for example, for several users to be logged into the same machine and simultaneously use an open source equivalent of Microsoft Word (such as LibreOffice).

In addition to the core graphical stack, a window manager is needed to create a useful environment. Linux distributions come with several window managers, including the heavyweight and popular GNOME and KDE environments. Both GNOME and KDE offer

an environment that is friendly, even to the casual Windows user. If you're concerned with speed and small footprint, you can look into the F Virtual Window Manager (FVWM), Lightweight X11 Desktop Environment (LXDE), and XFCE window managers.

So which approach is better—Windows or Linux—and why? That depends on what you are trying to do. The integrated environment provided by Windows is convenient and less complex than Linux, but out of the box, Windows lacks the X Window System feature that allows applications to display their windows across the network on another workstation. The Windows GUI is consistent, but it cannot be easily turned off, whereas the X Window System doesn't have to be running (and consuming valuable hardware resources) on a server.

NOTE With its latest server family of operating systems, Microsoft has somewhat decoupled the GUI from the base operating system (OS). You can now install and run the server in a so-called Server Core mode. Managing the server in this mode is done via the command line or remotely from a regular system, with full GUI capabilities.

My Network Places

The native mechanism for Windows users to share disks on servers or with each other is through My Network Places (the former Network Neighborhood). In a typical scenario, users *attach* to a share and have the system assign it a drive letter. As a result, the separation between client and server is clear. The only problem with this method of sharing data is more people-oriented than technology-oriented: People have to know which servers contain which data.

With Windows, a new feature borrowed from UNIX has also appeared: *mounting.* In Windows terminology, it is called *reparse points.* This is the ability to mount an optical drive into a directory on your C drive.

Right from its inception, Linux was built with support for the concept of mounting, and as a result, different types of file systems can be mounted using different protocols and methods. For example, the popular Network File System (NFS) protocol can be used to mount remote shares/folders and make them appear local. In fact, the Linux Automounter can dynamically mount and unmount different file systems on an as-needed basis. The concept of mounting resources (optical media, network shares, and so on) in Linux/UNIX might seem a little strange, but as you get used to Linux, you'll understand and appreciate the beauty in this design. To get anything close to this functionality in Windows, you have to map a network share to a drive letter.

A common example of mounting resources under Linux involves mounted home directories. The user's home directories can reside on a remote server, and the client systems can automatically mount the directories at boot time. So the **/home** (pronounced *slash home*) directory exists on the client, but the **/home/username** directory (and its contents) can reside on the remote server.

Under Linux, NFS, and other Network File Systems, users never have to know server names or directory paths, and their ignorance is your bliss. No more questions about which server to connect to. Even better, users need not know when the server configuration must change. Under Linux, you can change the names of servers and adjust this information on client-side systems without making any announcements or having to reeducate users.

Anyone who has ever had to reorient users to new server arrangements or major infrastructure changes will appreciate the benefits and convenience of this.

The Registry vs. Text Files

Think of the Windows Registry as the ultimate configuration database—thousands upon thousands of entries, only a few of which are completely documented.

"What? Did you say your Registry *got corrupted?*" <insert maniacal laughter> "Well, yes, we can try to restore it from last night's backups, but then Excel starts acting funny and the technician (who charges $130 just to answer the phone) said to reinstall …."

In other words, the Windows Registry system can be, at best, difficult to manage. Although it's a good idea in theory, most people who have serious dealings with it don't emerge from battling it without a scar or two.

Linux does not have a registry, and this is both a blessing and a curse. The blessing is that configuration files are most often kept as a series of text files (think of the Windows .ini files). This setup means you're able to edit configuration files using the text editor of your choice rather than tools such as **`regedit`**. In many cases, it also means you can liberally add comments to those configuration files so that six months from now you won't forget why you set up something in a particular way. Most software programs that are used on Linux platforms store their configuration files under the **/etc** (pronounced *slash etc*) directory or one of its subdirectories. This convention is widely understood and accepted in the FOSS world.

The curse of a no-registry arrangement is that there is no standard way of writing configuration files. Each application can have its own format. Many applications are now coming bundled with GUI-based configuration tools to alleviate some of these problems. So you can do a basic setup easily via the GUI tool and then manually edit the configuration file when you need to do more complex adjustments.

In reality, having text files hold configuration information usually turns out to be an efficient method and makes automation much easier too. Once set, these files rarely need to be changed; even so, they are straight text files and therefore easy to view and edit when needed. Even more helpful is that it's easy to write scripts to read the same configuration files and modify their behavior accordingly. This is especially helpful when automating server maintenance operations, which is crucial in a large site with many servers.

Domains and Active Directory

The idea behind Microsoft's Active Directory (AD) is simple: Provide a repository for any kind of administrative data, whether it is user logins, group information, or even just telephone numbers. In addition, AD provides a central place to manage authentication and authorization (using Kerberos and LDAP) for a domain. The domain synchronization model also follows a reliable and well-understood Domain Name System (DNS)–style hierarchy. As tedious as it may be, AD works pretty well when properly set up and maintained.

Out of the box, Linux does not use a tightly coupled authentication/authorization and data store model the way that Windows does with AD. Instead, Linux uses an abstraction model that allows for multiple types of stores and authentication schemes to work without any modification to other applications. This is accomplished through the Pluggable Authentication

Modules (PAM) infrastructure and the name resolution libraries that provide a standard means of looking up user and group information for applications. PAM also provides a flexible way of storing that user and group information using a variety of schemes.

For administrators looking to Linux, this abstraction layer can seem peculiar at first. However, consider that you can use anything from flat files to Network Information Service (NIS), Lightweight Directory Access Protocol (LDAP), or Kerberos for authentication. This means you can pick the system that works best for you. For example, if you have an existing infrastructure built around AD, your Linux systems can use PAM with Samba or LDAP to authenticate against the Windows domain model. And, of course, you can choose to make your Linux system not interact with any external authentication system. In addition to being able to tie into multiple authentication systems, Linux can easily use a variety of tools, such as OpenLDAP, to keep directory information centrally available as well.

Summary

This chapter offered an overview of what Linux is and what it isn't. We discussed a few of the guiding principles, ideas, and concepts that govern open source software and Linux by extension. We ended the chapter by covering some of the similarities and differences between core technologies in the Linux and Microsoft Windows Server worlds. Most of these technologies and their practical uses are dealt with in greater detail in the rest of this book.

If you are so inclined and would like to get more detailed information on the internal workings of Linux itself, you might want to start with the source code. The source code can be found at www.kernel.org. It is, after all, open source!

This page intentionally left blank

CHAPTER 2

Installing a Linux Server

From massive mainframe systems to simple Internet of Things (IoT) devices, smart TVs, phones, and Tera-Core servers—if you can imagine it, it's probably possible to get a Linux distribution on any given target platform or hardware of your choice. There are many ways to install the software too: via a USB/flash drive, optical media, network, and so on. Network installation options can be a wonderful help when you're installing a large number of hosts. Another popular method of installing a Linux distribution is from what is known as "Live Media," which simply allows you to try the software before committing to installing it.

For the purposes of this Linux server installation chapter, keep in mind that there is nothing particularly special about server class installations, except for the salient fact that servers very often perform very specialized functions and are usually dedicated to performing only a suite of specific tasks. As a result, you should avoid installing extraneous software and running irrelevant services on servers because it creates server bloat and can be a drag on system performance.

In this chapter, we discuss the installation process as it pertains to servers and their dedicated functions. We start off with some general server housekeeping considerations.

Hardware and Environmental Considerations

As you would with any operating system, before you get started with the installation process, you should determine what hardware configurations will work. Most mainstream Linux distros have a hardware compatibility list (HCL) of some sort available on their web site. For example, openSUSE's HCL database can be found at http://en.opensuse.org/Hardware, Ubuntu's HCL can be found at https://wiki.ubuntu.com/HardwareSupport, and a more generic (and older) HCL for most Linux flavors can be found at www.tldp.org/HOWTO/Hardware-HOWTO.

These sites provide a good starting reference point when you are in doubt concerning a particular piece of hardware. However, keep in mind that new Linux device drivers are being churned out on a daily basis around the world, and no single site can keep up with the pace

of development in the open source community. In general, most popular Intel-based, AMD-based, and ARM-based configurations work without difficulty. The same goes for the emerging IoT and open hardware sectors.

A general rule that applies to all operating systems is to avoid bleeding-edge hardware and software configurations in production environments. Although their specs might appear impressive, they may not yet have been through the maturing process that some of the slightly older hardware has undergone. For servers, this usually isn't too much of an issue, since there is no need for a server to have the latest and greatest toys, such as fancy video cards and sound cards. Your main goal, after all, is to provide a stable and highly available server for your users.

Server Design

By definition, well-oiled server-grade systems exhibit three important characteristics: stability, availability, and performance. These three factors are usually improved through the purchase of more and better hardware, which is unfortunate. You needn't always have to shell out thousands of dollars extra to get a system capable of excelling in all three areas—especially when you could have extracted the desired level of performance out of existing hardware with a little tuning. With Linux, this is not difficult; even better, the gains are outstanding!

One of the most significant design decisions you must make when managing a server may not even be technical, but administrative. You must design servers to be *unfriendly* to casual users. This means no cute multimedia tools, no sound card support, and no fancy web browsers (when at all possible). In fact, casual use of a server should be strictly prohibited as a rule.

NOTE Due to the specialized workloads of servers, you will find that servers often have slightly different configurations and design variables. For example, a server dedicated to handling logging information sent to it over the network is going to have especially large partitions set up for the appropriate logging directories compared to a file server that performs no logging of its own.

Another important consideration in planning a server deployment is making sure that it resides in the most appropriate environment. As a system administrator, you must ensure the physical safety of your servers by keeping them in a separate room under lock and key (or the equivalent). The only access to the servers for non-administrative personnel should be through the network. The server room itself should be well ventilated and kept cool. The wrong environment is an accident waiting to happen.

Once the system is in a safe place, planning for emergency power (battery backup) is also crucial—to keep systems running during power failures or other voltage anomalies.

Here are some specific things you can do to improve your server performance:

- Take advantage of the fact that the graphical user interface (GUI) is uncoupled from the core operating system, and avoid starting the graphical subsystem unless someone needs to sit at the console to run applications. After all, like any other application, the graphical stack uses memory and CPU time, both of which are better off going to the more essential server processes instead.

- Determine what functions the server is to perform, and disable all other unrelated functions. Not only are unused functions a waste of memory and CPU time, but they are just another issue you need to deal with on the security front.
- Unlike some other operating systems, Linux allows you to pick and choose the features you want in the kernel. (You'll learn about this in Chapter 10.) The default kernel will already be reasonably well tuned, so you won't have to worry about it. But if you do need to change a feature or upgrade the kernel, be picky about what you add. Make sure you really need a feature before adding it.

Uptime

All of this chatter about taking care of servers and making sure silly things don't cause them to crash stems from a longtime UNIX philosophy: *Uptime is good. More uptime is better.*

The Linux/UNIX **uptime** command tells the user how long the system has been running since its last boot, how many users are currently logged in, and how much load the system is experiencing. The last two metrics are useful measures that are necessary for day-to-day system health and long-term planning. For example, if the server load has been staying abnormally and consistently high, it might mean that it's time to invest in a faster/bigger/better server.

But the all-important number is how long the server has been running since its last reboot. Long uptime is regarded as a sign of proper care, maintenance, and, from a practical standpoint, system stability. You'll often find Linux administrators boasting about their server's uptime the way you hear car buffs boast about horsepower. This is also why you'll hear Linux administrators cursing at system changes (regardless of operating system) that require a reboot to take effect!

Methods of Installation

With the improved connectivity and speed of both local area networks (LANs) and Internet connections, it is becoming an increasingly popular option to perform installations over the network rather than using local media (optical CD/DVD-ROM, flash drive, and so on).

Depending on the particular Linux distribution and the network infrastructure already in place, you can design network-based installations around several protocols, including the following popular ones:

- **FTP (File Transfer Protocol)** This is one of the earliest methods for performing network installations.
- **HTTP (Hypertext Transfer Protocol)** The installation tree is served from a web server.
- **NFS (Network File System)** The distribution tree is shared/exported on an NFS server.

- **SMB (Server Message Block)** In supported distros, the installation tree can be shared on a Samba server or even shared from a Windows box.
- **PXE, DHCP, TFTP, and IPMI** These protocols and technologies can be chained together in various ways to seamlessly bootstrap OS installations and provision new servers. The company (Canonical) behind the Ubuntu distro's Metal as a Service (MaaS) project is a good example of a comprehensive solution that leverages these technologies.

The other, more typical method of installation is through the use of a media image provided by the vendor. This installer image can come in formats such as ISO9660, UDF, IMG, Compressed Raw Disk images, and so on.

Alongside downloadable installer images, some commercial Linux distributions also have options of physical/boxed installation media that can be ordered from them.

Another variant of installing Linux is installing via what is referred to as a "live operating system or live distro" environment. This environment can be a live USB or even a live CD/DVD. The word *Live* here, simply means that the same install medium can be used to boot and try out (test drive) the distribution first *without* actually installing anything onto the drive. It allows the user to have a rough idea of how hardware and other peripherals on the target system will behave. Keep in mind that even when things don't work perfectly during your tests in a live distro environment, you can usually get troublesome hardware working after the fact with a little tweak here and there—though your mileage will vary.

We will be performing a server class install in this chapter using an operating system image that was downloaded from the distribution web site.

Installing Fedora

In this section, we will install a 64-bit rawhide version of the Fedora distribution on a standalone system. Later chapters build on the groundwork done in this chapter by exploring as well as adding and configuring new subsystems.

NOTE The term *rawhide* refers to the very latest development version of Fedora. This version almost always points to the next release of Fedora. The Fedora Linux distribution is known to be the breeding ground for bleeding-edge, fast-changing, and experimental FOSS technologies—and for this reason it is highly discouraged to use standard/regular Fedora in production environments. The rawhide stream of Fedora is even *much more* experimental and unstable than regular Fedora! We thought long and hard before settling on the rawhide stream of Fedora to use as a our sample server, and we feel it is worth the risk to showcase the very latest FOSS technologies to our readers. Later on, we install the more conservative and stable Ubuntu LTS distro as the other demo distro. But don't worry if you choose to install a different version or architecture of Fedora. The installation steps involved between versions are similar. You'll also be fine if you choose to install a Linux distro other than Fedora; luckily, most of the concepts carry over among the various distributions. Some installers are just prettier than others!

Project Prerequisites

First, you need to download the ISO for Fedora that we will be installing. Fedora's project web page has a listing of several mirrors located all over the world. Whenever possible, you should choose the mirror geographically closest to you. The list of official mirrors can be found at http://mirrors.fedoraproject.org.

The latest ISO image for the rawhide version of Fedora that we will be using for this installation can be downloaded from the following top-level directory:

http://download.fedoraproject.org/pub/fedora/linux/development/rawhide/Server/x86_64/iso/

You can alternatively download the image from this mirror:

http://mirrors.kernel.org/fedora/development/rawhide/Server/x86_64/iso/

Once you are at the top level of the web server folder, you will be presented with various downloadable files representing other variants of Fedora rawhide, the upload date, and the approximate file size. For example, you will see files with names like:

Fedora-Server-dvd-x86_64-Rawhide-<DATE-STAMP>.n.1.iso

Fedora-Server-netinst-x86_64-Rawhide-<DATE-STAMP>.n.0.iso>

The files often have the latest upload date stamp (DATE-STAMP) as part of the filename in the form YYYYMMDD, where YYYY is the year, MM is the month, and DD is the day.

NOTE Linux distributions are often packaged by the architecture on which they were compiled to run. You would often find ISO images (and other software) named to reflect an architecture type. Examples of the architecture types are x86/i386, x86_64, and aarch64, where the x86_64 family refers to the 64-bit platforms (such as Phenom, EM64T, Ryzen, and Intel Core i5, i7, and i9).

The next step is to write the ISO image file to a suitable medium.

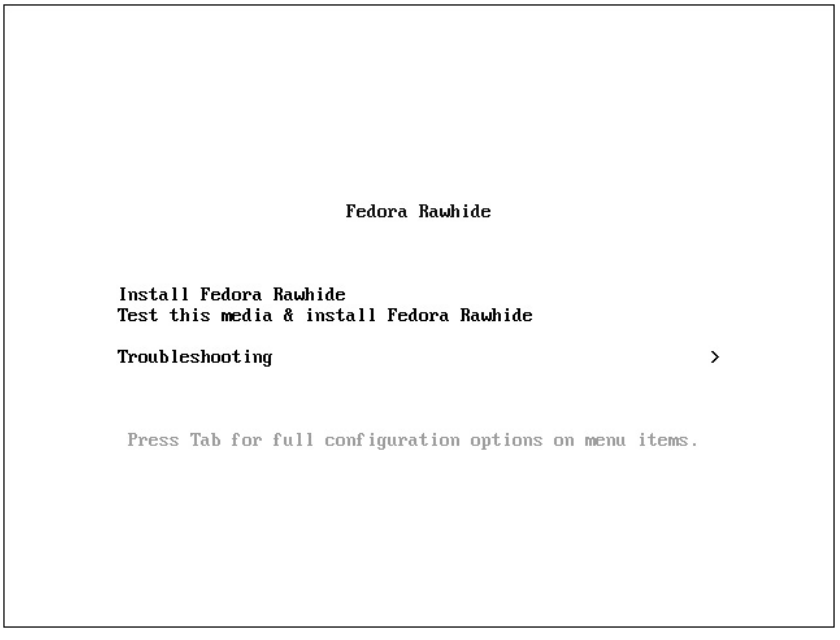
If you plan on performing the installation using external flash-based media, such as a USB stick, a Secure Digital (SD) card, or an external hard disk, the system needs to be able to boot off the medium. Appendix A discusses how to create a Linux installer on flash-based media.

NOTE On the off chance that you have a physical CD/DVD drive on your chosen hardware and blank optical CD/DVD-ROM media, you might want to perform the Fedora Installation using the same, by burning the installer image to the media using your favorite CD/DVD-burning program. Just remember that the file you downloaded is already an exact image of the installer and so should be treated as such. Most CD/DVD-burning programs have an option to create a CD or DVD from an ISO image. Note that if you burn the file you downloaded as you would a regular data file, you will end up with a single file on the root of your optical media. This is *not* what you want.

The Installation

Let's begin the installation process.

Insert and boot off the installation medium (optical disk, USB flash drive, and so on). The system's Unified Extensible Firmware Interface (UEFI) or Basic Input/Output System (BIOS) should be preconfigured to boot from the correct medium. This will present a welcome splash screen, as shown next.



If you do not press any key, the prompt will begin a countdown, after which the installation process will start by booting the highlighted Test This Media & Install Fedora ... option. You can also press ENTER to start the process immediately.

A quick media verification step will take place. This media verification step can save you the trouble of starting the installation only to find out halfway through that the installer will abort because of bad installation media.

After the media check runs to completion, and the media is successfully verified to be usable, the installer will automatically continue to the next screen.

Select the language you want to use to perform the installation in this screen. For this example, we select English (United States). Then click the Continue button.

If you are installing the rawhide version of Fedora, you might see a screen pop up warning you that you are about to install prerelease/non-production software. Click "I want to proceed" to continue.

Installation Summary

The Installation Summary screen is an all-in-one area where you make the important decisions about the operating system to be installed. The screen is roughly divided into the following sections:

- Localization (Keyboard, Language Support, and Time & Date)
- Software (Installation Source and Software Selection)
- System (Installation Destination and Network & Hostname)

We delve into each of these sections next and make changes where necessary.

Localization Section

As the name implies, this section is used for customizing items related to the locale of the system. This includes Keyboard, Language Support, and Time and Date.

Keyboard

From the Installation Summary screen, click the Keyboard option to specify the keyboard layout of the system. You can add additional keyboard layouts if you need to in the ensuing screen and specify their order.

For this example, we accept the default value (English US) and make no change. Click Done when you're satisfied with any changes you may have made here.

Language Support

Clicking the Language Support option on the Installation Summary screen enables you to specify support for additional languages that you may need on the finished system.

For this example, we accept the default value (English – United States) and make no change. Click Done when you're satisfied with any changes you may have made here.

Time & Date

Click the Time & Date option on the main Installation Summary screen to bring up another screen that will allow you to select the time zone in which the machine is located. Scroll through the list of regions and cities and select the area closest to you. You can also use the interactive map to select a specific city (marked by a colored dot) to set your time zone.

Depending on your installation source, the Network Time option could be set to ON or OFF by default. Accept the default ON setting; this allows the system to automatically set the correct time using the Network Time Protocol (NTP). Click Done when you're satisfied with your changes.

Software Section

Under the Software section of the Installation Summary screen, you can select the installation source as well as additional packages (applications) that get installed.

Installation Source

Since we are performing our installation using a full Fedora image, you will notice that Local Media is automatically specified under the Installation Source section of the main Installation Summary screen.

Clicking on Installation Source will reveal a screen that allows you to specify custom installation sources and additional repositories from which to retrieve installation packages via different network protocols, such as HTTP, HTTPS, FTP, and NFS. This is also where you can specify that the installer download and automatically install the latest packages available upstream—during the installation.

NOTE If you are performing an installation using a minimal Fedora network image that is designed to automatically download the required software from a network source, you will notice that the Closest Mirror option is selected by default for you as the installation source. The Closest Mirror option will automatically determine which country or locale your target system is connected from and use your public IP address to pick a network mirror that is geographically closest to you. This helps to speed up the installation procedure by downloading the needed packages from servers that are closest to you. A sample minimal Fedora network image can be downloaded from here: https://dl.fedoraproject.org/pub/fedora/linux/development/rawhide/Server/x86_64/iso/

Software Selection

Clicking the Software Selection option on the main Installation Summary screen presents you with the section of the installation where you can pick the exact software packages that get installed on the system.

The Fedora installer determines which packages are available based on the Fedora *spin* (or edition) that you start off with. Some sample Fedora spins are Workstation, Server, Cloud, KDE, XFCE, and Container. Each spin in turn has several predefined add-ons that comprise software package groups that provide a specific functionality. Sample functionality provided by package groups or add-ons are Basic Web Server, Container Management, Development Tools, Mail Server, Domain Membership, and Editors. Selecting any of the add-ons in the right pane will automatically select/install the actual individual software packages needed to provide the functionality. This organization allows you to select the functionality that you want installed and safely ignore the details.

Here we are using the Fedora Server Edition and thus we have certain predefined add-ons or package groups available to select from, if needed.

Accept the default setting and click Done at the top of the screen.

CAUTION Ideally, on a production system that is supposed to be used as a server for a specific function, you should seriously consider *not* installing any extraneous functionalities or add-ons beyond what is needed to provide the server's core functions. You should always aim to keep your server as lean as possible. Less is more!

System Section

The System section of the Installation Summary screen is used for customizing and making changes to the underlying hardware of the target system. This is where you create your hard drive partitions or volumes, specify the file system to be used, and specify the network configuration.

Installation Destination

From the Installation Summary screen, click the Installation Destination option. This takes you to the corresponding task area. This portion of the installation is probably the part that most new Linux users find the most awkward because of the different hardware naming conventions Linux uses. This needn't be a problem, however—all it takes is a slight shift in thinking. To help with this, keep in mind that *a disk is a disk is a disk* in Linux, Windows, BSD, and macOS.

You will see a screen displaying all the candidate disk drives that you have available on the target system. If you have only one disk drive on the system, as on our sample system, you will see the drive listed under Local Standard Disks with a check mark beside it. Clicking the disk icon will toggle on or off the disk selection check mark. We want it selected/checked here.

The other configurable option in this area is to configure other specialized and network disks such as SAN devices, iSCSI, FCoE, and so on. We are installing on a traditional disk storage device in our example, so we will not be adding any specialized or network disks.

Under the Storage Configuration Options section, select the Custom radio button. Then click Done at the top of the screen.

Once the installer determines that you have a usable disk, you will be presented with the Manual Partitioning screen, shown in Figure 2-1.

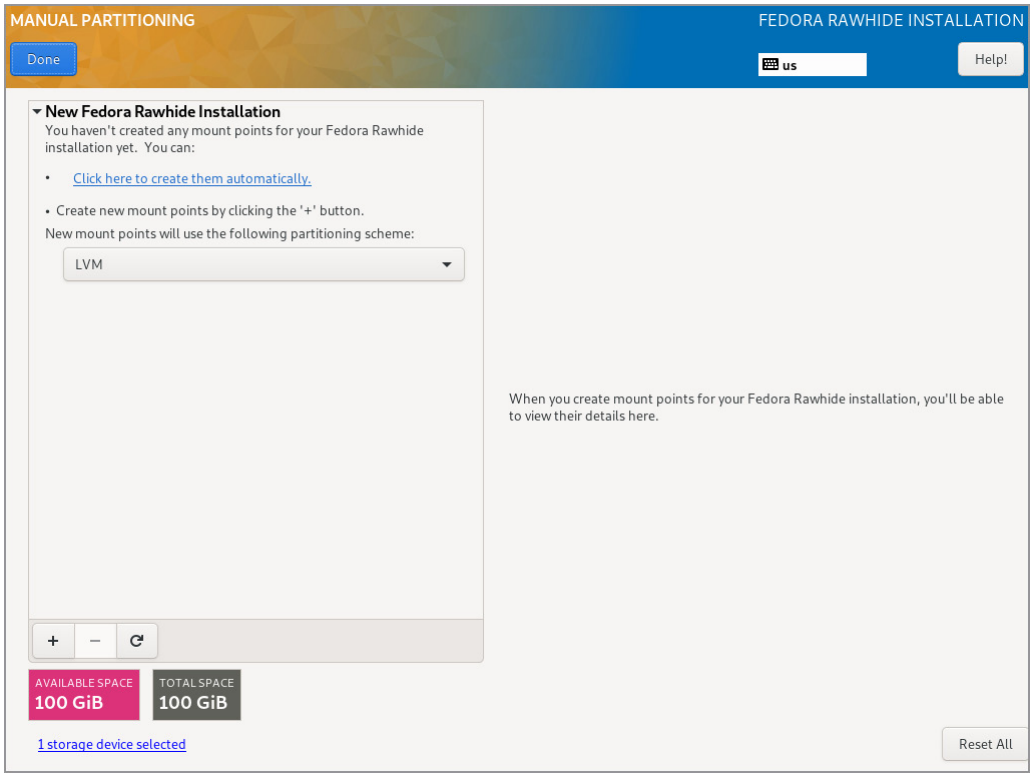


Figure 2-1 Manual Partitioning screen

NOTE If you are performing the installation on a disk with existing data, you can select the "I would like to make additional space available" option under the Storage Configuration area. This will allow the installer to present you with other options to resize or remove existing volumes or file systems on the disk in order to create space for the new OS.

On the Manual Partitioning screen, make sure that LVM is selected as the partitioning scheme for new mount points. LVM is introduced in the following section.

Before we delve into the partitioning setup proper, we will provide a quick overview of the common partitioning scheme and file system layouts.

Manual Storage Configuration Overview Note that even though we selected the custom partitioning option, we are still going to let the installer automatically slice up and lay out the disk for us. We choose this path just so that we can introduce some basic Linux server partitioning concepts. Doing this also gives us a chance to see the structure being automatically created for us—before it is finalized.

The installer provides the option to lay out the disk partition automatically; we will accept the default suggested layout.

Partitioning Schemes and File System Layouts

Let's digress a bit from the actual OS installation and do a quick overview of common partitioning schemes and file system layouts on Linux servers, along with the equivalent partitions in the Windows world. We will eventually implement a slightly modified version of this scheme for our server.

- **/** The root (pronounced *slash* or *root*) partition is identified by a forward slash (/). All other directories are attached (mounted) to this parent directory. It is equivalent to the system drive (**C:**) in Windows.
- **/boot** This partition or volume contains almost everything required for the initial boot process. The equivalent of this in Windows is the system partition (*not* the boot partition).
- **/boot/efi** This is the EFI system partition (ESP). This partition (or its equivalent) is mandated by modern systems that implement a Unified Extensible Firmware Interface (UEFI). UEFI is an alternative/replacement for the older Basic Input/Output System (BIOS). UEFI is the very first program that runs when the computer is powered on.

  The **/boot/efi** volume is not Linux specific and is thus similarly required on Windows and macOS systems with UEFI. The partition can store boot loaders, kernel images, device drivers, and other utilities.

(*continued*)

- **/usr** This is where all program files (binaries) commonly reside (similar to **C:\Program Files** in Windows).
- **/home** This is where users' home directories and personal data are stored. Having user data on a separate partition helps to prevent users from inadvertently consuming the entire disk and starving other critical system components that also need storage. This directory is synonymous with **C:\Users** in Windows operating systems.
- **/var** This is where [variable] operating system and application-related data generated during normal system operation is stored. This includes files such as system/event logs, lock files, caches, and mail. Because log files tend to grow in size quickly, it is important to store the logs on a separate partition so that their size or growth is constrained within a defined area. This can help to avoid a type of self-inflicted denial of service attack. Logs are generally stored in the **C:\WINDOWS\system32\config** directory in Windows.
- **/tmp** (Pronounced *slash temp.*) This is where temporary files are placed. Anything stored under the **/tmp** folder is usually not preserved between system reboots. **/tmp** is sometimes implemented as a special virtual memory file system called **tmpfs**.
- **Swap** Swap space is an area on the disk that can be used to temporarily offload/extend some of the work done in physical memory (RAM) on a system. It is a type of virtual file system. This isn't a user-accessible file system. Linux can also use a normal disk file to hold swap space, similar to the way Windows does. Swapping functionality is provided via a paging file in the Windows world.

  If the system RAM is less than 4GB, you will typically want to configure your swap file to be double the size of the physical memory that is in your system.

Once defined and created, each of these partitions is mounted at boot time. The mount process makes the contents of that partition available as if it were just another directory on the system. For example, the root directory (/) could be on the first partition. A subdirectory called **/usr** will exist on the root directory but will have nothing in it. A separate partition can then be mounted at **/usr**, such that going into the **/usr** directory will allow you to see the contents of the mounted partition or underlying block device. Think of the mount point as a sort of *anchor point*.

Once mounted, all the file systems/partitions/volumes appear as a unified directory tree rather than as separate drives; the installation software does not differentiate one partition from another. All it cares about is which directory each file goes into. As a result, the installation process automatically and correctly distributes its files across all the mounted file systems.

The disk partitioning tool used during the operating system installation provides an easy way to create partitions and associate them to the directories on which they will be mounted. Each mount point entry (or partition entry) will typically show the following information:

- **Device** Linux associates each partition with a separate device (file). For the purpose of this installation, you need to know only that entire individual disks are referred to with names that *loosely* follow the pattern **/dev/XYZ**, where:
  - **X** refers to the device type (for example, IDE/SATA/SAS/SCSI/NVMe).
  - **Y** is a letter representing an instance of a unique physical drive of a given type (for example, a, b, c, and so on).
  - **Z** represents a partition number on a given physical drive (for example, 1, 2, 3, and so on).

  Let's apply this pattern to some real-world examples:
  - The first 20TiB SATA SSD disk on a Linux server might be named and addressed as **/dev/sda**.
  - The first partition/slice on the second disk (**/dev/sdb**) will be named and addressed as **/dev/sdb1**.
  - The second partition/slice on the second SATA disk will be named and addressed as **/dev/sdb2**.
  - The first M.2 NVMe PCI-Express Internal SSD detected might be named and addressed as **/dev/nvme0n1**.
  - The fourth partition or slice on the M.2 SSD drive will be named **/dev/nvme0n1p4**.
  - The first virtual hard disk detected or attached to a virtual machine might be named and addressed as **/dev/vda**.
- **Mount Point** This field shows the directory location where the partition is mounted.
- **Desired Capacity** This field shows the partition's size in mebibyte (MiB) or gibibyte (GiB).
- **Device Type** This field shows the partition type (for example, Standard Partition, BTRFS, LVM, or LVM Thin Provisioning).
- **File System** This field shows the partition's file system type (for example, ext2, ext3, ext4, swap, BIOS Boot, xfs, Btrfs, or vfat).
- **Reformat** This field indicates whether the partition will be formatted.
- **Volume Group** This field shows the volume group that the partition belongs to. You can either accept the default value or create a new volume group name.

For the sake of simplicity in this chapter, we will accept the default partitioning scheme recommended by the installer. But in a later chapter (Chapter 8), we will implement some

server partitioning and file system layout best practices. This will be done by resizing and/or creating new volumes from scratch using whatever free space we have available on the disk.

The sample UEFI-based system on which this installation is being performed has a 100GB hard disk.

The automatic partitioning tool will by default create the **/boot** partition on a standard partition type. On UEFI-based systems, it will additionally automatically suggest and create a **/boot/efi** partition. It will also automatically create the / (root) and swap containers on an LVM-type partition. For more on LVM, see Chapter 8.

The following table shows a summary of the resulting partition/volume sizes and file systems from the automatic partition layout process. You can use this as a guide if you choose to manually carve up your disk yourself. You should, of course, adjust the suggested sizes to suit the overall size of the disk you are using.

Mount Point/Partition	Size
/boot	~ 1 GiB
/boot/efi (only created on UEFI systems)	~ 200 MiB
/	~ 15 GiB
Swap	~ 8 GiB
Free space	~ 76 GiB

Now that you have some background on partitioning under Linux, let's go back to the installation process itself.

The current unconfigured Manual Partitioning screen (shown previously in Figure 2-1) presents you with different options. We'll start by making the installer suggest and automatically create our LVM partitioning scheme for us.

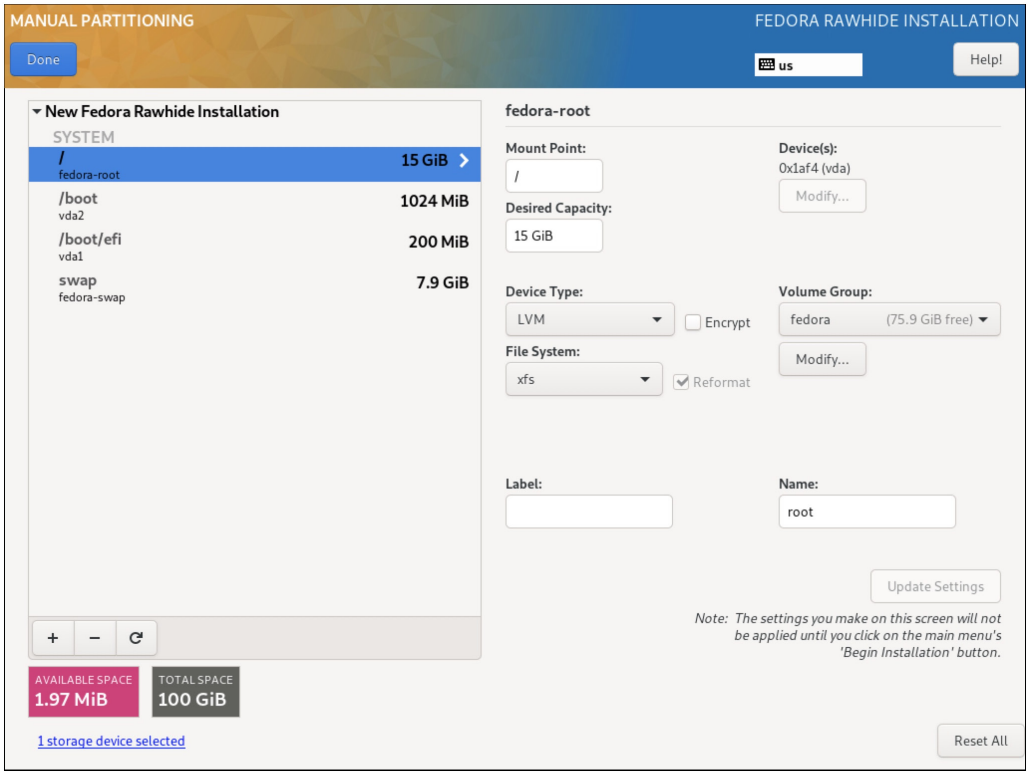
Automatically Create an LVM Disk Layout In the Manual Partitioning screen, make sure that the LVM option is selected under the "New Mount Points will use the following partitioning scheme" drop-down box.

Next, click the "Click here to create them automatically" option.

Choosing the LVM partitioning scheme and allowing the installer to automatically create the mount points, file systems, partitions, and volumes will result in an initial disk layout structure similar to the one shown in the following table.

Mount Point	Capacity	File System	Device Type
/boot	~1 GiB	ext4	Standard Partition
/boot/efi	~200 MiB	EFI System Partition	Standard Partition
/	~15 GiB	xfs	LVM
Swap	~8 GiB	swap	LVM
Free space/ unpartitioned area			LVM

The exact screen on our sample server is shown next:



NOTE The Fedora installer supports the creation of encrypted file systems; however, we will not use any encrypted file systems on our sample system.

Click Done in the upper-left corner of the Manual Partitioning screen to complete the disk-partitioning portion of the installation. You will be presented with a dialog box showing the summary of changes to be made to the disk.

Click Accept Changes to return to the Installation Summary screen and configure the final option, Network & Hostname.

Network & Hostname

The final task of the installation procedure deals with network configuration, where you can configure or tweak network-related settings for the system.

NOTE After you click on the Network & Hostname option, all correctly detected network interface hardware (such as Ethernet, wireless network cards, and so on) will be listed in the left pane of the network configuration screen.

Depending on the Linux distribution and the specific hardware setup, Ethernet devices in Linux have names similar to eth0, eth1, ens3, ens4, em1, em2, p1p1, enp0s3, and so on.

For each interface, you can either configure it using DHCP or manually set the IP address. If you choose to configure manually, be sure to have all the pertinent information ready, such as the IP address, netmask, and so on. Also, don't worry if you know that you don't have a DHCP server available on your network that will provide your new system with IP configuration information; the Ethernet interface will simply remain unconfigured. The hostname of the system can also be automatically set via DHCP, if you have a reachable and capable DHCP server on the network.

Clicking the Network & Hostname button in the main Installation Summary screen opens the corresponding configuration screen (see Figure 2-2). Among other things, you have the option to configure the hostname of the system (the name defaults to **localhost .localdomain**, as shown at the bottom of Figure 2-2). Note that you can easily change this name later on after the OS has been installed. For now, accept the default value supplied for the hostname.

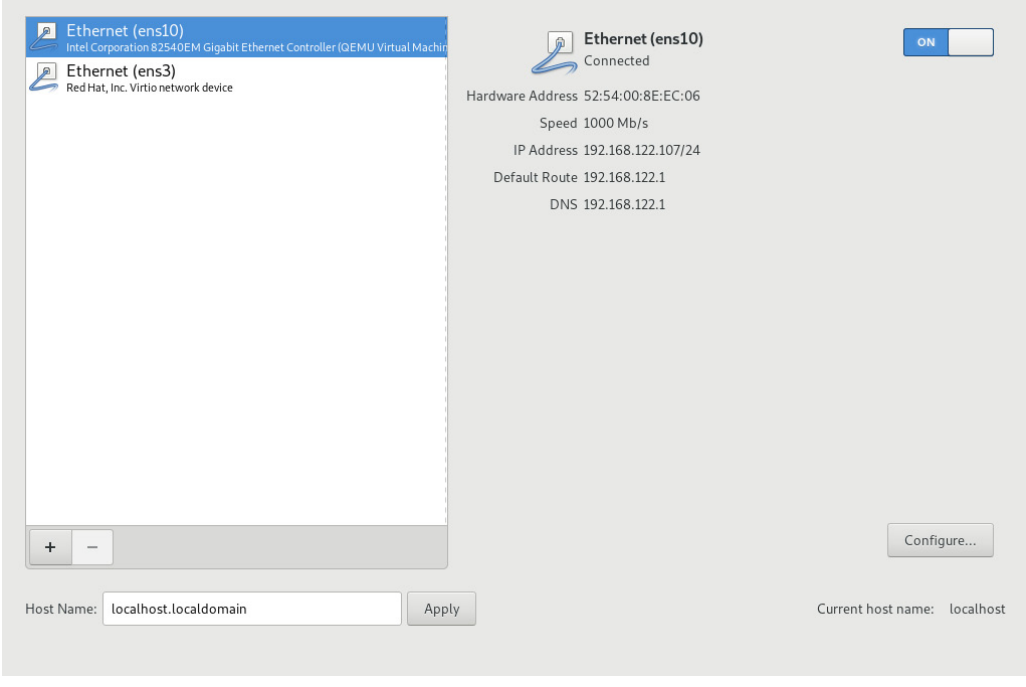


Figure 2-2 Network configuration

The next important configuration task is related to the network interfaces on the system. First, verify that an Ethernet card (or any network card) is listed in the left pane. Click any of the detected network devices in the left pane to select it. The configurable properties of the selected network adapter will appear in the right pane of the screen.

On our sample system, we have two Ethernet devices (ens3 and ens10), both of which are in a connected state. The type, name, quantity, and state of the network devices on your system may vary from the ones on our sample system.

After you select a network device, the right pane of the Network & Hostname screen will display the hardware address, speed, IP configuration information (IP address, subnet mask, default route, and DNS) of the device.

Whenever possible, the default behavior is to automatically configure the detected network interface using the Dynamic Host Configuration Protocol (DHCP).

Make sure the switch of the device you want to configure is flipped to the ON position in the right pane.

We will not make any changes on this screen, but if for whatever reason you need to manually specify IP settings for the network device, click the Configure button in the right pane and input your configuration.

Click Done to return to the main Installation Summary screen.

User Settings Section

This section can be used for creating a password for the *root* user account and also for creating new administrative or non-administrative accounts.

Set the Root Password

Click the Root Password field under User Settings to launch the Root Password task screen. In the Root Password text box, set a strong password for the root user. This user (also called the *superuser)* is the most privileged account on the system and typically has full control of the system. It is equivalent to the administrator account in Windows operating systems. Therefore, it is crucial that you protect this account with a very good password. Be sure not to choose dictionary words or names as passwords, because they are easy to guess and crack.

Enter the same password again in the Confirm text box. Click Done.

Create a User Account

Back in the main installation summary screen, click the User Creation field under User Settings to launch the Create User task screen. This task area allows you to create a privileged or non-privileged (non-administrative) user account on the system. Creating and using a non-privileged account for day-to-day tasks on a system is a good system administration practice. You'll learn how to create additional users manually in Chapter 6. But for now on our sample server, we'll create a regular user that can invoke superuser (administrator) powers when needed.

Complete the fields in the Create User screen with the following information and then click Done:

Full name	master
Username	master
Make this user administrator	Checked
Require a password to use this account	Checked
Password	72erty7!2
Confirm password	72erty7!2

The Installation

Once you are satisfied with your choices for the various installation tasks, the next phase of the installation process will begin the installation proper.

Start the Installation

Once you are satisfied with your choices for the various installation tasks, click the Begin Installation button on the main Installation Summary screen. The installation will begin, and the installer will show the progress of the installation.

NOTE If you develop cold feet after you click the Begin Installation button, you can still safely back out of the installation without any loss of data (or self-esteem). To quit the installer, simply reset your system either by clicking the Quit button, pressing CTRL-ALT-DEL on the keyboard, or pushing the reset or power switch.

When the installation begins, various tasks will begin running in the background, such as partitioning the disk, formatting the partitions or LVM volumes, checking for and resolving software dependencies, writing the operating system to the disk, and so on.

Complete the Installation

Once you've completed any of the mandatory subtasks and the installer has run its course, complete the installation by clicking the Reboot System button (see Figure 2-3). The system will reboot itself.

Log In

The system is now set up and ready for use. You will see a login screen similar to the one shown here.

```
Web console: https://localhost:9090/

localhost login:
```

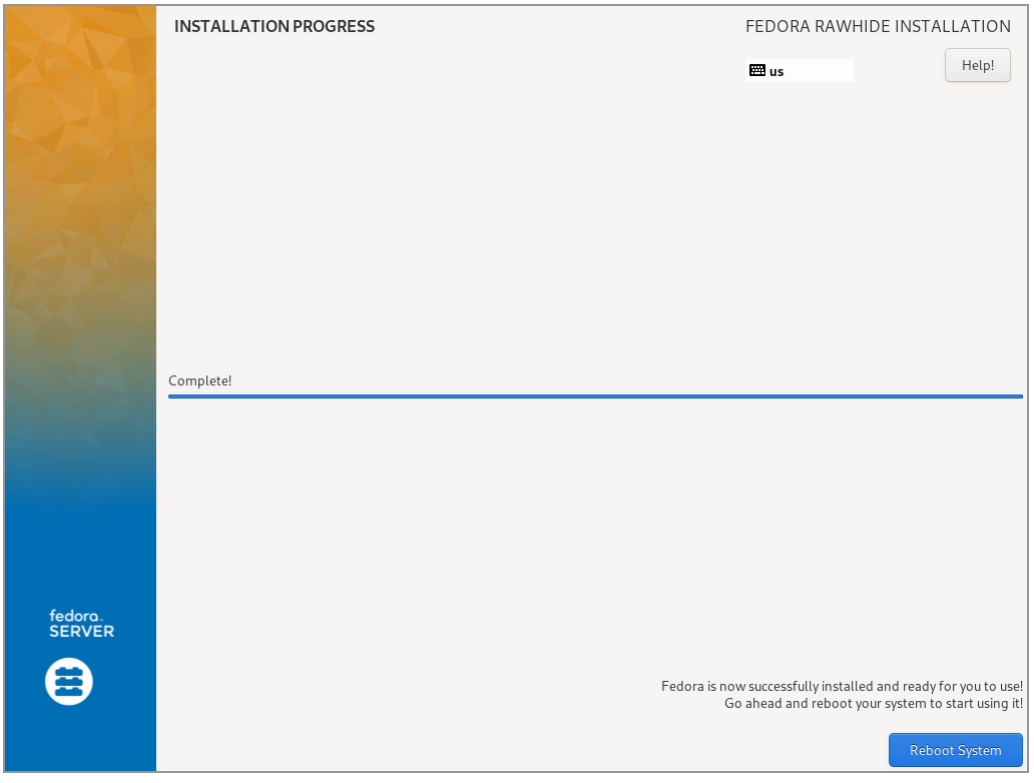


Figure 2-3 Installation completion

To log onto the system, type **master** at the login prompt and press ENTER. At the Password prompt, type **72erty7!2** (master's password) and press ENTER.

Installing Ubuntu Server

This section provides a quick overview of installing the Ubuntu Linux distribution in a server configuration. We won't be as liberal with details as we were for the Fedora server installation, because the same concepts that we discussed in detail earlier carry over here.

First, you need to download the ISO image for the latest Long Term Support (LTS) release of Ubuntu Server (in our example, this is Version 20.04.1 LTS, 64 bit). The ISO image that was used on our sample system was downloaded from https://ubuntu.com/download/server.

We will be performing the installation via flash-based media, such as a USB stick, a Secure Digital (SD) card, or an external hard disk, using the downloaded ISO image. Prior to starting the installation, you need to properly write the downloaded ISO image to the target medium. After writing the ISO image onto the appropriate medium, you should have a

bootable Ubuntu Server distribution. Appendix A discusses in detail how to create a Linux installer on flash-based media.

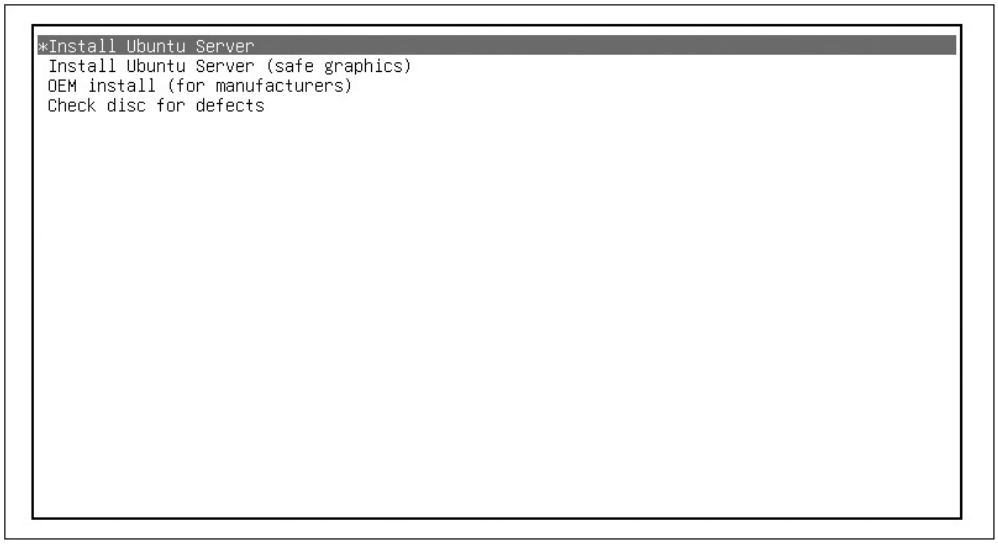
Unlike the Fedora installer or the Ubuntu Desktop installer, the Ubuntu Server installer is text-based only and is therefore not quite as pretty.

Complete the following steps to start and complete the installation.

Start the Installation

Start the installation as follows:

1. Insert the Ubuntu Server install media into the proper port on the system.
2. Make sure the system is set to use the optical drive as its first boot device in the system's UEFI or BIOS.
3. Reboot the system if it is currently powered on.
4. Once the system boots from the install media, you will see an initial boot loader screen with a few menu options. At the boot loader screen shown next, use the ARROW keys on your keyboard to select the Install Ubuntu Server option and then press ENTER.



5. Select English in the Welcome or preferred language screen and then press ENTER.
6. Next comes the keyboard configuration section of the installer. On our sample system, we accept the default automatically and correctly detected keyboard layout (English, US) and variant (English, US).

Just in case your keyboard was not correctly automatically detected, you can use the TAB or ARROW keys to navigate to the Identify Keyboard option. Doing this will allow you to manually identify your keyboard layout and variant.

Use the TAB or ARROW keys to navigate to the Done button and then press ENTER.

7. At the ensuing Welcome to Ubuntu screen, you'll be presented with a few options for the type of installation you want perform.

Make sure the Install Ubuntu option is highlighted/selected and then press ENTER.

Configure the Network

Next comes the network connections portion of the installer.

The installer has correctly detected that our sample server has two network or Ethernet cards (ens3 and ens4). The network cards have also been automatically set up to be configured via DHCP. We can very easily override this default behavior and manually specify the IP settings and other parameters for the cards by highlighting each card and pressing ENTER.

To keep things simple, we accept the default behavior here and use the TAB or ARROW keys to navigate to the Done field and then press ENTER to continue.

Configure Proxy

The configure proxy portion of the installation allows you specify the address of a proxy server that the system can use for connecting to the Internet. The address can be specified in the following format:

```
http://[user][:pass]@host[:port]/
```

Our sample server does not require a proxy server and so we leave the proxy address field blank. Select the Done button and then press ENTER.

Configure Ubuntu Archive Mirror

This part of the installer allows you to specify alternative mirrors or archives that the installer can get packages or software from. For example, you might have copies of packages on another server on the local area network that you want to point the installer to.

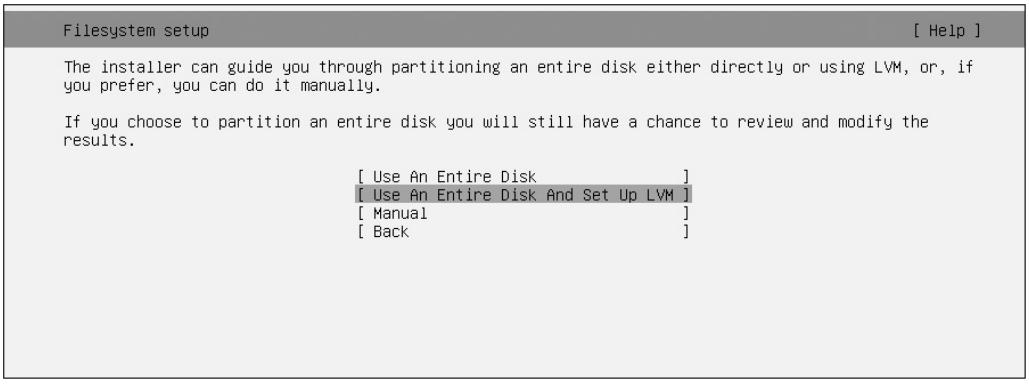
We are happy with and accept the default mirror address on our sample server.

Select the Done button and then press ENTER.

File System Setup

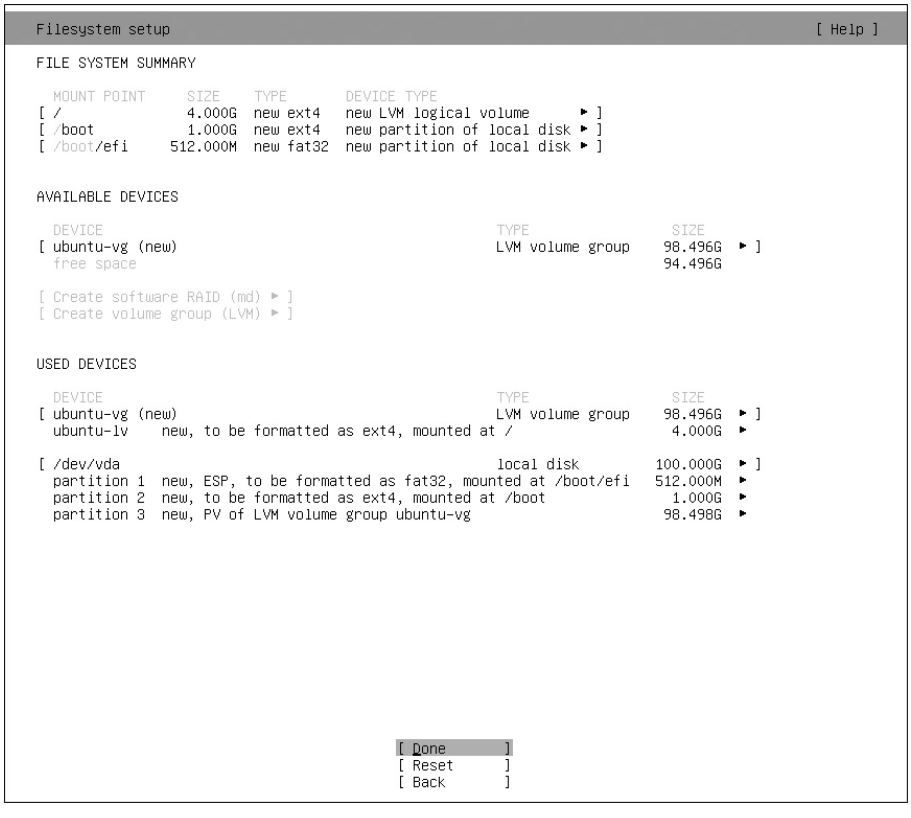
The file system setup screen of the installer that appears allows you to customize the actual partition structure of the system. It is here that you can elect to set up different file systems for different uses (such as **/var**, **/home**, and so on). The same concept used in creating the different partitions during the Fedora installation earlier transfers over for Ubuntu. For the sake of brevity, we will instead accept the default partition and LVM layout recommended by the installer. As a result, we will end up with the following file systems: **/boot**, **/boot/efi**, / (root), and **swap**. We are going to use the entire 100GB disk on our demo server and let the installer automatically set it up using LVM.

You'll have a few options in the current file system setup screen. Use the ARROW keys on your keyboard to select the Use An Entire Disk And Set Up LVM option, as shown here, and then press ENTER.



Another screen will appear, prompting you to select the disk to install to. Accept the default and correctly detected disk and press ENTER.

You'll next be shown a file system summary screen detailing how the installer will lay out the disk. For our sample server, the summary looks like this:



Use the ARROW keys to select the Done button and then press ENTER.

Due to the real potential for losing any existing data on the target disk, you will be presented with an extra warning/cautionary screen to confirm destructive action. We are installing our demo Ubuntu server on a brand-new and empty disk, so we select the Continue button and press ENTER.

NOTE You might get a different prompt if you are installing the operating system on a disk with existing partitions or volumes. In that case, you will need to confirm that you want to remove any existing partitions or volumes in order to continue.

Profile Setup

The next task in the Ubuntu server installation process is to set up a profile. This includes setting up a user account, creating a machine hostname for the server, and so on.

Complete the fields in the ensuing screen with the following information:

1. In the Your Name field, type in the full name **master admin**. Use the TAB or ENTER key to change to the next field.
2. Type **ubuntu-server** in the Your server's name field. Use the TAB or ENTER key to change to the next field.
3. Type **master** in the Pick a username field. Use the TAB or ENTER key to change to the next field.
4. Enter the password **72erty7!2** in the Choose a password field. Use the TAB or ENTER key to change to the next field.
5. Retype the same password in the Confirm your password field. Use the TAB or ARROW keys to select the Done button.
6. Press ENTER when the Done button is selected/highlighted

SSH Setup

The next task in the installation process is to install an OpenSSH server, import relevant SSH keys, and complete some minimal OpenSSH configuration tasks. A running OpenSSH server is useful for remotely accessing and managing systems and so we'll go ahead and install it here.

1. Use the TAB or ARROW keys to navigate to the Install OpenSSH server field.
2. Use the SPACE bar or ENTER keys on your keyboard to select/enable or put a check-mark on the Install OpenSSH server option.
3. We will not be importing any SSH keys or Usernames at this time. Accept the default setting (No) in the Import SSH identity and Username fields.
4. Accept any other remaining configuration defaults in the SSH Setup screen
5. Use the TAB or ARROW keys to select the final Done button.
6. Press ENTER when the Done button is selected/highlighted.

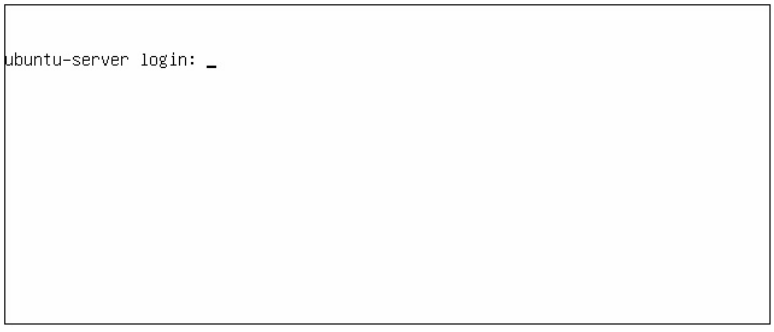


Figure 2-4 Classic Ubuntu Linux login prompt

Featured Server Snaps

The concept of *snaps* in the Ubuntu server world is a different implementation of preconfigured package groups, add-ons, modules, or software collections than in the Fedora/RedHat/CentOS world. Snaps provide a specific functionality and are self-contained file systems, apps, libraries, and so on.

We will not be installing any snaps at this time on our demo Ubuntu server. Use the TAB or ARROW keys to select Done and then press ENTER.

Install Complete

If you blink your eyes, you'll miss it—the installation will be done!

Select the Reboot button and press ENTER. You might be prompted to remove the installation media (flash drive, SD card, optical media, and so on) and then press ENTER.

Once the system reboots, you will be presented with a simple login prompt (see Figure 2-4). You can log in as the user that was previously created during the installation. The username is **master** and the password is **72erty7!2**.

Summary

You have successfully completed the installation process. If you are having problems with the installation, be sure to visit Fedora's web site at https://getfedora.org or the Ubuntu web site at www.ubuntu.com and take a look at the various manuals and tips available.

The version release notes are also a good resource for specific installation issues. Even though the install process discussed in this chapter used Fedora and Ubuntu as the operating systems of choice, you can rest assured that the installation concepts for other Linux distributions are virtually identical.

We sprinkled (and discussed) a liberal amount of GNU Linux/FOSS-specific concepts (for example, hard disk naming conventions, partitioning, volume management, network configuration, software management, and so on) throughout our installation walkthrough. These concepts will be covered in more detail in later chapters.

This page intentionally left blank

CHAPTER 3

Deploying Linux Servers in the Cloud

With the prevalence of cloud providers today, it's more feasible and easier than ever for everybody to launch and run their own Linux servers. End users have a wide variety of providers and implementations to select from, such as OpenStack, Linode, Amazon Web Services (AWS), Google Cloud Platform (GCP), Azure, and IBM Cloud.

The speed and ease with which a brand-new Linux server distribution can be brought online almost makes the hard work we put into Chapter 2 seem like a waste! Almost everything comes prebaked for you in the cloud. You only need to know how to turn on or off the right switches to get up and running in the cloud—and have the appropriate billing information handy, of course. With this in mind, you can view this chapter as an alternate Linux server installation chapter—but without any actual installation!

Behind the Cloud

Cloud environments are powered by a gazillion virtual machines (VMs). VMs are powered by the operating system (OS). The OS is powered by hardware or physical machines—loads and loads of hardware. You can rest assured there are no magical unicorns making things happen in the cloud. Cloud environments are powered by hardware.

What this means for you as a Linux system administrator is that *more than ever,* your skills and knowledge are heavily needed—even in a cloud-driven world. So instead of thinking in terms of a few physical hard drives attached to servers, you now think in terms of petabytes and pools of storage to be shared by potentially hundreds of smaller VMS. Instead of thinking in terms of terabytes of installed physical memory (RAM), you now think in terms of splitting that memory into smaller sizes to be shared by lots of smaller virtual machines (or containers). Instead of thinking in terms of a couple of Ethernet cards on a box, you think in terms of hundreds of virtual network cards and switches all communicating using the magic of software-defined networking (SDN). In a nutshell, this all means that the work (and concerns) of a system administrator increases by orders of magnitude in a cloud environment!

Obtaining and Spinning Up New Virtual Linux Servers

In this section, we talk about some of the many quick methods for spinning up a new Linux server—but using other people's hardware and infrastructure! Once it is successfully spun up, you should be able to apply the theories and follow the exercises in the rest of the book, just as if you were following along on your own physical machine.

Because we've not yet covered some other pertinent details (such as working on the command line), our coverage here is necessarily going to be very high level. The virtualization chapter later on delves deeper into this topic. At a minimum, you will need access to the command-line interface of an existing system (preferably Linux), with the right tools installed, in order to be able to use the commands detailed in the next sections. Even though we prefer Linux-based systems, you won't be stranded if you are following along on another platform such as macOS or Windows—you'll just have to consult the relevant provider documentation for how to obtain and set up the tools on either of those platforms.

Free-to-Run Virtual Linux Servers

By "free-to-run," we mean you don't necessarily need to give your billing information or credit card details to any third-party provider to get a virtual Linux server up and running. As a result, this often involves some hands-on prep work to be done by you or somebody else! We only provide a few examples of this type of server in this section.

virt-builder

This is a utility for downloading or building new virtual machines via customizable templates. It is easy to use, and all you need is access to an existing, running Linux operating system. The authors of **virt-builder** have created a decent library of templates for building various versions and spins of popular Linux distributions, such as Fedora, Ubuntu, Debian, OpenSUSE, and CentOS. After installing the software package that provides the **virt-builder** utility, you can use it to query and view a list of available operating systems by running the following command:

```
$ virt-builder -list
opensuse-tumbleweed      x86_64     openSUSE Tumbleweed
centos-8.0               x86_64     CentOS 8.0
debian-10                x86_64     Debian 10 (stretch)
fedora-34                x86_64     Fedora® 34 Server
freebsd-11.1             x86_64     FreeBSD 11.1
…<OUTPUT TRUNCATED>…
```

To view any relevant installation notes (such as login passwords, usernames, and so on) that might be available for the fedora-34 distro returned in the preceding sample listing, type this command:

```
$ virt-builder --notes fedora-34
```

Next, run the following command to build and download the disk image file for our sample fedora-34 distro. By default, the file will be downloaded to the current working directory.

```
$ virt-builder fedora-34
```

After successfully downloading the raw virtual disk image file (fedora34.img in our example), you should be able to use any garden-variety hypervisor platform to boot up and run the VM encapsulated in the image file. Chapter 30 covers how to use the **virt-install** utility to import and run disk images on the native Linux Kernel hypervisor (KVM).

OpenStack

OpenStack is an amalgamation of several individual FOSS projects that can be integrated to provide a single and complete cloud computing platform suitable for public and private clouds. Collectively, the individual projects are responsible for powering some very important services that OpenStack relies heavily on, such as Compute (Nova), Networking (Neutron), Block Storage (Cinder), Identity (Keystone), Image (Glance), Object Storage (Swift), Database (Trove), and Messaging (Zaqar). The OpenStack project has the backing of lots of technology industry stalwarts that use the project as well as contribute to its development. Once OpenStack is properly set up and configured, you are limited only by your imagination for what you can do with it.

Although the individual projects that comprise OpenStack usually have their own native tools (and quirks), the overarching **openstack** binary aims to be a unified tool that can be used to perform numerous functions across the entire stack. You can learn more about and download the **openstack** client toolset from https://pypi.org/project/python-openstackclient/.

After configuring and authorizing the tool to work on the target **openstack** deployment, you can bring up a new Linux server VM by supplying the correct parameters and running the following:

```
$ openstack server create \
--flavor <FLAVOR_ID> --image <IMAGE_ID> --key-name <KEY_NAME> \
--security-group <SEC_GROUP_NAME>   <INSTANCE_NAME>
```

Commercial Cloud Service Providers

Commercial cloud providers are any of the many for-profit companies that rent out parts of their compute infrastructure for a fee. Even though most of these providers have so-called free tiers of their services, you can rest assure that there is no such thing as a completely free lunch. The free tiers are designed to wet your appetite just enough to get you addicted/hooked on their services. Most commercial service providers require you to sign up and establish security tokens and keys that are unique to that service or provider. Generally, to use the command-line interface (CLI) tools in the following sections, you need to have security tokens handy!

Application Programming Interface (API)

One phrase you'll come across very frequently in the cloud world is application programming interface, or API. Here, we explain how APIs came to be and what problems they solve.

When you are running your workloads in the cloud, you are relying on the cloud provider's physical infrastructure. This means that you don't have access to physical switches, buttons, cables, ports, or anything like you would have in your own server closet or data center. The only way to manage your environment is via whatever virtual (software) interfaces the provider makes available to you. The interface of choice that most providers make available is via an API. APIs can be crudely defined as a canned set of rules designed to complete or trigger certain functions that a provider makes available to authorized users. For example, a provider can have a set of APIs that end users can invoke to launch a brand-new Fedora server VM with 128GB of RAM in a data center located in Antarctica. APIs can help provide a software abstraction layer that end users can interact with in a controlled manner.

The net result of all this is that OpenStack, Google, AWS, Azure, Bob, and his uncle all have their own unique APIs that you must use to interact with them. Thankfully, most of the APIs are implemented via a common and well-understood Representational State Transfer (REST) interface. Once you are authorized and understand the nuances of a system's API, you can easily bring up a single or a thousand new Linux servers using a few well-crafted commands!

Linode

Linode has a long and rich history in the Infrastructure as a Service (IaaS) world, specifically in the area of its Linux-based distro offerings. A VM in the Linode world is often referred to as a "linode" (Linux node).

Besides the graphical web front-end for managing resources in the Linode cloud, end users also have the option of a rich command line interface aptly named Linode CLI. The Linode CLI is a wrapper around the rich Linode API. You can download and learn more about Linode CLI from https://github.com/linode/linode-cli.

Once it's properly configured and you supply the correct parameters, you should be able to bring up a sample linode VM under your account with a command similar to the one here:

```
$ linode-cli linodes create \
--type g5-standard-2  --region us-east \
--image <IMAGE_NAME>  --root_pass
```

Amazon Web Services (AWS)

AWS provides cloud computing services and products for various applications and industries. There are literarily hundreds of services under the AWS umbrella. Amazon Elastic Compute Cloud (EC2) is the specific AWS offering that provides the compute infrastructure that interests us here.

AWS and numerous third parties have curated a decent collection of virtual machines on the EC2 platform. The AWS homegrown tool for interacting with its various cloud services is called **aws** or **aws-cli**. You can learn more about the aws tool at https://aws.amazon.com/cli/.

After signing up for an AWS account and properly configuring the tool, you can launch a new VM in AWS by supplying the correct values for the parameters (Amazon Machine Image ID, number of instances, and so on) in the following sample command:

```
$ aws ec2 run-instances \
--image-id <AMAZON_MACHINE_IMAGE_ID>  \
--count <NUMBER_OF_INSTANCES> --instance-type <TYPE_OF_INSTANCE> \
--key-name <KEY_PAIR_NAME> --security-group-ids  <SECURITY_GROUP_ID>
```

Google Cloud Platform (GCP)

GCP is made up of a plethora of services—and the number of services keeps growing. The specific component under GCP that is closest to offering a traditional Linux (virtual) server experience to end users is the Google Compute Engine (GCE). Google's homegrown command-line toolset for end users to interact with its products and service under GCP is delivered via its Cloud SDK. The specific tool for interacting with GCE is called **gcloud**. You can download and learn more about the Cloud SDK and **gcloud** at https://cloud.google.com/sdk/.

After signing up for a GCP account and setting up your environment, bringing up a new Linux server can be as simple as supplying the correct values for the parameters (image family, image project, and so on) and running the following command:

```
$ gcloud compute instances create  <INSTANCE_NAME> \
--image-family <IMAGE_FAMILY> \
--image-project <IMAGE_PROJECT>
```

Azure

Azure is Microsoft's cloud computing arm. As of the time of this writing, it comprises over several hundred different cloud services! The Azure component we are interested in here is aptly name "Virtual Machines," and it allows end users to create Linux and Windows virtual machines on Azure's IaaS platform.

Azure CLI (**az**) is the command-line tool for managing various things in the Azure world. You can download and learn more about this tool at https://github.com/Azure/azure-cli.

As with the other commercial public cloud providers, you'll need to sign up for an account before you can start spinning up servers on Azure's infrastructure. Once you have the **az** tool properly set up and you've created any needed resources (such as a resource group), you should be able to spin up a sample Ubuntu server in Azure by running a command similar to the one here:

```
$ az  vm create \
--resource-group <Resource_Group_Name> \
--name <VM_Name> --image <Image_Name>
```

Summary

Even though almost everything in the cloud is virtual, it is still very important that you as the system administrator have a good understanding of the main principles of administering physical servers. The concepts are very similar; only the scale is different. The rest of this book covers a good chunk of the main principles of Linux server administration, such that you will be ready to work on both physical and virtual cloud-based servers.

As we wrote at the beginning of this chapter, one of the beauties of the cloud computing model is that almost everything comes prebaked for you in the cloud. However, to take full advantage of the ease of use and accessibility of the cloud, you need to be able to speak the language of the provider and play by their rules—and this is often done via APIs and custom tools wrapped around the APIs. Chapter 30 provides more insight into the world of virtualization on which all cloud environments are built.

PART

II

Single-Host Administration

CHAPTER 4
The Command Line

CHAPTER 5
Managing Software

CHAPTER 6
Managing Users and Groups

CHAPTER 7
Booting and Shutting Down

CHAPTER 8
File Systems

CHAPTER 9
Core System Services

CHAPTER 10
The Linux Kernel

CHAPTER 11
Knobs and Dials: API (Virtual) File Systems

This page intentionally left blank

CHAPTER

4 The Command Line

The level of power, control, and flexibility that the command line offers Linux/FOSS users has been one of its most endearing and enduring qualities. There is also a flip side to this, however: for the uninitiated, the command line can also produce extremes of emotions, including awe, frustration, and annoyance. Casual observers of Linux gurus at work are often astounded at the results of a few carefully crafted and executed commands. This awesomeness comes at a cost—it can make using Linux appear less intuitive to the average user. For this reason, graphical user interface (GUI) front-ends for various Linux tools, functions, and utilities have been written.

More experienced users, however, may find that it is difficult for a GUI to present all of the available options. Typically, doing so would make the interface just as complicated as the command-line equivalent. The GUI design is often oversimplified, and experienced users ultimately return to the comprehensive capabilities of the command line. After all is said and done, the fact remains that it just looks plain *cool* to do things at the command line!

Before we begin our study of the command-line interface (CLI) under *GNU's Not Unix* (GNU)/Linux-based systems, understand that this chapter is far from an exhaustive resource. Rather than trying to cover all the tools without any depth, this chapter describes a handful of tools that are most critical for the day-to-day work of a system administrator.

NOTE This chapter assumes that you are logged into the system as a regular/non-privileged user. You can follow the examples here whether you are logged into the system at the console or via a GUI desktop environment.

If you are using the GNOME desktop environment, for example, you can start a virtual terminal in which to issue commands. To launch a virtual terminal application, press the ALT-F2 key combination on your keyboard to bring up the Run Application dialog box. Type the name of a terminal emulator (for example, **xterm**, **gnome-terminal**, or **konsole**) into the Run text box and then press ENTER. You can alternatively look under the Applications menu for any of the installed terminal emulator applications. All the commands you enter in this chapter should be typed into a virtual terminal or console at the shell prompt.

An Introduction to Bash

In Chapter 6, you will learn that an important component of creating new users pertains to the user's login shell. The login shell is the first program that runs when a user logs into a system. The shell is comparable to the Windows Program Manager, except that in the case of Linux, the system administrator (and user) has a say in the choice of shell program used.

The formal definition of a shell is "a command language interpreter that executes commands." A less formal definition might be simply "a program that provides an interface to the system." The Bourne Again Shell (Bash), in particular, is a command-line-only interface containing a handful of built-in commands; it has the ability to launch other programs and to control programs that have been launched from it (job control).

A variety of shells exist, most with similar features but different means of implementing them. Again, for the purpose of comparison, you can think of the various shells as being like web browsers; among several different browsers, the basic functionality is the same—displaying content from the Web. In any situation like this, everyone proclaims that his or her shell is better than the others, but it all really comes down to personal preference.

In this section, we'll examine some of Bash's built-in commands. A complete reference on Bash could easily fill a large book in itself, so we'll stick with the commands that a system administrator (or regular user) might use frequently. However, it is highly recommended that you eventually study Bash's other functions and operations. The differences between Bash and other popular shells are very subtle, but it isn't a bad idea for a system administrator to be as familiar with as many shells (and their idiosyncrasies) as possible.

Job Control

When working in the Bash environment, you can start multiple programs from the same prompt. Each program is considered a *job.* Whenever a job is started, it takes over the terminal. On today's machines, the terminal is either a straight stand-alone text/console interface or a window displayed within a graphical environment (Xorg, WayLand, and so on). The terminal interfaces in graphical environments are called a *pseudo-tty,* or *pty* for short. If a job has control of the terminal, it can issue control codes so that text-only interfaces can be made more attractive (via colorization or other visual cues). Once the program is done, it gives full control back to Bash, and a prompt is redisplayed for the user.

Not all programs require this kind of terminal control, however. Some, including programs that interface with the user through the X Window System, can be instructed to give up terminal control and allow Bash to bring back (re-present) the user prompt, even though the invoked program is still running.

In the following example, with the user master logged into the system via a graphical desktop environment, the user launches the Firefox web browser from the CLI or shell, with the additional condition that the program (Firefox) gives up control of the terminal (this condition is specified by appending the ampersand symbol to the program name):

```
[master@server ~]$  firefox &
```

Immediately after you press ENTER, Bash will present its prompt again. This is called *backgrounding* the task.

If a program is already running and has control of the terminal, you can make the program give up control by pressing CTRL-Z in the terminal window. This will stop the running job (or program) and return control to Bash. At any given time, you can find out how many jobs Bash is tracking by typing this command:

```
[master@server ~]$  jobs
[1]+  Running                 firefox &
```

The running programs that are listed will be in one of two states: running or stopped. The preceding sample output shows that the Firefox program is in a running state. The output also shows the job number in the first column: `[1]`.

To bring a job back to the foreground—that is, to give it back control of the terminal—you would use the **`fg`** (foreground) command, like this:

```
[master@server ~]$  fg NUMBER
```

Here, ***`NUMBER`*** is the job number you want in the foreground. For example, to place the Firefox program (with job number 1) launched earlier in the foreground, type this:

```
[master@server ~]$  fg  1
firefox
```

If a job is stopped (that is, in a stopped state), you can start it running again in the background, thereby allowing you to keep control of the terminal and resume running the job. Or a stopped job can run in the foreground, which gives control of the terminal back to the program.

To place a running job in the background, type this:

```
[master@server ~]$  bg NUMBER
```

Here, ***`NUMBER`*** is the job number you want to background.

NOTE You can background any process. Applications that require terminal input or output will be put into a stopped state if you background them. You can, for example, try running the **`top`** utility in the background by typing **`top &`**. Then you can check the state of that job with the **`jobs`** command.

Environment Variables

Every instance of a shell, and every process that is running, has its own "environment"—these are settings that give it a particular look, feel, and, in some cases, behavior. These settings are typically controlled by environment variables. Some environment variables have special meanings to the shell, but there is nothing stopping you from defining your own and using them for your own needs. It is through the use of environment variables that most shell scripts are able to do interesting things and remember results from user inputs as well as program outputs. If you are already familiar with the concept of environment variables in Microsoft Windows, you'll find that many of the things you know about them will apply to Linux as well; the only difference is how they are set, viewed, and removed.

Printing Environment Variables

To list all of your environment variables, use the **printenv** command. Here's an example:

```
[master@server ~]$  printenv
HOSTNAME=server
TERM=xterm
SHELL=/bin/bash
HISTSIZE=1000
...<OUTPUT TRUNCATED>...
```

To show a specific environment variable, specify the variable as a parameter to **printenv**. For example, here is the command to see the environment variable **TERM**:

```
[master@server ~]$  printenv   TERM
xterm
```

Setting Environment Variables

To set an environment variable, use the following format:

```
variable=value
```

Here, ***variable*** is the variable name and ***value*** is the value you want to assign the variable. For example, to set the environment variable **FOO** to the value **BAR**, type this:

```
[master@server ~]$  FOO=BAR
```

Whenever you set environment variables in this way, they stay local to the running shell. If you want that value to be passed to other processes that you launch, use the **export** built-in command. The format of the **export** command is as follows:

```
export   variable
```

Here, ***variable*** is the name of the variable. For example, to set the value of the variable **FOO** and export it at the same time, you would enter this command:

```
[master@server ~]$  export  FOO=BAR
```

If the value of the environment variable you want to set has spaces in it, surround the variable with quotation marks. Using the preceding example, to set **FOO** to "Welcome to the BAR of FOO.", you would enter this:

```
[master@server ~]$  export  FOO="Welcome to the BAR of FOO."
```

You can then use the **printenv** command to see the value of the **FOO** variable you just set by typing this:

```
[master@server ~]$  printenv  FOO
Welcome to the BAR of FOO.
```

NOTE Modern cloud-native utilities and web application stacks make heavy use of environment variables to set parameters on the fly that help to serve various purposes, such as authentication, configuring runtime variables, and so on. Therefore, you might see instructions asking you to, for example, set your AWS secret key credentials by setting the following environment variable at your shell: AWS_SECRET_ACCESS_KEY=EXAMPLEI/K7MDENG/bPxRfiCY.

Unsetting Environment Variables

To remove an environment variable, use the **unset** command. Here's the syntax for the **unset** command:

```
unset variable
```

Here, ***variable*** is the name of the variable you want to remove. For example, here's the command to remove the environment variable **FOO**:

```
[master@server ~]$  unset  FOO
```

NOTE This section assumes that you are using Bash. You can choose to use many other shells; the most popular alternatives are the C shell (csh) and its brother, the Tenex/Turbo/Trusted C shell (tcsh), which uses different mechanisms for getting and setting environment variables. Bash is documented here because it is often the default shell for new Linux user accounts in most Linux distributions.

Pipes

Pipes are a mechanism by which the output of one program can be sent as the input to another program. Individual programs can be chained together to become extremely powerful tools using pipes.

Let's use the **grep** program to provide a simple example of how pipes can be used. When given a stream of input, the **grep** utility will try to match the line with the parameter supplied to it and display only matching lines. You will recall from the preceding section that the **printenv** command prints all the environment variables. The list it prints can be lengthy, so, for example, if you were looking for all environment variables containing the string "TERM", you could enter this command:

```
[master@server ~]$  printenv | grep TERM
TERM=xterm
```

The vertical bar (|) character represents the pipe between **printenv** and **grep**.

The command shell under Windows also utilizes the pipe function. The primary difference is that all commands in a Linux pipe are executed concurrently, whereas Windows runs each program in order, using temporary files to hold intermediate results.

Redirection

Through *redirection,* you can take the output of a program and have it automatically sent to a file (remember that everything in Linux is regarded as a file!). The shell rather than the program itself handles this process, thereby providing a standard mechanism for performing the task. Having the shell handle redirection is therefore much cleaner and easier than having individual programs handle redirection themselves.

Redirection comes in three classes: output to a file, append to a file, and send a file as input.

To send the output of a program into a file, end the command line with the greater-than symbol (>) and the name of the file to which you want the output redirected. If you are redirecting to an existing file and you want to append additional data to it, use two symbols (>>) followed by the filename. For example, here is the command to send the output of a directory listing into a file called **/tmp/directory_listing**:

```
[master@server ~]$   ls   >  /tmp/directory_listing
```

Continuing this example with the directory listing, you could append the string "Directory Listing" to the end of the **/tmp/directory_listing** file by typing this command:

```
[master@server ~]$  echo  "Directory Listing" >> /tmp/directory_listing
```

The third class of redirection, using a file as input, is done by using the less-than sign (<) followed by the name of the file. For example, here is the command to feed the **/etc/passwd** file into the **`grep`** program:

```
[master@server ~]$  grep   "root"   <   /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
```

Command-Line Shortcuts

Most of the popular Linux shells have a tremendous number of shortcuts. Learning and getting used to the shortcuts can be a huge cultural shock for users coming from the Windows world. This section explains the most common of the Bash shortcuts and their behaviors.

Filename Expansion

Under traditional Linux-based shells such as Bash, wildcards on the command line are expanded before being passed as a parameter to the application. This is in sharp contrast to the default mode of operation for DOS-based tools, which often have to perform their own wildcard expansion. The Linux method also means that you must be careful where you use the wildcard characters. The wildcard characters themselves in Bash are identical to those in **cmd.exe** in the Windows world.

The asterisk (*) matches against all filenames, and the question mark (?) matches against single characters. If you need to use these characters as part of another parameter for whatever reason, you can escape them by preceding them with a backslash (\) character. This causes the shell to interpret the asterisk and question mark as regular characters instead of wildcards.

NOTE In Linux, wildcard patterns are not exactly the same as regular expressions, although there are some similarities. Wildcards are used for matching filenames, and regular expressions are used for matching text.

The distinction is important, since regular expressions are substantially more powerful than just wildcards alone. All of the shells that come with GNU/Linux support regular expressions. You can learn more about wildcards and regular expressions in their respective manual pages (**`man 7 glob`** and **`man 7 regex`**).

Environment Variables as Parameters

Under Bash, you can use environment variables as parameters on the command line.

For example, issuing the parameter **`$FOO`** will cause the value of the **`FOO`** environment variable to be passed rather than the string "$FOO".

Multiple Commands

Under Bash, multiple commands can be executed on the same line by separating the commands with semicolons (;). For example, here's how to execute this sequence of commands (**`cat`** and **`ls`**) on two separate lines:

```
[master@server ~]$  ls  -l
[master@server ~]$  cat  /etc/passwd
```

You could instead type the following:

```
[master@server ~]$  ls -l ; cat /etc/passwd
```

Since the shell is also a programming language, it understands the semantics of programming languages. For example, you can run commands serially *only* if the first command succeeds. This can be done by using the double ampersand (&&) symbol. For example, you can use the **`ls`** command to try to list a file that does *not* exist in your home directory, and then execute the **`date`** command right after that on the same line:

```
[master@server ~]$  ls  does-not-exist.txt  &&  date
ls: cannot access does-not-exist.txt: No such file or directory
```

This command will run the **`ls`** command, but that command will fail because the file it is trying to list does not exist, and, therefore, the **`date`** command will not be executed either. But if you switch the order of commands around, you will notice that the **`date`** command will succeed, while the **`ls`** command will fail:

```
[master@server ~]$  date  &&  ls  does-not-exist.txt
Sun Jan 30 18:06:37 UTC 2174
ls: cannot access does-not-exist.txt: No such file or directory
```

Backticks

Any text enclosed within backticks (`) is treated as a command to be executed. This allows you to embed commands within backticks and pass the result as parameters to other commands, for example. You'll see this technique used often in this book and in various system scripts.

For example, you can pass the value of a number (a process ID number) stored in a file and then pass that number as a parameter to the **kill** command. A sample use of this is for killing (stopping) the Domain Name System (DNS) server **named (or other services/daemons that works the same way)**. When **named** starts, it writes its process identification (PID) number into the file **/var/run/named/named.pid**. Thus, the generic and dirty way of killing the **named** process is to look at the number stored in **/var/run/named/named.pid** using the **cat** command, and then issue the **kill** command with that value. Here's an example:

```
[root@server ~]$  cat  /var/run/named/named.pid
253
[root@server ~]$  kill 253
```

One problem with killing the **named** process in this way is that it cannot be easily automated—we are counting on the fact that a human will read the value in **/var/run/named/named.pid** in order to pass the **kill** utility the number. Another issue isn't so much a problem as it is a nuisance: It takes two steps to stop the DNS server.

Using backticks, however, you can combine the steps into one and do it in a way that can be automated. The backticks version would look like this:

```
[root@server ~]$  kill   `cat /var/run/named/named.pid`
```

When Bash sees this command, it will first run **cat /var/run/named/named.pid** and store the result. It will then run **kill** and pass the stored result to it. From our point of view, this happens in one graceful step.

NOTE So far in this chapter, we have looked at features that are internal to Bash (or "Bash built-ins" as they are sometimes called). The remainder of the chapter explores several common commands accessible outside of Bash.

Documentation Tools

Linux-based systems comes with two superbly useful tools for making documentation accessible: **man** and **info**. Currently, a great deal of overlap exists between these two documentation systems, because many applications are moving their documentation to the **info** format. This format is considered superior to **man** because it allows the documentation to be hyperlinked together in a web-like way, but without actually having to be written in Hypertext Markup Language (HTML) format.

The **man** format, on the other hand, has been around for decades. For thousands of Linux utilities/programs, their man (short for *manual)* pages are their only source of documentation. Furthermore, many applications continue to utilize the **man** format because many other UNIX-like operating systems use it.

TIP Many Linux distributions also include a great deal of documentation in the **/usr/doc** or **/usr/ share/doc** directory.

The man Command

Man pages are a form of built-in and self-documentation system on Linux systems that cover the use of tools and their corresponding configuration files. The syntax of the **man** command is as follows:

```
man program_name
```

Here, ***program_name*** identifies the program in which you're interested. For example, to view the man page for the **ls** utility that we've been using, type this:

```
[master@server ~]$  man ls
```

While reading about Linux and Linux-related information sources online, you may encounter references to commands followed by numbers in parentheses—for example, **ls** (1). The number represents the section of the manual pages (see Table 4-1). Each section covers various subject areas to accommodate the fact that some tools (such as **printf**) are commands/functions in the C programming language as well as command-line commands.

To refer to a specific man page section, simply specify the section number as the first parameter and then the command as the second parameter. For example, to get the C programmers' information on **printf**, you'd enter this:

```
[master@server ~]$  man 3 printf
```

To get the plain command-line information (user tools), you'd enter this:

```
[master@server ~]$  man 1 printf
```

If you don't specify a section number with the **man** command, the default behavior is that the lowest applicable section number gets printed first.

Manual Section	Subject
1	Standard commands, executable programs, or shell commands
1p	POSIX versions of standard commands (the lowercase *p* stands for POSIX)
2	Linux kernel system calls (functions provided by the kernel)
3	C library calls
4	Device driver information
5	Configuration files
6	Games
7	Packages
8	System tools

Table 4-1 Man Page Sections

TIP A handy option to the **man** command is an **-f** preceding the command parameter. With this option, **man** will search the summary information of all the man pages and list pages matching your specified command, along with their section number. Here's an example:

```
$ man -f printf
printf (3p)           - print formatted output
printf (1)            - format and print data
```

The texinfo System

Another common form of documentation is texinfo. Established as the GNU standard, texinfo is a documentation system similar to the hyperlinked World Wide Web format. Because documents can be hyperlinked together, texinfo is often easier to read, use, and search in comparison to man pages.

To read the texinfo documents on a specific tool or application, invoke **info** with the parameter specifying the tool's name. For example, to read about the **wget** program, you'd type:

```
[master@server ~]$ info  wget
```

In general, you will want to verify whether a man page exists before using **info** (because there is still a great deal more information available in man format than in texinfo). On the other hand, some man pages will explicitly state that the texinfo pages are more authoritative and should be consulted instead.

Files (Types, Ownership, and Permissions)

This section covers basic file management tools and concepts under Linux. We'll start with specifics on some useful general-purpose commands, and then we'll step back and look at some background information.

Under Linux, almost everything is abstracted to a file. Originally, this was done to simplify the programmer's job. Instead of having to communicate directly with device drivers, special files (which look like ordinary files to the application) are used as a bridge. We discuss the different types of file categories in the following sections.

Normal Files

Normal files are just that—normal. They contain data and can also be executables. The operating system makes no assumptions about their contents or their names or extensions.

Directories

Directory files are a special instance of normal files. Directory files list the locations of other files, some of which may be other directories. This is similar to folders in Windows.

In general, the nitty-gritty of directory files won't be of importance to your daily operations, unless you need to open and read the file yourself rather than using existing applications to navigate directories; doing this would be similar to trying to read the DOS file allocation table directly rather than using **cmd.exe** to navigate directories or using the **findfirst**/**findnext** system calls.

Hard Links

Each file in the Linux file system gets its own i-node. An *i-node* keeps track of a file's attributes and its location on the disk. If you need to be able to refer to a single file using two separate filenames, you can create a hard link. The *hard link* will have the same i-node as the original file and will, therefore, look and behave just like the original.

With every hard link that is created, a reference count is incremented. When a hard link is removed, the reference count is decremented. Until the reference count reaches zero, the file will remain on disk.

NOTE A hard link cannot exist between two files on separate file systems (or partitions). This is because the hard link refers to the original file by i-node, and a file's i-node is only unique on the file system on which it was created.

Symbolic Links

Unlike hard links, which point to a file by its i-node, a *symbolic link* points to another file by its name. This allows symbolic links (often abbreviated symlinks) to point to files located on other file systems, even other network drives.

Block Devices

Traditional hard disks are a type of block or storage device. All devices are accessed through their device file abstractions on the file system. Files of type *block device* are used to interface with devices such as disks. A block device file has two identifying traits:

- It has a major number.
- It has a minor number.

When viewed using the **`ls -l`** command, it shows **`b`** as the first character of the permissions field. Here's an example:

```
[master@server ~]$  ls -l /dev/sda
brw-rw----. 1 root disk 8, 0 Feb  3 19:03 /dev/sda
```

In the sample output, note the **`b`** at the beginning of the file's permissions; the **`8`** in the fifth field is the major number, and the **`0`** in the sixth field is the minor number.

A block device file's major number identifies the represented device driver. When this file is accessed, the minor number is passed to the device driver as a parameter, telling it which instance of the device it is accessing. For example, if there are two serial ports, they will share the same device driver and thus the same major number, but each serial port will have a unique minor number.

Character Devices

Similar to block devices, *character devices* are special files that allow you to access devices through the file system. The obvious difference between block and character devices is that block devices communicate with the actual devices in large blocks or chunks, whereas

character devices work one character at a time. A hard disk is a block device; a modem is a character device. Character device permissions start with a *c,* and the file has a major number and a minor number. Here's an example:

```
[master@server ~]$  ls  -l  /dev/ttyS0
crw-rw----. 1 root dialout 4, 64 Feb  3 19:03 /dev/ttyS0
```

Listing Files: ls

Out of necessity, we have been using the **ls** command in previous sections without properly explaining it. We will look at the **ls** command and some of its options here.

The **ls** command is used to list all the files in a directory. Of more than 50 available options, those listed in Table 4-2 are the most commonly used. The options can be used in any combination.

To see an extended (long) list of all files (including hidden files) in a directory, type:

```
[master@server ~]$  ls -la
```

To list a directory's non-hidden files that start with the letter *A,* type this:

```
[master@server ~]$  ls A*
```

If no such file exists in your working directory, **ls** prints out a message telling you so.

CAUTION Linux is case-sensitive. For example, a file named **thefile.txt** is different from a file named **Thefile.txt**.

Change Ownership: chown

The **chown** command allows you to change the ownership of a file to another user. Only the root (superuser) user can do this; therefore, normal users may not assign file ownership or steal ownership from another user. The syntax of the command is as follows:

```
chown  [-R]  username  filename
```

Option for ls	Description
-l	Long listing. In addition to the filename, shows the file size, date/time, permissions, ownership, and group information.
-a	All files. Shows all files in the directory, including hidden files. Names of hidden files begin with a period.
-t	Lists in order of last modified time.
-r	Reverses the listing.
-1	Single-column listing.
-R	Recursively lists all files and subdirectories.

Table 4-2 Common ls Options

Here, ***username*** is the login of the user to whom you want to assign ownership, and ***filename*** is the name of the file in question. The filename may be a directory as well.

The **-R** option applies when the specified filename is a directory name. This option tells the command to descend recursively through the directory tree and apply the new ownership to the named directory itself and all of the files and directories within it.

NOTE The **chown** command supports a special syntax that allows you to also specify a group name to assign to a file. The format of the command becomes this:

```
$ chown username.groupname filename
```


Change Group: chgrp

The **chgrp** command-line utility lets you change the group settings of a file. It works much like **chown**. Here is the format:

```
chgrp [-R] groupname filename
```

Here, ***groupname*** is the name of the group to which you want to assign filename ownership. The filename can be a directory as well.

The **-R** option applies when the specified filename is a directory name. As with **chown**, the **-R** option tells the command to descend recursively through the directory tree and apply the new ownership to all of the files and directories within it.

Change Mode: chmod

Directories and files within the Linux file system have permissions associated with them. By default, permissions are set for the owner of the file, the group associated with the file, and everyone else who can access the file (also known as owner, group, and other, respectively).

When you list files or directories, you see the permissions in the first column of the output. Permissions are divided into four parts. The first part is represented by the first character of the permission. Normal files have no special value and are represented with a hyphen (**-**) character. If the file has a special attribute, it is represented by a letter. The two special attributes we are most interested in here are directories (**d**) and symbolic links (**l**).

The second, third, and fourth parts of a permission are represented in three-character chunks. The first part indicates the file owner's permission. The second part indicates the group permission. The last part indicates the world permission. In the context of Linux, "world" means all users (everyone) in the system, regardless of their group settings.

Following are the letters used to represent permissions and their corresponding values. When you combine attributes, you add their values. The **chmod** command is used to set permission values.

Letter	Permission	Value (Numeric)
r	Read	4
w	Write	2
x	Execute	1

Using the numeric command mode is typically known as the *octal* permissions, since the value can range from 0 to 7. To change permissions on a file, you simply add or subtract these values for each permission you want to apply.

For example, if you want to make it so that only the user (owner) can have full access (read, write, and execute: **rwx**) to a file called **foo**, you would type this:

```
[master@server ~]$  chmod 700 foo
```

What is important to note is that using the octal mode *replaces* any permissions that were previously set. So if a file in the **/usr/local** directory is tagged with a SetUID bit, and you run the command **`chmod -R 700 /usr/local`**, that file will no longer be a SetUID program.

If you only want to change certain bits, you should use the symbolic mode of **`chmod`**. This mode turns out to be much easier to remember, and you can add, subtract, or overwrite permissions.

The symbolic form of **`chmod`** allows you to set the bits of the owner, the group, or others. You can also set the bits for all three classes at the same time.

For example, if you want to change a file called **foobar.sh** so that it is executable (**`x`**) for the owner (**`u`**), you can run the following command:

```
[master@server ~]$  chmod  u+x  foobar.sh
```

If you want to change the group's bit to execute also, use the following:

```
[master@server ~]$  chmod  ug+x  foobar.sh
```

If you need to specify different permissions for others, just add a comma and its permission symbols. For example, to make the **foobar.sh** file executable for the user and the group, but also remove read, write, and executable permissions for all others, you could try this:

```
[master@server ~]$  chmod   ug+x,o-rwx   foobar.sh
```

If you do not want to add or subtract a permission bit, you can use the equal (=) sign instead of a plus (+) sign or minus (–) sign. This will write the specific bits to the file and erase any other bit for that permission. The preceding examples used + to add the execute bit to the User and Group fields. If you want *only* the execute bit, you would replace the + with =. You can also use a fourth character: **`a`**. This will apply the permission bits to *all* the fields.

The following list shows the most common combinations of the three permissions. Other combinations, such as **`-wx`**, also exist, but they are rarely used.

Letter	Permission	Value
`---`	No permissions	0
`r--`	Read only	4
`rw-`	Read and write	6
`rwx`	Read, write, and execute	7
`r-x`	Read and execute	5
`--x`	Execute only	1

Permission	Numeric Equivalent	Description
`-rw-------`	600	Owner has read and write permissions.
`-rw-r--r--`	644	Owner has read and write permissions; group and world have read-only permission.
`-rw-rw-rw-`	666	Everyone has read and write permissions. Not recommended; this combination allows the file to be accessed and changed by anyone.
`-rwx------`	700	Owner has read, write, and execute permissions. Best combination for programs or executables that the owner wants to run.
`-rwxr-xr-x`	755	Owner has read, write, and execute permissions. Everyone else has read and execute permissions.
`-rwxrwxrwx`	777	Everyone has read, write, and execute permissions. Like the 666 setting, this combination should be avoided.
`-rwx--x--x`	711	Owner has read, write, and execute permissions; everyone else has execute-only permissions. Useful for programs that you want to let others run but not copy.
`drwx------`	700	This is a directory created with the **`mkdir`** command. Only the owner can read and write to this directory. Note that a directory must have the executable bit set, to make it traversable.
`drwxr-xr-x`	755	This directory can be changed only by the owner, but everyone else can view its contents.
`drwx--x--x`	711	A handy combination for keeping a directory world-readable but restricted from access by the **`ls`** command. A file can still be read only by someone who knows the exact filename.

Table 4-3 File Permissions

For each file, three of these three-letter chunks are grouped together. The first chunk represents the permissions for the owner of the file, the second chunk represents the permissions for the file's group, and the last chunk represents the permissions for all users on the system. Table 4-3 shows some permission combinations, their numeric equivalents, and their descriptions.

File Management and Manipulation

This section covers the basic command-line tools for managing files and directories. The use and functions of some of these tools are similar to their use on other operating systems.

Copy Files: cp

The **cp** command is used to copy files. It has a substantial number of options. See its man page for additional details. By default, this command works silently, displaying status information only if an error condition occurs. Following are the most common options for **cp**:

Option	Description
-f	Forces copy; does not ask for verification
-i	Interactive copy; before each file is copied, verifies with user
-R, **-r**	Copies directories recursively

First, let's use the **touch** command to create an empty file called **foo.txt** in the user master's home directory:

```
[master@server ~]$  touch  foo.txt
```

Then use the **cp** (copy) command to copy **foo.txt** to **foo.txt.html**:

```
[master@server ~]$  cp  foo.txt  foo.txt.html
```

To copy all files in the current directory ending in .html to the **/tmp** directory, type this:

```
[master@server ~]$  cp  *.html  /tmp
```

To interactively recopy all files in the current directory ending in .html to the **/tmp** directory, type this command:

```
[master@server ~]$  cp  -i  *.html  /tmp
cp: overwrite `/tmp/foo.txt.html'?
```

You will notice that using the interactive (**-i**) option with **cp** forces it to prompt or warn you before overwriting existing files with the same name in the destination. To continue the copy and overwrite the existing file at the destination, type **yes** or **y** at the prompt, like this:

```
[master@server ~]$  cp -i  *.html  /tmp
cp: overwrite `/tmp/foo.txt.html'? yes
```

Move Files: mv

The **mv** command is used to move files from one location to another. Files can be moved across partitions/file systems as well. Moving files across partitions involves a copy operation, and as a result, the move command can take longer. But you will find that moving files within the same file system is almost instantaneous.

Following are the most common options for **mv**:

Option	Description
-f	Forces move
-i	Interactive move

To move a file named **foo.txt.html** from **/tmp** to your present working directory, for example, you'd use this command:

```
[master@server ~]$  mv   /tmp/foo.txt.html    .
```

NOTE That last dot (.) is not a typo!! It literarily means "this directory."

Besides being used for moving files and folders around the system, **mv** can also be used simply as a renaming tool.

To rename the file **foo.txt.html** to **foo.txt.htm**, type the following:

```
[master@server ~]$  mv   foo.txt.html   foo.txt.htm
```

Link Files: ln

The **ln** command lets you establish hard links and soft links (see "Files (Types, Ownership, and Permissions)" earlier in this chapter). The general format of **ln** is as follows:

```
ln   original_file    new_file
```

Although **ln** has many options, you'll rarely need to use most of them. The most common option, **-s**, creates a symbolic link (similar to a shortcut) instead of a hard link.

To create a symbolic link called **link-to-foo.txt** that points to the original file called **foo.txt**, issue this command:

```
[master@server ~]$  ln  -s  foo.txt   link-to-foo.txt
```

Find a File: find

The **find** command lets you search for files using various search criteria. **find** has a large number of options that you can read about in its man page. Here is the general format of **find**:

```
find   start_directory [options]
```

start_directory is the directory from which the search should start.

To find all files in the current directory (that is, the "." directory) that have not been accessed in at least seven days, you'd use the following command:

```
[master@server ~]$  find  .  -atime  7
```

Type this command to find all files in your present working directory whose names are **core** and then delete them (that is, automatically run the **rm** command on the search result):

```
[master@server ~]$  find . -name core -exec rm {} \;
```

TIP The syntax for the **-exec** option with the **find** command as used here can be difficult to remember, so you can also use the **xargs** method instead of the **exec** option used in this example. Using **xargs**, the command would then be this:

```
$ find . -name 'core' | xargs rm
```

To find all files in your PWD whose names end in .txt (that is, files that have the .txt extension) and are also *less* than 100 kilobytes (KB) in size, issue this command:

```
[master@server ~]$  find . -name '*.txt' -size -100k
```

To find all files in your PWD whose names end in .txt and are also *greater* than 100KB in size, issue this command:

```
[master@server ~]$  find . -name '*.txt' -size +100k
```

File Compression: gzip

The **gzip** utility is used for reducing the size (compressing) or expanding files. It is able to achieve impressive compression ratios. By convention, the **.gz** extension or suffix is used for naming files compressed with **gzip**.

Note that **gzip** compresses the file in place, meaning that after the compression process, the original file is removed, and the only thing left is the compressed file.

To compress a file named **foo.txt.htm** in your PWD, type this:

```
[master@server ~]$  gzip  foo.txt.htm
```

And then to decompress it, use **gzip** again with the **-d** option:

```
[master@server ~]$  gzip  -d  foo.txt.htm.gz
```

Issue this command to use the best compression method (**-9** or **–best** option) to compress all files ending in *.htm* in your PWD:

```
[master@server ~]$  gzip  -9  *.htm
```

File Compression: bzip2

The **bzip2** tool uses a different compression algorithm that usually results in smaller files than those compressed with the **gzip** utility, and it uses semantics that are similar to **gzip**. In other words, **bzip2** offers better compression ratios in comparison to **gzip**.

By convention, file archives compressed using the **bzip2** utility usually have the **.bz** extension or suffix. For more information, read the man page on **bzip2** (**man bzip2**).

File Compression: xz

xz is a general-purpose data compression and decompression utility. **xz** purportedly produces better compression ratios than **gzip** and **bzip2**. The important bits behind **xz**'s compression algorithms are in the public domain—and this makes it a popular compression format for numerous open source projects that require compression.

Let's create a sample file named **foo2.txt.htm** and then compress it (using **xz**):

```
[master@server ~]$   touch  foo2.txt.htm
[master@server ~]$   xz  foo2.txt.htm
```

Use the **ls** command to verify that a new compressed file named **foo2.txt.htm.xz** was created for you:

```
[master@server ~]$  ls  -l  *.xz
```

Then decompress the **foo2.txt.htm.xz** file and use the **--keep** option with the **xz** utility so the original archive file is not deleted after successful decompression:

```
[master@server ~]$  xz  --keep  -d  foo2.txt.htm.xz
```

Create a Directory: mkdir

The **mkdir** command is used for creating directories or folders. An often-used option of the **mkdir** command is the **-p** option. This option will force **mkdir** to create parent directories if they don't exist already. For example, if you need to create **/tmp/bigdir/subdir/mydir** and the only directory that exists is **/tmp**, using **-p** will cause **bigdir** and **subdir** to be automatically created along with **mydir**.

To create a single directory called **mydir**, under the **/tmp** folder use this command:

```
[master@server ~]$  mkdir  /tmp/mydir
```

To create a directory tree like **bigdir/subdir/finaldir** in your PWD, type this:

```
[master@server ~]$  mkdir  -p  bigdir/subdir/finaldir
```

Remove Files or Directories: rm

The **rm** command is used for removing (deleting) files or directories. It is one of the more often used utilities on any system. By default, **rm** does not remove directories and, as such, you will have to pass it a specific option (**-r**) to make it remove a directory. This command also accepts the **-i** parameter, which makes it prompt you interactively before deleting anything.

Use the **touch** command to create a file called **myfile** and then remove the file:

```
[master@server ~]$ touch  myfile  &&  rm  myfile
```

To remove a directory located under the **/tmp** folder called **mydir**, you'd type this:

```
[master@server ~]$  rm  -r  /tmp/mydir
```

If you want to interactively (the **-i** option) get rid of all the directories from **bigdir** to **finaldir** that were created earlier, you'd issue this command:

```
[master@server ~]$  rm  -rf  bigdir/subdir/finaldir
```

TIP You can also use the popular **rmdir** command to delete directories.

Show Present Working Directory: pwd

Inevitably, you will find yourself at the terminal or shell prompt of an already logged-in workstation and you won't know where you are in the file system hierarchy or directory tree. To get this information, you need the **pwd** command. Its only task is to print the current working directory. To display your current working directory, use this command:

```
[master@server ~]$  pwd
/home/master
```

Tape Archive: tar

If you are familiar with the WinZip program, you are accustomed to the fact that the compression tool not only reduces file size but also consolidates files into compressed archives. Under GNU/Linux, this process is separated into two tools: **gzip** and **tar**.

The **tar** command combines multiple files into a single large file. It is separate from the compression tool, so it allows you to select which compression tool to use or whether you even want compression. In addition, **tar** is able to read and write to devices, thus making it a good tool for backing up to tape devices.

TIP Although the *tape archive,* or **tar**, program includes the word "tape," it isn't necessary to read or write to a tape drive when you're creating archives. In fact, you'll rarely use **tar** with a tape drive in day-to-day situations any longer (traditional backups aside).

Here's the syntax for the **tar** command:

```
tar option  filename
```

Some of the options for the **tar** command are shown here:

Option	Description
-c	Creates a new archive
-t	Views the contents of an archive
-x	Extracts the contents of an archive
-f	Specifies the name of the file (or device) in which the archive is located
-v	Provides verbose descriptions during operations
-j	Filters the archive through the **bzip2** compression utility
-z	Filters the archive through the **gzip** compression utility

In order to see sample usage of the **tar** utility, first create a folder called **junk** in the PWD that contains some empty files named **1**, **2**, **3**, **4**:

```
[master@server ~]$  mkdir  junk ;  touch junk/{1,2,3,4}
```

Now create an archive called **junk.tar** containing all the files in the folder called **junk** that you just created by typing this:

```
[master@server ~]$  tar  -cf  junk.tar junk
```

Create another archive called **2junk.tar** containing all the files in the **junk** folder, but this time, add the **-v** (verbose) option to show what is happening as it happens:

```
[master@server ~]$  tar  -vcf   2junk.tar  junk
junk/
junk/1
...
```

The archive that we just created was not compressed in any way. The files and directory have only been combined into a single file.

To create a **gzip**-compressed archive called **3junk.tar.gz** containing all of the files in the **junk** folder and to show what is happening as it happens, issue this command:

```
[master@server ~]$  tar  -cvzf   3junk.tar.gz   junk
```

To extract the contents of the **gzip**ped **tar** archive just created, issue this command:

```
[master@server ~]$  tar -xvzf   3junk.tar.gz
```

TIP The **tar** command is one of the few GNU/Linux utilities that cares about the order in which you specify its options. If you issued the preceding **tar** command as **tar -xvfz 3junk.tar.gz**, the command would fail, because the **-f** option was not immediately followed by a filename.

If you like, you can also specify a physical device to **tar** to and from. This is handy when you need to transfer a set of files from one system to another and for some reason you cannot create a file system on the device. (Or, sometimes, it's just more entertaining to do it this way!)

Assuming you have a USB disk plugged into your system and the USB device is mapped to **/dev/null**, you can try creating an archive on the USB disk by typing this:

```
[master@server ~]$  tar -cvzf /dev/null  junk
```

CAUTION The command **tar -cvzf /dev/null** will treat our phantom USB disk device (**/dev/null**) as a raw device and *erase* anything that is already on it. This is why, in our example, we deliberately (and incorrectly) used the null device (**/dev/null**), which will not result in any harm to your system.

The moral of this caution is to be careful when running any commands that write to real and raw devices, such as **/dev/sdb**, **/dev/vda1**, **dev/sr0** and so on.

To pull (extract) that archive off of a disk, you would type this:

```
[master@server ~]$  tar  -xvzf  /dev/null
```

Concatenate Files: cat

The **cat** program fills an extremely simple role: it concatenates and displays files. More creative things can be done with it, but nearly all of its usage will be in the form of simply displaying the contents of text files—much like the **type** command under Microsoft's CMD.

Because multiple filenames can be specified on the command line, it's possible to concatenate files into a single, large, continuous file. This is different from **tar** in that the resulting file has no control information to show the boundaries of different files.

To display the **/etc/passwd** file, use this command:

```
[master@server ~]$    cat  /etc/passwd
```

To display the **/etc/passwd** file and the **/etc/group** file, issue this command:

```
[master@server ~]$  cat  /etc/passwd /etc/group
```

Type this command to concatenate **/etc/passwd** with **/etc/group** and send the output into the file **users-and-groups.txt**:

```
[master@server ~]$  cat  /etc/passwd /etc/group > users-and-groups.txt
```

To append the contents of the file **/etc/hosts** to the **users-and-groups.txt** file you just created, type this:

```
[master@server ~]$  cat  /etc/hosts  >>  users-and-groups.txt
```

TIP If you want to **cat** a file in reverse, you can use the **tac** command.

Display a File One Screen at a Time: more or less

The **more** and **less** commands take an input file and display it one screen at a time. The input file can come either from its **stdin** or from a command-line parameter.

To view the **/etc/passwd** file one screen at a time, use this command:

```
[master@server ~]$  more   /etc/passwd
```

To view the **/etc/passwd** file one screen at a time, using the **less** command, type:

```
[master@server ~]$  less   /etc/passwd
```

To view the directory listing generated by the **ls** command one screen at a time, type:

```
[master@server ~]$  ls | more
```

Show the Directory Location of a File: which

The **which** command searches the locations specified in the PATH environment variable (**$PATH**) to find the name of an executable specified on the command line. If the file is found, the command output includes the actual path to the file.

Use the following command to find out in which directory the binary for the **rm** command is located:

```
[master@server ~]$   which   rm
/bin/rm
```

You might find this similar to the **find** command. The difference here is that since **which** searches only **$PATH**, it is much faster. Of course, it is also much more limited in features than **find**!

Locate a Command: whereis

The **whereis** tool searches the locations specified in the PATH and MANPATH environment variables and displays the name of the program and its absolute directory, the source file (if available), and the man page for the command (again, if available).

To find the location of the program, source, and manual page for the **grep** command, type:

```
[master@server ~]$   whereis   grep
grep:
/usr/bin/grep
/usr/share/man/man1/grep.1.gz
/usr/share/man/man1p/grep.1p.gz
```

Editors

Editors are easily among the bulkiest of the common system utilities, but they are also incredibly useful. Without them, making any kind of change to a text file would be a tremendous undertaking. Regardless of your Linux distribution, you will have gotten a few editors automatically installed for you during the operating system installation. You should take a few moments to get comfortable with them.

NOTE Different Linux distributions favor some editors over others. As a result, you might have to find and install your preferred editor if it doesn't come installed with your distribution by default.

vi

The **vi** editor has been around UNIX-based systems since the 1970s, and its interface shows it. It is arguably one of the last editors to use a separate command mode and data entry mode; as a result, most newcomers may find it unpleasant to use. But before you give **vi** the cold shoulder, take a moment to get comfortable with it. In difficult situations, you might not have the luxury of a pretty graphical editor at your disposal, but you will find that **vi** is ubiquitous across all GNU/Linux systems.

Another version of **vi** that is readily available for installation on most Linux distributions is **vim** (VI iMproved). It has a lot of what made **vi** popular in the first place and many features that make it relevant and useful in modern computing environments—there's even a GUI version of the editor!

To start **vi**, simply type this:

```
[master@server ~]$  vi
```

The **vim** editor has an online tutor that can help you get started with it quickly. To launch the tutor, type this:

```
[master@server ~]$  vimtutor
```

Another easy way to learn more about **vi** is to start it and enter **:help**. If you ever find yourself stuck in **vi**, press the ESC key several times and then type **:q!** to force an exit without saving. If you instead want to save the file you are editing and quit **vi**, type **:wq!**.

emacs

It has been argued that **emacs** can easily be an entire operating system all by itself! It's big, feature-rich, expandable, programmable, and all-around amazing. If you're coming from a GUI background, you'll probably find **emacs** a pleasant environment to work with at first. You might to install **emacs** if it doesn't come installed by default on your distro. On its face, it works like Notepad in terms of its interface. Yet underneath is a complete interface to the GNU development environment, a mail reader, a news reader, and a web browser. Believe it or not, it even has a cute built-in help system that's disguised as your very own personal psychotherapist! You can have some *interesting* conversations with this automated/robotic psychotherapist.

To start **emacs**, simply type the following:

```
[master@server ~]$  emacs
```

Once **emacs** has started, you can visit the therapist by pressing ESC-X and then typing **doctor**. To get help using **emacs**, press CTRL-H.

pico

The **pico** program is an editor inspired by simplicity. Typically used in conjunction with the Pine e-mail reading system, **pico** can also be used as a stand-alone editor. **joe** is another simple CLI text editor that functions similarly to **pine**. Both **pine** and **joe** work in a manner similar to Notepad, but **pico** has its own set of key combinations. Thankfully, all available key combinations are always shown at the bottom of the screen.

To start **pico**, simply type this:

```
[master@server ~]$  pico
```

TIP The **pico** program will perform automatic word wraps. If you're using it to edit configuration files, for example, be careful that it doesn't word-wrap a line into two lines if it should really be parsed as a single line.

sed

sed is a stream-based line editor. It is not a traditional file-editing program, but is instead a powerful purpose-built programming language that also happens to have editing functions in its repertoire. It is best suited for manipulating data that has some known patterns. The power of sed comes from its commands. It has commands for searching for patterns, appending text, replacing text, deleting text, printing text, and so on. One way that system administrators often make use of sed is making in-place edits of system configuration files by searching for keywords (configuration parameters) in the files. Let's look at some simple sed one-liners.

Use the **echo** command as the input source (instead of a file) and change the word *night* to the word *day* using sed's substitute (**s**) command:

```
[master@server ~]$ echo night | sed 's/night/day/'
day
```

Let's do something similar to the previous command, but this time use **sed** to do an in-place edit of a file.

Use **echo** to create a file named **contacts.txt** with two entries (**name: adere** on the first line and **phone: 555-723-9709** on the second line), like so:

```
[master@server ~]$ echo -e 'name: adere\nphone: 555-723-9709' > contacts.txt
```

We'll first do a dry run, which will only print (**p**) what will be done. Use **sed**'s substitute (**s**) command and regex (regular) expression to search for all lines in the file that begin with the string "name: " and change it to the new line "name: hiromi":

```
[master@server ~]$ sed -n 's/^name:\s*.*$/name: hiromi/p' contacts.txt
name: hiromi
```

The output from the test run looks like what we want. We will now use **sed**'s in-place (**-i** or **--in-place**) option to make the change and also save a backup of the original unchanged file to a new file named **contacts.txt.bak** by typing the following command:

```
[master@server ~]$ sed -i.bak 's/^name:\s*.*$/name: hiromi/' contacts.txt
```

Use the **cat** command to view the contents of the modified and backup files:

```
[master@server ~]$ cat  contacts.txt  contacts.txt.bak
```

Let's see **sed**'s delete command in action. We'll use the **seq** command as the input source to print a sequence of numbers from 1 to 3 and then use **sed**'s delete (**d**) command to delete the second line (**2**) of the output. Type the following command:

```
[master@server ~]$ seq  3  | sed  2d
1
3
```

Miscellaneous Tools

The following tools don't fall into any specific category covered in this chapter, but they are often used for daily system administration chores.

Disk Utilization: du

You will often need to determine where disk space is being consumed, and by whom, especially when you're running low on it! The **du** command allows you to determine the disk utilization on a directory-by-directory basis.

Following are some of the options available:

Option	Description
-c	Produces a grand total at the end of the run
-h	Prints sizes in human-readable format
-k	Prints sizes in kilobytes rather than block sizes
-s	Summarizes; prints only a total for each argument

To display the total amount of space being used by all the files and directories in your PWD in human-readable format, use this command:

```
[master@server ~]$  du -sh  .
26.2M
```

NOTE You can use the pipe feature of the shell, discussed earlier in the chapter, to combine the **du** command with some other utilities (such as **sort** and **head**) to gather some interesting statistics about the system.

The **sort** command is used for sorting lines of text in alphanumeric, numeric order, and the **head** command is used for printing or displaying any specified number of lines of text to the standard output (screen).

So, for example, to combine **du**, **sort**, and **head** together to list the 12 largest files and directories taking up space, under the **/home/master** directory, you could run this:

```
$ du -a  /home/master  | sort -n -r | head -n 12
```


Disk Free: df

The **df** program displays the amount of free space available on mounted file systems. The drives/partitions/volumes/network shares must be mounted in order to get this information. Some parameters for **df** are listed in the following table; additional options are listed in the **df** manual page.

Option	Description
-h	Generates free-space amount in human-readable numbers rather than free blocks
-l	Lists only the locally mounted file systems; does not display any information about network-mounted file systems

To show the free space for all locally mounted drives, use this command:

```
[master@server ~]$  df  -l
```

To show the free space in a human-readable format for the file system on which **/tmp** is located, type this command:

```
[master@server ~]$  df  -h  /tmp
```

List Processes: ps

The **ps** command lists all the processes in a system, as well as their state, size, name, owner, CPU time, wall clock time, and much more. Many command-line parameters are available; those most often used are described in Table 4-4.

The most common set of parameters used with the **ps** command is **auxww**. These parameters show all the processes (regardless of whether they have a controlling terminal), each process's owners, and all the processes' command-line parameters.

Option	Description
`-a`	Shows all processes with a controlling terminal, not just the current user's processes
`-r`	Shows only running processes
`-x`	Shows processes that do not have a controlling terminal
`-u`	Shows the process owners
`-f`	Displays parent/child relationships among processes
`-l`	Produces a list in long format
`-w`	Shows a process's command-line parameters (up to half a line)
`-ww`	Shows a process's command-line parameters (unlimited width fashion)

Table 4-4 Common ps Options

Let's examine some sample output of an invocation of **`ps auxww`**:

```
[master@server ~]$  ps auxww  |  head
USER     PID %CPU %MEM     VSZ    RSS TTY    STAT START   TIME COMMAND
root       2  0.0  0.0       0      0 ?        S    Oct13   0:00 [kthreadd]
root       3  0.0  0.0       0      0 ?        S    Oct13   0:01 [ksoftirqd/0]
<........OUTPUT TRUNCATED........>
root      11  0.0  0.0       0      0 ?        S    Oct13   0:02 [rcuos/3]
```

The first line of the output provides column headers for the listing. The column headers are described in Table 4-5.

ps Column	Description
`USER`	The owner of the process.
`PID`	Process identification number.
`%CPU`	Percentage of the CPU taken up by a process. Note: For a system with multiple processors, this column will add up to more than 100 percent.
`%MEM`	Percentage of memory taken up by a process.
`VSZ`	Amount of virtual memory a process is taking.
`RSS`	Amount of actual (resident) memory a process is taking.
`TTY`	Controlling terminal for a process. A question mark in this column means the process is no longer connected to a controlling terminal.
`STAT`	State of the process. Possible states include Sleeping (S), Zombie (Z), Active/Running (R), Traced (T), and High Priority (<).
`START`	Date the process was started.
`TIME`	Amount of time the process has spent on the CPU.
`COMMAND`	Name of the process and its command-line parameters.

Table 4-5 ps Header Description

Show an Interactive List of Processes: top

The **top** command is an interactive version of **ps**. Instead of giving a static view of what is going on, **top** refreshes the screen with a list of processes every 2–3 seconds (user-adjustable). From this list, you can reprioritize processes or **kill** them. Figure 4-1 shows a **top** screen.

The **top** program's main disadvantage is that it's a CPU hog. On a congested and under-resourced system, multiple running instances of this program might not be very helpful. This can happen, for example, when multiple users start running **top** to see what's going on, only to find several other people running the program as well, slowing down the overall system even more!

Send a Signal to a Process: kill

This program's name is a little misleading: It doesn't really kill processes. What it does is send signals to running processes. The operating system, by default, supplies each process with a standard set of *signal handlers* to deal with incoming signals. From a system administrator's standpoint, the most common handlers are for signal numbers **1**, **9**, and **15**, which translate to the hang-up process, kill process, and terminate process, respectively.

When **kill** is invoked, it requires at least one parameter: the process identification number (PID) as derived from the **ps** command.

Signals

An optional parameter available for **kill** is **-*n***, where the ***n*** represents a signal number. If you don't specify the **-*n*** option, signal **15** will be sent by default. We'll discuss signals **1**, **9**, and **15** here.

```
top - 04:52:33 up 13:04,  2 users,  load average: 0.09, 0.06, 0.06
Tasks: 109 total,   2 running, 107 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.1 us,  0.2 sy,  0.0 ni, 99.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:   2049816 total,   256480 used,  1793336 free,    24008 buffers
KiB Swap:  2097148 total,        0 used,  2097148 free.    88636 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
    7 root      20   0       0      0      0 R   0.3  0.0   0:08.70 rcu_sched
 2195 master    20   0  121776   2960   2560 R   0.3  0.1   0:00.11 top
    1 root      20   0   40836   9208   3924 S   0.0  0.4   0:04.84 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.10 kthreadd
    3 root      20   0       0      0      0 S   0.0  0.0   0:01.75 ksoftirqd/0
    5 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:0H
    8 root      20   0       0      0      0 S   0.0  0.0   0:01.96 rcuos/0
    9 root      20   0       0      0      0 S   0.0  0.0   0:01.49 rcuos/1
   10 root      20   0       0      0      0 S   0.0  0.0   0:07.35 rcuos/2
   11 root      20   0       0      0      0 S   0.0  0.0   0:02.65 rcuos/3
   12 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu_bh
   13 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcuob/0
   14 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcuob/1
   15 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcuob/2
   16 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcuob/3
```

Figure 4-1 top output

The hang-up signal, **1** (**SIGHUP** or **HUP**), is very handy; for example, it can be used to tell certain server applications to go and reread their configuration files.

The kill signal, **9** (**SIGKILL**), is the impolite way of stopping a process. Rather than asking a process to stop, the operating system simply kills the process. The only time this will fail is when the process is in the middle of a system call (such as a request to open a file), in which case the process will die once it returns from the system call.

The kill signal, **15** (**SIGTERM**), can be used to request a process to gracefully terminate itself. When passed only the PID, **kill** sends signal **15** (**SIGTERM**) by default. Some programs intercept this signal and perform a number of actions so that they can shut down cleanly. Others just stop running in their tracks. Either way, **SIGTERM** isn't a guaranteed method for making a process stop.

Security Issues

The ability to terminate a process is obviously a powerful one, thereby making security precautions important. Users may kill only processes they have permission to kill. If non-root users attempt to send signals to processes other than their own, error messages are returned. The root user is the exception to this limitation; root may send signals to all processes in the system. Of course, this means root needs to exercise great care when using the **kill** command.

Examples Using the kill Command

The following examples are arbitrary; the PIDs used are completely fictitious and will be different or nonexistent on your system.

Recall that whenever a signal to send is not specified, **SIGTERM** (signal **15**) will be sent by default. Use this command to terminate (signal **15**) a process with PID number 205989:

```
[root@server ~]#  kill  205989
```

For an almost guaranteed kill of process number 593999, issue this command:

```
[root@server ~]#  kill  -9  593999
```

Type the following to send the HUP signal to process number 593888:

```
[root@server ~]#  kill  -SIGHUP   593888
```

This command does the same thing:

```
[root@server ~]#  kill  -1  593888
```

TIP To get a listing of all the possible signals available, along with their numeric equivalents, issue the **kill -l** command.

Show System Information: uname

The **uname** program produces some system details that can be helpful in several situations. Perhaps you've managed to log into a dozen different computers remotely and have lost track of where you are!

To get the operating system's name, release, system hostname, and kernel release name/version, enter the following command:

```
[master@server ~]$  uname -a
```

TIP Another command that offers distribution-specific information is the **lsb_release** command. Specifically, it can show Linux Standard Base (LSB) related information, such as the distribution name, distribution code name, release or version information, and so forth. A common option used with the **lsb_release** command is **-a**. For example:

```
$  lsb_release   -a
```

Who Is Logged In: who

On multiuser systems that have many user accounts that can be simultaneously logged in locally or remotely, the system administrator may need to know who is logged on.

A report showing all logged-on users as well as other useful statistics can be generated by using the **who** command:

```
[master@server ~]$  who
master    pts/0        2022-02-07 10:10 (192.168.2.12)
master    pts/1        2022-02-07 10:35 (10.1.122.1)
```

A Variation on who: w

The **w** command displays the same information that **who** displays, plus a whole lot more. The details of the report include who is logged in, what their terminal is, from where they are logged in, how long they've been logged in, how long they've been idle, and their CPU utilization. The top of the report also gives you the same output as the **uptime** command.

```
[master@server ~]$  w
19:38:10 up  9:31,  3 users,  load average: 0.08, 0.08, 0.09
USER     TTY      FROM             LOGIN@  IDLE   JCPU   PCPU WHAT
master     pts/0   192.168.2.12    10:10    13:48  0.45s  0.45s -bash
master    pts/1    10.1.122.1      10:35    0.00s  0.74s  0.01s w
```

Switch User: su

The **su** (switch user or substitute user) command is used for running commands as a different user. Once you have logged into the system as one user, you need not log out and back in again in order to assume another identity (root user, for instance). Instead, use the **su** command to switch. This command has few command-line parameters.

Running **su** without any parameters will automatically try to make you the root user. You'll be prompted for the root password, and, if you enter it correctly, you will drop down to a root shell. If you are already the root user and want to switch to another ID, you don't need to enter a password when you use this command.

For example, if you're logged in as the user master and want to switch to the root user, type this command:

```
[master@server ~]$  su
```

You will be prompted for root's password.

If you're logged in as root and want to switch to, say, user master, enter this command:

```
[root@server ~]$  su  master
```

You will not be prompted for master's password.

The optional hyphen (**-**) parameter tells **su** to switch identities and run the login scripts for that user. For example, if you're logged in as root and want to switch over to user master with all of its login and shell configurations, type this command:

```
[root@server ~]#  su  -  master
```

TIP The **sudo** command is used extensively (instead of **su**) on modern GNU/Linux distros to execute commands as another user and to temporarily elevate the privilege level of a regular user when necessary. When configured properly, **sudo** offers finer grained controls than **su** does. During the installation of our sample Fedora server (Chapter 2), we elected to make the **master** user an administrator. The implication of this on Red Hat like distros such as Fedora, RHEL, and CentOS is that the user was also automatically added to the special **wheel** group and will thus have **sudo** privileges. The equivalent group on Debian like distros such as Ubuntu is the aptly named **sudo** group!

Putting It All Together (Moving a User and Its Home Directory)

This section demonstrates how to put together some of the topics and CLI utilities covered so far in this chapter (as well as some new ones covered in more detail in Chapter 5). You will see how the elegant design of GNU/Linux allows you to combine simple commands to perform advanced system administration operations. The example operation will be to move a user and the user's files around on the system.

Specifically, in the following exercises, you are going to create and then move the user named project4 from his default home directory **/home/project4** to **/export/home/project4**. You will also have to set the proper permissions and ownership of the user's files and directories so that the user can access them.

Unlike the previous exercises, which were performed as a regular user (the user master), you will need superuser privileges to perform the steps in this exercise. Use the **su** command to change your identity temporarily from the current logged-in user to the superuser (root). You will need to provide root's password, when prompted.

At the virtual terminal prompt, type:

```
[master@server ~]$  su  -
```

Create the user to be used for this project. The username is project4. Type the following:

```
[root@server ~]#   useradd  project4
```

Use the **grep** command to view the entry for the new user in the **/etc/passwd** file:

```
[root@server ~]#   grep   project4  /etc/passwd
project4:x:1001:1001::/home/project4:/bin/bash
```

Use the **ls** command to display a listing of the user's home directory:

```
[root@server ~]#   ls   -al  /home/project4
...<OUTPUT TRUNCATED>...
-rw-r--r--. 1 project4 project4  193 Aug  4 02:09 .bash_profile
-rw-r--r--. 1 project4 project4  231 Aug  4 02:09 .bashrc
```

Check the total disk space being used by the user:

```
[root@server ~]#   du  -sh  /home/project4
16K    /home/project4
```

Use the **su** command to change your identity temporarily from the root user to the newly created project4 user:

```
[root@server ~]#   su  -  project4
[project4@server ~]$
```

As user project4, view your present working directory:

```
[project4@server ~]$  pwd
/home/project4
```

As user project4, create some empty files:

```
[project4@server ~]$  touch   a   b   c   d   e
```

Go back to being the root user by exiting out of project4's profile:

```
[project4@server ~]$   exit
```

Create the **/export** directory that will house the user's new home:

```
[root@server ~]#  mkdir  -p  /export
```

Now use the **tar** command to archive and compress project4's current home directory (**/home/project4**) and untar and decompress it into its new location:

```
[root@server ~]#   tar  czf  -  /home/project4 |  (cd  /export ; tar -xvzf -)
```

TIP The dashes (**-**) you used here with the **tar** command forces it to send its output to standard output (**stdout**) first and then receive its input from standard input (**stdin**).

Use the **ls** command to ensure that the new home directory was properly created under the **/export** directory:

```
[root@server ~]#   ls   -R  /export/home/
/export/home/:
project4
/export/home/project4:
a  b  c  d  e
```

Make sure that the project4 user account has complete ownership of all the files and directories in his new home:

```
[root@server ~]#  chown  -R   project4.project4  /export/home/project4/
```

Now delete project4's current home directory:

```
[root@server ~]#  rm  -rf  /home/project4
```

We are almost done. Try to temporarily assume the identity of project4 again:

```
[root@server ~]#  su  -  project4
su: warning: cannot change directory to /home/project4: No such file or
directory
-bash-4.3$
```

There's one more thing left to do. We have deleted the user's home directory (**/home/project4**). The path to the user's home directory is specified in the **/etc/passwd** file (see Chapter 6), and since we already deleted that directory, the **su** command helpfully complained.

Exit out of project4's profile using the **exit** command:

```
-bash-4.3$  exit
```

Now we'll use the **usermod** command to update the **/etc/passwd** file automatically with the user's new home directory:

```
[root@server ~]#   usermod   -d   /export/home/project4   project4
```

Use the **su** command again to become project4 temporarily:

```
[root@server ~]#   su   -   project4
[project4@server ~]$
```

While logged in as project4, use the **pwd** command to view your present working directory:

```
[project4@server ~]$   pwd
/export/home/project4
```

The output shows that our migration worked out well.

Exit out of project4's profile to become the root user, and then delete the user called project4 from the system:

```
[project4@server ~]$  exit
logout
[root@server ~]#   userdel  -r  project4
```

That's it—we are done!

Summary

This chapter discussed Linux's command-line interface, the Bourne Again Shell (Bash), many command-line tools, and a few editors. As you continue through this book, you'll find many references to the information in this chapter, so be sure that you get comfortable with working at the CLI. You might find it a bit annoying at first, especially if you are accustomed to using a GUI for performing many of the basic tasks mentioned here—but stick with it. You might even find yourself eventually working faster at the command line than with the GUI!

Obviously, this chapter can't cover all the command-line tools available as part of your default Linux installation. It is highly recommend that you take some time to look into some of the reference books available. In addition, there is a wealth of texts on shell scripting/programming at various levels and from various points of view. Get whatever suits you; shell scripting/programming is a skill well worth learning, even if you don't do system administration.

And above all else, R.T.F.M.—that is, Read The *Fine* Manual (documentation).

This page intentionally left blank

CHAPTER

5 Managing Software

A good chunk of your time as a system administrator will be spent managing the various software components (or applications) on the system, whether it's in keeping them up to date, maintaining what you already have installed, searching for software, or installing new software.

Of the many approaches to managing software on a Linux system, the chosen approach often boils down to the Linux distribution, the administrator's skill level, and other philosophical (nontechnical) considerations. From a purely technical perspective, software management under the mainstream Linux distros can be done via the following methods:

- **RPM** The Red Hat Package Manager is the common method for Red Hat–like systems such as Fedora, Red Hat Enterprise Linux (RHEL), OpenSUSE, Amazon Linux, and CentOS. DNF, Yum, and zypper are popular front-ends for managing RPM packages on RPM-based Linux distributions.
- **DPMS** The Debian Package Management System is the basis for software management on Debian-based systems, such as Ubuntu, Debian, Raspbian, Kubuntu, and Mint.
- **Source code** The more traditional approach for the Linux die-hards and purists involves compiling and installing the software by hand using the standard GNU compilation method or the specific software directives.
- **Bundles/miscellaneous** This is a relatively newer (and not well-defined) approach to thinking about, distributing, and managing software. There is no single emerging standard or implementation of this approach, but the general goal is the same—to produce a platform-agnostic and portable bundle of an application that can be easily installed and executed anywhere and everywhere. In an ideal world, a given software bundle should be able to run the same way across all Linux distros, all the *BSDs, macOS, and even Windows systems. Sample implementations of this include Snap, Flatpak, AppImage, and Pkcon.

The Red Hat Package Manager

RPM is a software management system that is used for the installation and removal of software packages—typically, precompiled software. An RPM file is a package that contains files needed for the software to function correctly. A package consists of an archive of files and other metadata, including configuration files, binaries, and even pre- and postscripts to run while installing the software. RPM is wonderfully easy to use, and several graphical interfaces have been built around it to make it even easier. Several Linux distros and various third parties use this tool to distribute and package their software. In fact, almost all of the software mentioned in this book is available in RPM form.

NOTE In the present context, we are assuming that the RPM files contain precompiled binaries. However, this is not always the case!

In keeping with the true spirit of open source, most FOSS developers and vendors also make the source code for software available in RPM form. These are called source RPMs (src.rpm).

The RPM tool performs the installation and uninstallation of RPMs. The tool also maintains a central database of what RPMs you have installed, where they are installed, when they were installed, and other information about the package.

In general, software that comes in the form of an RPM is less work to install and maintain than software that needs to be compiled. The trade-off is that by using an RPM, you are implicitly accepting the default parameters supplied by the RPM maintainer. In most cases, these defaults are acceptable. However, if you need to be more intimately aware of what is going on with a piece of software, or you require functionality that is unusual or different from what is available in the RPM, you might find that by compiling the source yourself, you learn more about the hidden software features and other tunable options.

Assuming that all you want to do is to install a simple package, RPM is perfect. You can find several great resources for RPM packages, beyond the base distribution repositories, at web sites such as the following: http://mirrors.kernel.org, http://rpm.pbone.net, and http://rpmfusion.org.

RPM comes with Fedora, CentOS, openSUSE, and countless other Red Hat derivatives, including, most shocking of all, the Red Hat Enterprise version of Linux.

NOTE Although the acronym RPM happens to have "Red Hat" in it's name, the software packaging format can be used on other platforms as well. In fact, RPM has even been ported to other operating systems, such as Solaris, AIX, Debian, and IRIX. The source code to RPM is open source software, so anyone can take the initiative to make the system work for them. Learn more about RPM at www.rpm.org.

Following are the primary functions of the RPM program:

- Querying, verifying, updating, installing, and uninstalling software
- Maintaining a database that stores various items of information about the packages
- Packaging other software into an RPM form

Getting Down to Business

In Chapters 2 and 3, we walked through setting up the operating system. Now that you have a working system, you will need to log into the system to carry out the exercises in this and other chapters of the book. Most of the exercises will implicitly ask you to type a command. Although it might seem obvious, whenever you are asked to type a command, you will have to type it into a console at the shell prompt. This is akin to the command, CMD, or PowerShell prompt in Microsoft Windows but is much more powerful and intuitive.

You can type a command at the shell in several ways. One way is to use a nice, windowed (GUI) terminal; another is to use the system console. The windowed consoles are known as "terminal emulators" (or "pseudo-terminals"), and there are tons of them.

The base Fedora Server operating system that you installed in Chapters 2 and 3 may or may not have given you an option to install a desktop GUI environment, and you would have been thrown rudely into a black login console after the system rebooted. The exercises in Chapter 4 also assumed that you were following along at the text-only terminal console. In this software management chapter, you will now have the option to install a GUI desktop environment of your choice.

So without further ado, your first order of business may be to install your desktop environment of choice. Some of the options are GNOME, KDE, XFCE, Mate, LXDE, and so on.

In the Fedora distro world, GNOME is the default desktop environment, and it just so happens to be the default shipped with the Fedora Workstation product spin. So assuming you installed the Fedora Server flavor, as demonstrated in Chapter 2, and you now want to install and use the GNOME desktop environment, type the following:

```
$ sudo  dnf  -y --allowerasing  groupinstall "Fedora Workstation"
```

If you are running the rawhide stream of Fedora you might have to add the **`--nogpgcheck`** option to the previous command if you get weird errors about gpg signatures.

Once the installation is completed successfully, you'll have to set this graphical environment as the new target or runlevel for this system to boot into by typing the following:

```
$ sudo  systemctl set-default graphical.target
```

Finally, reboot the system to boot into and log into the graphical desktop environment:

```
$ sudo reboot
```

(*continued*)

Once you have installed a full-blown graphical desktop environment and you are logged into the system via such an environment, you can still follow along with all the exercises in this book by entering the commands in the GUI terminal application (or pseudo-terminal).

In most environments, you can usually launch a pseudo-terminal by right-clicking the desktop and selecting Launch Terminal from the context menu. If you don't have that particular option, look for an option in the applications menu that says Run Command (or press ALT-F2 to launch the Run Application dialog box). After the Run dialog box appears, type the name of a terminal emulator into the Run text box. A popular terminal emulator that is almost guaranteed to exist on all Linux systems is the venerable **xterm**. If you are in a GNOME desktop, the **gnome-terminal** is the default. If you are using KDE, the default is **konsole**. Just type **terminal** in the search box and you'll be presented with whatever is available.

NOTE Installing and uninstalling software on a system is considered an administrative or privileged function. This is why you will notice that most of the commands in the following sections are performed with elevated privileges. The method of achieving or implementing this privileged elevation status varies by distribution, but the ideas and tools used are common across almost all the distros.

On the other hand, querying the software database is not considered a privileged function.

Managing Software Using RPM

The following sections cover the details of querying, installing, uninstalling, and verifying software on Red Hat–type Linux distributions such as Fedora, RHEL, CentOS, and openSUSE.

Querying for Information the RPM Way (Getting to Know One Another)

One of the best ways to begin any relationship is by getting to know the other person. Some of the relevant information could include the person's name, what they do for a living, their birthday, and their likes and dislikes. The same rules apply to RPM packages. In a similar way, after you obtain a piece of software, you should get to know the software before making it a part of your system.

As you continue working with and exploring Linux, you will find that software names are somewhat intuitive, and you can usually tell what a package is and does just by looking at its name. For example, to the uninitiated, it might not be immediately obvious that a file named **gcc-10.1.1.rpm** is a package for the GNU Compiler Collection (GCC). But once you get used to the nuances and you know what to look for, these types of things will become more intuitive. You can use RPM to query for various types of information, such as the package's build date, its weight (or its size), its likes and dislikes (or its dependencies), and so on.

Let's start working with RPM. Begin by logging into the system as a user with administrative privileges, such as **root** or the **master** user with superuser or sudo privileges. Launch a terminal if you are logged in via the GUI.

Querying for All Packages

Use the **rpm** command to list all the packages that are currently installed on your system. At the shell prompt, type:

```
[master@fedora-server ~]$  rpm --query --all
```

NOTE Like most Linux commands, the **rpm** command has its own long and short (or abbreviated) forms of options and arguments. For example, the short form of the **--query** option is **-q**, and the short form of **--all** is **-a**. We will interchange the use of the short and long forms of command options in this book.

Querying Details for a Specific Package

Let's zero in on one of the packages listed in the output of the preceding command: the Bash application. Use **rpm** to see if you indeed have the Bash application installed on your system:

```
[master@fedora-server ~]$ rpm --query bash-*
bash-5.*.x86_64
```

The output should be something similar to the second line, which shows that the Bash package is installed. It also shows the version number appended to the package name.

NOTE When you're dealing with software packages in Linux distros, the exact software version number on your system might be different from the version number on our sample system. Factors such as updating the system and Linux distro version itself affect and determine exact package versions.

This is one of the reasons why we might sometimes truncate the version number in package or software names in some exercises. For example, instead of writing **bash-9.8.4.2.rpm** we might just write **bash-9.***.

One thing you can be assured of is that the main package name will almost always be the same—that is, Bash is Bash is Bash in openSUSE, Fedora, Mandrake, CentOS, RHEL, Ubuntu, and so on.

This brings us to the next question. What is Bash and what does it do? To find out, type the following:

```
[master@fedora-server ~]$  rpm  -qi bash
Name        : bash
Version     : 5.*
Group       : System Environment/Shells
....<OUTPUT TRUNCATED>....
Vendor      : Fedora Project
URL         : http://www.gnu.org/software/bash
Summary     : The GNU Bourne Again shell
Description :
The GNU Bourne Again shell (Bash) is a shell or command language interpreter
that is compatible with the Bourne shell (sh)...
```

This output gives us a lot of information. It shows the version number, the release, the description, the packager, and more.

The Bash package looks rather impressive. Let's see what else comes with it. This command lists all the files that come along with the Bash package:

```
[master@fedora-server ~]$ rpm -ql bash
```

To list the configuration files (if any) that come with the Bash package, type this:

```
[master@fedora-server ~]$  rpm -qc bash
/etc/skel/.bash_logout
/etc/skel/.bash_profile
/etc/skel/.bashrc
```

The querying capabilities of **rpm** are extensive. RPM packages have a lot of information stored in *tags,* which make up the metadata of the package. You can query the RPM database for specific information using these tags. For example, to find out the date that the Bash package was installed on your system, you can type the following command:

```
[master@fedora-server ~]$ rpm -q --qf "[ %{INSTALLTIME:date} \n]" bash
Thurs 1 Dec 2022 22:40 PM EDT
```

TIP Because Bash is a standard part of most Linux distros and would have been installed automatically when you initially installed the OS, you will find that its install date will be close to the day you installed the OS!

You can query for more than one package at the same time and also query for multiple pieces of tag information. For example, to display the names and package groups for the Bash and nano packages, type this:

```
[master@fedora-server ~]$ rpm -q --qf "[%{NAME} - %{GROUP} - %{SUMMARY} \n]" \
bash nano
bash - System Environment/Shells - The GNU Bourne Again shell
nano - Applications/Editors - A small text editor
```

To determine what other packages on the system depend on the Bash package, type the following:

```
[master@fedora-server ~]$ rpm -q --whatrequires bash
```

TIP The RPM queries noted here were issued on software that is currently installed on the system. You can perform similar queries on package files obtained from various sources that have not yet been installed. To do this, you simply add the **-p** option to the end of the query command. For example, say you've just downloaded a package named **joe-9.1.6.noarch.rpm** into your current working directory and you want to query the uninstalled package to get more information about it. You would type this:

```
$ rpm -qip  joe-9.1.6.noarch.rpm
```

Installing Software with RPM (Moving in Together)

Okay, you are now both ready to take the relationship to the next stage. You have decided to move in together. This can be a good thing, because it allows both of you to see and test how truly compatible you are. This stage of relationships is akin to installing the software package on your system—that is, moving the software into your system.

In the following procedures, you will install a simple text-based web browser application called **lynx** onto your system. First, you will need to get a copy of the RPM package for **lynx**. You can get this program from several places: the Internet, the install medium, and so on.

In the following sections we'll cheat a little and use the **dnf** package manager to download copies of programs that we need. Downloading packages is just one of the numerous things **dnf** is designed to do.

Alternatively, you can browse for and download any needed files directly from the vendor web site:

http://dl.fedoraproject.org/pub/fedora/linux/development/rawhide/Everything/x86_64/os/Packages/

Let's step through the process of obtaining and installing an RPM package:

1. Log into the system as a user with administrative rights. We use the **master** account on the sample system but you can also use the **root** account. If you are logged in via the GUI, complete the following steps via a virtual terminal.
2. Use the **mkdir** and **cd** commands to first create a **Downloads** folder (just in case it doesn't already exist) and then change your current working directory to the **Downloads** folder in your home directory:

```
[master@fedora-server ~]$ mkdir ~/Downloads  &&  cd  ~/Downloads
```

3. Download the compressed file used for this chapter using the **dnf** program:

```
[master@fedora-server Downloads]$ dnf download lynx
```

4. Double-check to make sure the package was correctly downloaded. Use the **ls** command to list all the files that start with the letters *lyn* in the destination directory.

```
[master@fedora-server  Downloads]$ ls  lyn*
lynx-*.rpm
```

5. Now that you have confirmed that the file is there, perform a test install of the package. This will run through the motions of installing the package without actually installing anything on the system. This is useful in making sure that all the needs (dependencies) of a package are met. Type the following:

```
[master@fedora-server Downloads]$ rpm --install --verbose --hash  --test lynx*
Verifying...          ############### [100%]
Preparing...          ############### [100%]
```

Everything looks okay. If you get a warning message about the signature, you can safely ignore it for now.

6. Now perform the actual installation:

```
[master@fedora-server Downloads]$ sudo rpm -ivh lynx-*
Verifying...                          ################ [100%]
Preparing...                          ################ [100%]
Updating / installing...
   1:lynx-*                ###################### [100%]
```

7. Run a simple query to confirm that the application is installed on your system:

```
[master@fedora-server Downloads]$ rpm -q lynx
lynx-*.x86_64
```

The output shows that **lynx** is now available on the system.

In case you were wondering, **lynx** is a text-based web browser. You can launch it by simply typing **lynx** at the shell prompt. To quit **lynx**, simply press Q. You will get a prompt at the lower-right corner of your terminal to confirm that you want to quit lynx. Press ENTER to confirm.

As you can see, installing packages via RPM can be easy. But sometimes installing packages directly can be trickier, usually due to failed or missing dependencies. For example, the **lynx** package might require that the **bash** package be installed on the system before **lynx** can be successfully installed.

TIP You can easily make accessible the contents of the operating system image that was used in Chapter 2 by mounting the ISO file. For example, to mount the Fedora DVD image named **Fedora-Server-dvd-x86_64*.iso** at an existing **/media/iso** directory, you can type the following:

```
# mount Fedora-Server-dvd-x86_64*.iso  /media/iso/
```

The pain that we are going to put you through in the following sections while trying to manually install the **gcc** package is somewhat necessary, because it offers a glimpse into how things work in the background, and, more importantly, it'll let you have better appreciation for the easier methods (dnf, yum, zypper, and so on) we'll discuss later on.

Let's walk through installing a more complex package to see how dependencies are handled with RPM. We'll start by first downloading the package we want to install.

1. This housekeeping step is strictly not required, but is highly recommended especially if you are running a bleeding-edge rawhide stream of fedora, like we are on our sample server.

   Let's bring all currently installed packages up to date. We'll use the update option with the **dnf** program like so:

   ```
   [master@localhost Downloads]$ sudo dnf -y update --nogpgcheck
   ```

   This process might take a while—depending on how many new updates are available.

2. Now we'll get on with the original objective. First, download the latest available version of **gcc** into the **Downloads** folder:

   ```
   [master@fedora-server Downloads]$ dnf download gcc
   ```

3. Try installing the downloaded **gcc** package by typing the following:

```
[master@fedora-server Downloads]$ sudo  rpm -ivh gcc-*.rpm
error: Failed dependencies:
     binutils >= * is needed by gcc-*.x86_64
     cpp = * is needed by gcc-*.x86_64
     glibc-devel >= * is needed by gcc-*.x86_64
       libisl.so.*()(64bit) is needed by gcc-*.x86_64
       libmpc.so.*()(64bit) is needed by gcc-*.x86_64
```

   The output does *not* look good. It tells us that **gcc-.*** (and it's dependencies) depends on (needs) some other packages and libraries: **binutils**, **cpp**, **glibc-devel**, **libisl.so.*** and **libmpc.so***.

4. One quick way to figure out the names of packages and determine *what* provides *what*, is to use the **whatprovides** option with the dnf command. That's how to unravel the sometimes cryptic names of libraries and packages. For example, to determine which package provides the missing **libmpc.so.*** library shown in the previous output, type:

```
[master@fedora-server Downloads]$ dnf   whatprovides  libmpc.so.*
libmpc-* : C library for multiple precision complex arithmetic
Provide    : libmpc.so.3
```

   From the output, we can tell the package named **libmpc** provides the needed library. You'll need to do this for any of the dependencies whose package names aren't clear.

5. Let's go ahead and use **dnf** to download these dependencies:

```
[master@fedora-server Downloads]$ sudo dnf download --arch=x86_64 \
binutils cpp glibc-devel isl libmpc
```

6. We've gathered the requisite packages needed so far into the **Downloads** folder. Let's add new packages to the transaction list and try again:

```
[master@fedora-server Downloads]$ sudo rpm -ivh \
gcc* binutils* cpp* glibc-devel*  isl* libmpc*
error: Failed dependencies:
binutils-gold >= * is needed by binutils-*.x86_64
    glibc = * is needed by glibc-devel-*.x86_64
    glibc-headers is needed by glibc-devel-*.x86_64
    glibc-headers = * is needed by glibc-devel-*.x86_64
    libxcrypt-devel >= *  is needed by glibc-devel-*.x86_64
```

   Uh-oh. It looks like this particular partner is not going to be easy to move in with. The output tells us that there are still a few dependencies missing such as binutils-gold*, glibc-headers*, libxcrypt-devel, and so on.

TIP The act of performing multiple RPM package installations, upgrades, removals in one shot is called an *RPM transaction*, which is what we did for the **gcc** package installation example.

7. We can continue down this manual path of trying to figure out all the dependencies individually or we can save some time and trick/coax **dnf** into doing most of the tedious work for us. This can be done by passing the **install** and **downloadonly** option to **dnf**. For our current **gcc** application example, the command to do this is:

```
[master@fedora-server Downloads]$ sudo dnf -y install gcc  \
--downloadonly   --downloaddir=$HOME/Downloads
```

   Once this is done, the **Downloads** folder will have all the extra packages needed to successfully install the stubborn **gcc** application manually!

8. Let's attempt one single large **rpm** transaction to install **gcc** along with all the other downloaded packages it depends on. We'll use the **`--upgrade`** (**`-U`**) option this time just in case the downloaded dependencies are newer than their counterpart versions that are already installed. Type the following:

```
[master@fedora-server Downloads]$ sudo rpm -Uvh --replacepkgs   *.rpm
…<OUTPUT TRUNCATED>…
6:libmpc-*          ################# [ 38%]
10:glibc-headers-*  ################# [ 63%]
12:glibc-devel-*    ################# [ 75%]
13:gcc-*            ################# [ 100%]
```

9. It was tough, but you managed to get the GNU C Compiler (**gcc**) package manually installed!

TIP A popular option used in installing packages via RPM is the **`-U`** (for upgrade) option. It is especially useful when you want to install a newer version of a package that already exists. It will simply upgrade the already installed package to the newer version. This option also takes care of preserving any custom configurations that you might have for the application.

For example, if you had **lynx-9-8.rpm** installed and you wanted to upgrade to **lynx-9-9.rpm**, you would type **`rpm -Uvh lynx-9-9.rpm`**. Note that you can use the **`-U`** option to perform a regular installation of a package even when you are not upgrading.

Uninstalling Software with RPM (Ending the Relationship)

Things didn't quite work out the way you both had anticipated. Now it is time to end the relationship. The other partner was never any good anyhow, so you simply need to clean them out of your system.

Cleaning up after itself is one of the areas in which RPM truly excels. And this is one of its key selling points as a software manager. RPM is able to do this so well because it maintains a local database of sorts that stores copious information about what was installed, where, and when.

NOTE A slight caveat applies here. As with Windows install/uninstall tools, all the wonderful things that RPM can do are also dependent on the software packager or vendor. For example, if a software application was badly packaged and its removal scripts were not properly formatted, you might still end up with bits of the package on your system, even after uninstalling. This is one of the reasons why you should always get software only from trusted sources.

Removing software with RPM is quite easy and can be done in a single step. For example, to remove the **lynx** package that we installed earlier, we simply need to use the **-e** option, like so:

```
[master@fedora-server ~]$ sudo rpm -e lynx
```

This command will usually not give you any feedback if everything went well. To get a more verbose output for the uninstallation process, add the **-vvv** option to the command.

A handy feature of RPM is that it will also protect you from removing packages that are needed by other packages. For example, if we try to remove the **kernel-headers** package (recall that the **gcc** package installed earlier depended on it), we'd see the following:

```
[master@fedora-server ~]$ sudo  rpm -e kernel-headers
error: Failed dependencies:
      kernel-headers >= * is needed by (installed) glibc-headers-*
```


NOTE Remember that the **glibc-headers*** package required the **kernel-headers** package, too, and so RPM will do its best in helping you maintain a stable software environment. But if you are adamant about shooting yourself in the foot, RPM will also allow you to do that (perhaps because you know what you are doing).

If, for example, you wanted to forcefully perform the uninstallation of the **kernel-headers** package, you would add the **--nodeps** option to the uninstallation command, like this:

[master@fedora-server ~]$ **sudo rpm --nodeps -e kernel-headers**

Other Things RPM Can Do

In addition to basic installation and uninstallation of packages with RPM, you can do numerous other things with it. In this section, we walk through some of these other functions.

Verifying Packages

The RPM tool can be used to verify the sanctity of packages. The verification works by RPM looking up the package information in its database, which is considered to be the source of truth. It then compares that information with the actual binaries and files that are on your system.

In today's very security conscious world, this kind of test is useful for telling you instantly if anyone has tampered with the software installed on your system. For example, to verify that the **bash** package is as it should be, type the following:

```
[master@fedora-server ~]$  rpm -V bash
```

The absence of any output is a good sign.

You can also verify specific files on the file system that a particular package installed. For example, to verify that the **/bin/ls** command is valid, you would type this:

```
[master@fedora-server ~]$  rpm -Vf  /bin/ls
```

Again, the lack of output is a good thing.

Code	Meaning
S	File size differs.
M	Mode differs (includes permissions and file type).
5	MD5 sum differs.
D	Device major/minor number mismatch.
L	readLink-path mismatch.
U	User ownership differs.
G	Group ownership differs.
T	Modification time (mtime) differs.

Table 5-1 RPM Verification Error Attributes

If something was amiss—for example, if the **/bin/ls** command had been replaced by a dud version—the verify output might be similar to this:

```
[master@fedora-server ~]$ rpm -Vf /bin/ls
SM5....T.    /usr/bin/ls
```

If something is wrong, as in this example, RPM will inform you which of its tests failed (see Table 5-1 for the meaning of some of the error codes). Some sample tests are the MD5 checksum test, file size, and modification times. The key takeaway here is that RPM can also be a valuable addition to your security box of tricks.

Table 5-1 provides a summary of the various error codes and their meanings.

You can use the following command to verify all the packages installed on your system:

```
[master@fedora-server ~]$  rpm -Va
```

This command verifies *all* the packages installed on your system—that's potentially a lot of files! Therefore, you might have to give it some time to complete.

Package Validation

Another feature of RPM is that it allows the packages to be digitally signed. This provides a type of built-in authentication mechanism that allows a user to ascertain that the package in their possession was truly packaged by the expected (trusted) party and also that the package has not been tampered with along the line somewhere.

You sometimes need to manually tell your system whose digital signature to trust. This explains the reason why you might see some warnings in the earlier procedures when you were trying to install a package (such as this message: warning: lynx-*.rpm: Header V3 RSA/SHA256 Signature, key ID 34ba*: NOKEY).

To prevent this warning message, you might need to import Fedora's official digital key into your system's key ring, like so (replace **<VERSION>** with your version of Fedora):

```
[master@fedora-server ~]$   sudo  rpm --import   \
/etc/pki/rpm-gpg/RPM-GPG-KEY-fedora-<VERSION>
```

You might also have to import other vendors' keys into the key ring.

To be extra certain that even the local key you have is not a dud, you can import the key directly from the vendor's web site. You will find a listing of Fedora's current package-signing keys at https://getfedora.org/security/. For instance, to import Fedora's GPG key(s) directly from Fedora's project site, you execute the next command:

```
[master@fedora-server ~]$ sudo rpm --import  \
https://getfedora.org/static/fedora.gpg
```

To list all signing keys imported into and trusted by **rpm**, type the following:

```
[master@fedora-server ~]$  rpm -q gpg-pubkey \
 --qf '%{NAME}-%{VERSION}-%{RELEASE}\t%{SUMMARY}\n'
```

Table 5-2 lists some frequently used **rpm** command options and is provided here for quick reference purposes only.

Command-Line Option	Description
`--install`	Installs a new package.
`--upgrade`	Upgrades or installs packages currently installed to a newer version.
`--erase`	Removes or erases an installed package.
`--query`	Used for querying or retrieving information about various attributes concerning installed (or uninstalled) packages.
`--force`	Tells RPM to forego any sanity checks and just do what you tell it, even if it thinks you're trying to fit a square peg into a round hole. Be careful with this option; it's like using a sledgehammer. Typically, you use it when you're knowingly installing an odd or unusual package version and RPM's safeguards are trying to keep you from doing so.
`-h`	Prints hash marks to indicate progress during an installation. Use with the `-v` option for a pretty display.
`--percent`	Prints the percentage completed to indicate progress or status.
`-nodeps`	Causes RPM to not perform any dependency checks if RPM is complaining about missing dependency files, but you want the installation to happen anyway.
`-q`	Queries the RPM system for information.
`--test`	Checks to see whether an installation would succeed without performing an actual installation. If it anticipates problems, it displays what they'll be.
`-V`	Verifies RPMs or files on the system.
`-v`	Tells RPM to be verbose about its actions.

Table 5-2 Common RPM Options

Yum

Yum is a popular packaging/updating tool for managing software on Linux systems. It is basically a wrapper program for RPM, with great usability enhancements. It has been around for a while and is now quite mature. Yum has changed and enhanced the traditional approach to package management on RPM-based systems. According to the Yum project's web page (http://yum.baseurl.org/):

> Yum is an automatic updater and package installer/remover for rpm systems. It automatically computes dependencies and figures out what things should occur to install packages. It makes it easier to maintain groups of machines without having to manually update each one using rpm.

This summary is an understatement. Yum can do a lot beyond that. Several Linux distributions rely heavily on the capabilities provided by Yum.

Using Yum is simple whenever it is installed on supported systems. You mostly need a single configuration file (**/etc/yum.conf**). Other configuration files can be stored under the **/etc/yum.repos.d/** directory that points to other Yum-enabled (Yumified) software repositories. Fortunately, several Linux distributions (such as Fedora, CentOS, and RHEL) ship with Yum already installed and preconfigured.

To use Yum on a supported Fedora system (or older Red Hat–like distro) to install a package called **gcc**, for example, you would type the following at the command line:

```
[master@fedora-server  ~]$ sudo yum install gcc
```

Yum will automatically take care of any dependencies that the package might need and install the package for you.

Yum also has extensive search capabilities that will help you find a package, even if you don't know its correct name. All you need to know is part of the name. For example, if you want to search for all packages that have the word "headers" in the name, you can try a Yum option like this:

```
[master@fedora-server ~]$ yum search headers
```

This will return a long list of matches. You can then look through the list and pick the package you want.

NOTE By default, Yum tries to access repositories that are located somewhere on the Internet. Therefore, to get the best out of Yum, your system needs to be on a network and have access to the Internet or some other locally managed repositories.

DNF

DNF is the new package manager slated to replace Yum on Red Hat–type distros. It is feature compatible with Yum and supports virtually all the same options that Yum supports, which makes it an easy tool to pick up for those already familiar with Yum. Users of Yum should be able to literarily replace the word **`yum`** with the word **`dnf`** for all package management needs when using the program from the command line.

DNF offers the following improvements over Yum and aims to excel in areas where Yum has issues:

- Better performance and memory footprint
- Faster and simpler package manager
- Smaller code base
- Better support for plug-ins

To use DNF on a Fedora system (or any other Red Hat–like distro) to install a package called **gcc**, for example, you would type the following at the command line:

```
[master@fedora-server  ~]$ sudo dnf install gcc
```

DNF will automatically take care of any dependencies that the package might need and install the package for you—even faster than Yum would.

Like Yum, DNF also has extensive search capabilities that will help you find a package or locate the packages that provide libraries. All you need to know is part of the name. For example, if you want to search for all packages that would satisfy (resolve) a library dependency issue for a library file named **libmpfr.so.8**, you can try a **dnf** command like this:

```
[master@fedora-server ~]$   dnf whatprovides libmpfr.so.*
```

TIP Red Hat–like distros, such as Fedora and CentOS, make use of a system called Package Groups to pool together packages that provide a specific functionality. One popular sample group is called "C Development Tools and Libraries," and it includes commonly used developer tools like **gcc**, **make**, and so on. You will need some of these tools later on in this chapter when building the **hello** program from source code. The entire package group can be installed as follows:
 # dnf groupinstall --nogpgcheck "C Development Tools and Libraries"

GUI RPM Package Managers

For those who like a good GUI tool to help simplify their lives, several package managers with GUI front-ends are available. Doing all the dirty work behind these pretty GUI front-ends on Red Hat–based distros is RPM. The GUI tools allow you to do quite a few things without forcing you to remember command-line parameters. Some of the more popular tools that ship with various distributions or desktop environments are listed in the sections that follow.

Fedora or Ubuntu

GNOME is the default desktop environment on Fedora or Ubuntu distros; as such, the design language of the GUI for both looks very similar. You can launch the GUI package management tool (see Figure 5-1) in either by pressing ALT-F2 on your keyboard, typing **gnome-software** in the Enter A Command text box, and pressing ENTER.

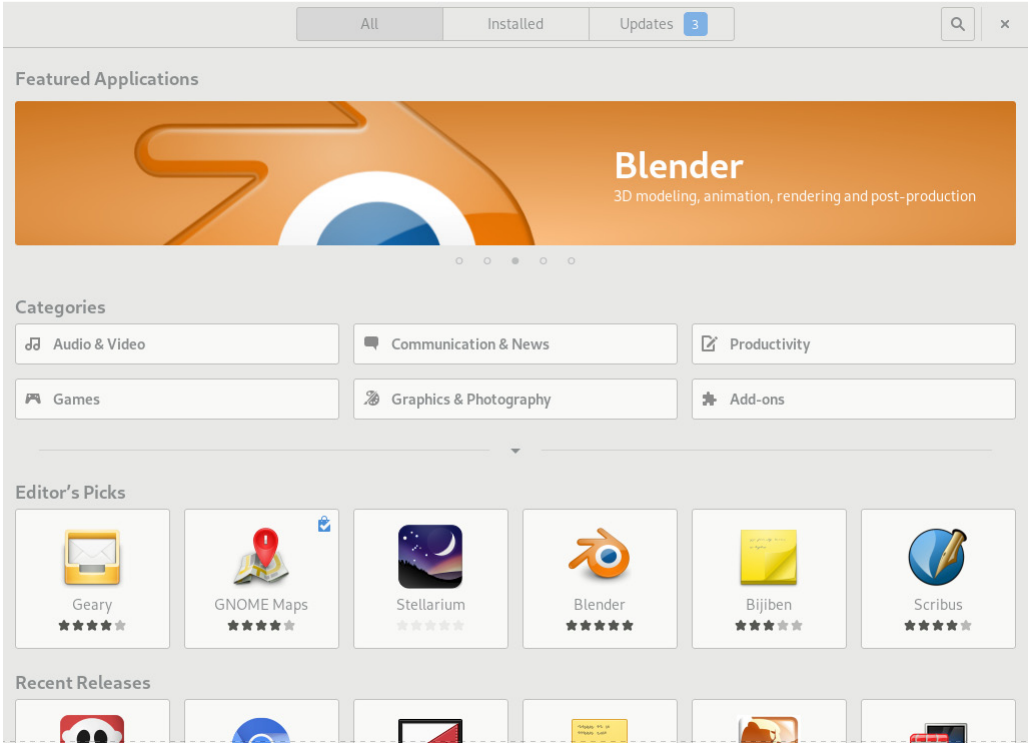


Figure 5-1 Fedora GUI package manager

You can also launch the Fedora package manager from the command line by typing the following:

```
[master@server ~]$ gnome-software
```

openSUSE and SLE

In openSUSE and SUSE Linux Enterprise (SLE), most of the system administration tasks can be done via a tool called YaST, which stands for Yet Another Setup Tool. YaST is made up of different modules. For adding and removing packages graphically on the system, the relevant module is called **sw_single**. To launch this module from the command line of a system running the SUSE distribution, you would type the following:

```
opensuse-server:~ #  yast2  sw_single
```

The Debian Package Management System

The Debian Package Management System (DPMS) is the framework for managing software on Debian and Debian-like systems. As is expected of any software management system, DPMS provides for easy installation and removal of software packages. Debian package names end with the *.deb* extension.

At the core of DPMS is the **dpkg** (Debian Package) application, which works in the back-end of the system, and several other command-line tools and GUI tools that have been written to interact with it. Packages in Debian are fondly called ".deb files" (pronounced dot deb). **dpkg** can directly manipulate .deb files. Various other wrapper tools have been developed to interact with **dpkg**, either directly or indirectly.

APT

APT is a highly regarded and sophisticated toolset. It is an example of a wrapper tool that interacts directly with **dpkg**. APT is actually a library of programming functions that is used by other middle-ground tools, such as **apt-get** and **apt-cache**, to manipulate software on Debian-like systems. Several user-land applications have been developed that rely on APT. (*User-land* refers to non-kernel programs and tools.) Examples of such applications are **synaptic**, **aptitude**, and **dselect**. The user-land tools are generally more user-friendly. APT has also been successfully ported to run on other operating systems.

One fine difference between APT and **dpkg** is that APT does not directly deal with .deb packages; instead, it manages software via the locations (repositories) specified in configuration files. The main file is **sources.list**. APT utilities use the **sources.list** file to locate archives (or repositories) of the package distribution system in use on the system.

As the system administrator, you have lots of choices for what tools to use for software management, but at the end of the day your tool of choice depends on your level of comfort and familiarity with whatever is available!

Figure 5-2 shows what can be described as the "DPMS triangle." The tool at the apex of the triangle (**dpkg**) is the most obtuse and powerful to use, followed by the next easiest to use (APT), and then the user-friendly user-land tools.

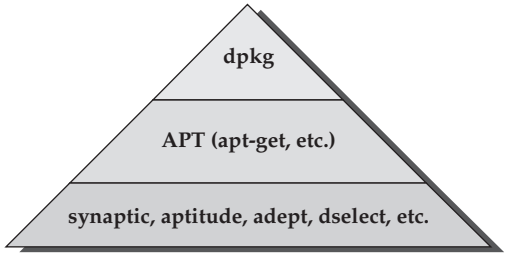


Figure 5-2 DPMS triangle

Software Management in Ubuntu

As mentioned earlier, software management in the Debian-like distros such as Ubuntu is done using DPMS and all the attendant applications built around it, such as APT and **dpkg**. In this section, we'll look at how to perform basic software management tasks on Ubuntu.

Querying for Information

On your Ubuntu server, the command to list all currently installed software is

```
master@ubuntu-server:~$  dpkg -l
```

The command to get basic information about the installed Bash package is

```
master@ubuntu-server:~$ dpkg -l bash
```

The command to get more detailed information about the Bash package is

```
master@ubuntu-server:~$ dpkg --print-avail bash
```

To view the list of files that comes with the Bash package, type

```
master@ubuntu-server:~$ dpkg-query  -L  bash
```

The querying capabilities of **dpkg** are extensive. You can use DPMS to query for specific information about a package. For example, to find out the size of the installed Bash package, you can type

```
master@ubuntu-server:~$ dpkg-query   -W  \
--showformat='${Package} ${Installed-Size} \n' bash
```

Installing Software in Ubuntu

You can install software on Ubuntu systems in several ways. You can use **dpkg** to install a .deb file directly, or you can use **apt-get** (or just plain **apt**) to install any software available in the Ubuntu repositories on the Internet or even locally. A copy of the relevant file that you can use for this section on an Ubuntu distro can be downloaded from http://linuxserverexperts.com/8e/chapter-05.tar.xz.

To follow along, using this bundle, just download the file, extract its contents, locate the file you need, and use it (see the earlier RPM example for details).

NOTE Installing and uninstalling software on a system is considered an administrative or privileged function. This is why you will notice that any commands that require superuser privileges are preceded by the **sudo** command whenever the command is being run by a regular user.

The **sudo** command can be used to execute commands in the context of a privileged user (or another user). On the other hand, querying the software database is not considered a privileged function.

To use **dpkg** to install a .deb package named **joe_*.deb**, which can be found in the directory that you downloaded and decompressed it into, type

```
master@ubuntu-server:~/chapter-04$ sudo dpkg --install  joe_*.deb
```

Command	Meaning
`update`	Retrieve new lists of packages
`upgrade`	Perform an upgrade
`install`	Install new packages
`remove`	Remove packages
`autoremove`	Remove automatically all unused packages
`purge`	Remove packages and config files
`dist-upgrade`	Distribution upgrade
`check`	Verify that there are no broken dependencies

Table 5-3 Common apt-get Options

Using **`apt-get`** to install software is a little easier, because APT will usually take care of any dependency issues for you. The only caveat is that the repositories configured in the **sources.list** file (**/etc/apt/sources.list**) have to be reachable either over the Internet or locally.

To use **`apt-get`** to automatically download and install a package called **lynx,** type

```
master@ubuntu-server:~$  sudo apt-get -y install lynx
```

Some common **`apt-get`** options are listed in Table 5-3.

TIP Debian-like distros such as Ubuntu sometimes group specific packages together that serve a common purpose into what is called a *meta-package*. One very useful meta-package is called **`build-essential`**, and it includes commonly used developer tools like **gcc**, **make**, and **g++**. You will need some of these individual applications later on in this chapter when building from source. Installing **`build-essential`** is as simple as running the following command:

```
#  apt-get -y  install  build-essential
```

Removing Software in Ubuntu

Uninstalling software (such as **lynx**) in Ubuntu using **`dpkg`** is as easy as typing the following:

```
master@ubuntu-server:~$ sudo dpkg --remove lynx
```

You can also use **`apt-get`** to remove software by using the **`remove`** option. To remove the **lynx** package using **`apt-get`**, type this:

```
master@ubuntu-server:~$ sudo apt-get remove lynx
```

A less commonly used method for uninstalling software with APT is to use the **`install`** switch and then append a minus (**–**) sign to the package name to be removed. This can be useful when you want to install and remove another package in one shot.

To remove the already installed **joe** package and simultaneously install another package called **lynx** using this method, type the following:

```
master@ubuntu-server:~$ sudo apt-get install lynx joe-
…….
Reading state information... Done
The following packages will be REMOVED:
  joe
The following NEW packages will be installed:
  lynx
<.....OUTPUT TRUNCATED....>
```

APT makes it easy to remove (or purge) software and any attendant configuration file(s) completely from a system. Assuming you want to remove the **lynx** application completely from the system, you would type this:

```
master@ubuntu-server:~$ sudo apt-get --purge remove lynx
```

GUI Package Managers for Debian-Based Systems (Ubuntu)

Several GUI and menu-driven software management tools are available on Debian-based distros such as Ubuntu. For desktop-class systems, GUI tools are installed by default.

Some of the more popular GUI tools in Ubuntu are **synaptic** and **adept**.

Ubuntu also has a couple of tools that are not exactly GUIs but offer a similar ease of use as their fat GUI counterparts. These tools are console-based or text-based and menu-driven. Examples of such tools are **`aptitude`** (see Figure 5-3) and **`dselect`**.

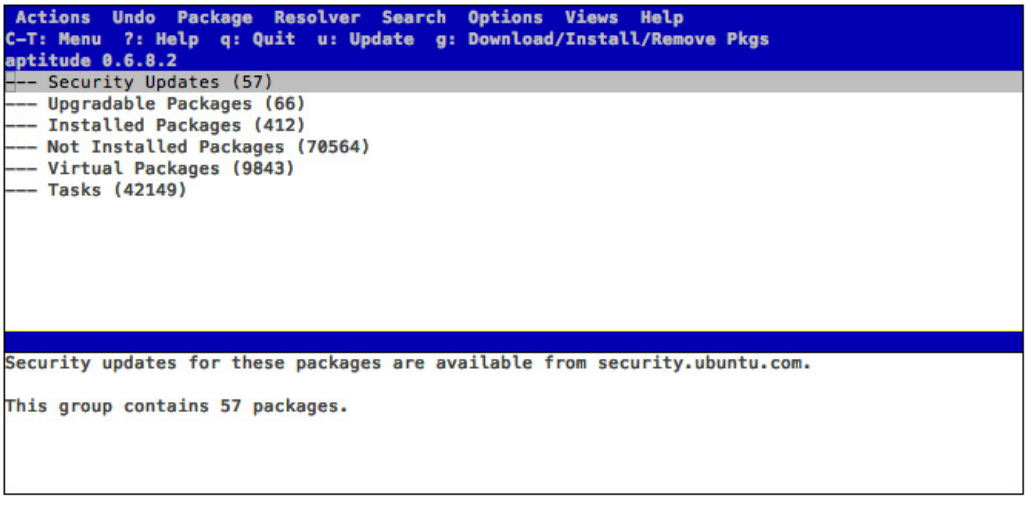


Figure 5-3 Aptitude Package Manager

Compile and Install GNU Software

One of the key benefits of open source software is that you have access to the source code. If the developer chooses to stop working on it, you can continue (if you know how). If you find a problem, you (or others) can fix it. In other words, you are in control of the situation and not at the mercy of a commercial developer you can't control. But having the source code means you need to be able to compile it too. Otherwise, all you have is a bunch of text files!

Although almost every piece of software in this book is available in RPM or .deb format, we will explore the process of compiling and building software from source code. Being able to do this has the benefit of allowing you to pick and choose compile-time options, which is something you can't easily do with prebuilt RPMs. Also, an RPM might be compiled for a specific architecture, such as the Intel x86, but that same code might run better if you compile it natively on, for example, your *Peta-extra-Tera-Core-class* CPU.

In this section, we will step through the process of compiling the **hello** package, a GNU software package that might seem useless at first but exists for good reasons. Most GNU software conforms to a standard method of compiling and installing; the **hello** package tries to conform to this standard, so it makes an excellent example.

Getting and Unpacking the Source Package

Although other relationship left a bad taste in your mouth, you are ready to try again. Perhaps things didn't quite work out because there were so many other factors to deal with—RPM with its endless options and seemingly convoluted syntax. And so, out with the old, in with the new. Maybe you'll be luckier this time around if you have more control over the flow of things. Although a little more involved, working directly with source code will give you more control over the software and how things take form.

Software that comes in source form is generally made available as a *tarball*—that is, its individual files are archived into a single large file and then compressed. The tools commonly used to do this are **tar**, which handles the process of combining many files into a single large file, and other utilities that handle the compression tasks (such as **gzip**, **xz**, and so on).

NOTE Typically, a single directory is selected in which to build and store tarballs. This allows the system administrator to keep the tarball of each package in a safe place in the event they need to pull something out of it later. It also lets all the administrators know which packages are installed on the system in addition to the base system. A good directory choice for this is **/usr/local/src**, since software installed from source code and local to a system is generally installed in **/usr/local**.

Let's try installing the hello package, one step at a time. We'll begin by first obtaining a copy of the source code.

We'll begin using the **wget** utility to pull down the latest copy of the hello program from the web site www.gnu.org/software/hello (or directly http://ftp.gnu.org/gnu/hello/hello-2.10.tar.gz). We use hello version 2.10 (**hello-2.10.tar.gz**) in this example. The file will be temporarily saved to the current user's **Downloads** directory. Type the following:

```
master@server:~$ wget -P  $HOME/Downloads \
http://ftp.gnu.org/gnu/hello/hello-2.10.tar.gz
```

After downloading the file, you will need to unpack (or untar) it. When unpacked, a tarball will generally create a new directory for all of its files. The hello tarball (**hello-2.10.tar.gz**), for example, creates the subdirectory **hello-2.10**. Most packages follow this standard. If you find a package that does not follow it, it is a good idea to create a subdirectory with a reasonable name and place all the unpacked source files there. This allows multiple builds or versions to occur at the same time without the risk of conflicts.

Change your current working directory to the ~/**Downloads**/ directory where the hello tarball was downloaded and saved to:

```
[master@server ~]$ cd ~/Downloads
```

Next, use the **tar** command to unpack and decompress the hello archive:

```
[master@server  Downloads]$ tar -xvzf hello-2.10.tar.gz
hello-2.10/
hello-2.10/build-aux/
....<OUTPUT TRUNCATED>....
hello-2.10/NEWS
hello-2.10/README-release
```

The **z** parameter in this **tar** command invokes **gzip** to decompress the file before the untar process occurs. The **v** parameter tells **tar** to show the name of the file it is untarring. This way, you'll know the name of the directory where all the sources are being unpacked.

NOTE You might encounter source code archives that end with the .tar.bz2 extension. Bzip2 is another popular compression algorithm, and GNU **tar** does support decompressing it on the command line with the **y** or **j** option, instead of the **z** parameter.

A new directory, called **hello-2.10**, should have been created for you during the untarring. Change to the new directory and list its contents:

```
[master@server  Downloads]$ cd hello-2.10  ; ls
```

Looking for Documentation

You have both now found each other (via the Internet). Now is probably a good time to look around and see if either of you comes with any special needs (documentation).

A good place to look for software documentation will be in the root of the software's directory tree. Once you are inside the directory with all of the source code, begin looking for documentation.

NOTE Always read the documentation that comes with the source code! If there are any special compile directions, notes, or warnings, they will most likely be mentioned here. You will save yourself a great deal of agony by reading the relevant files first.

So, then, what are the relevant files? These files typically have names like **README**, **INSTALL**, **README.1ST**, **README.NOW**, and so on. The developer might also have documentation in a directory aptly named **docs**.

The **README** file generally includes a description of the package, references to additional documentation (including the installation documentation), and references to the author of the package. The **INSTALL** file typically has directions for compiling and installing the package. These are not, of course, absolutes. Every package has its quirks. The best way to find out is simply to list the directory contents and look for obvious signs of additional documentation. Some packages use different capitalization: **readme**, **README**, **ReadMe**, and so on. (Remember that Linux is case-sensitive.)

While you're in the **/usr/local/src/hello-2.10** directory, use a pager (such as *less* or *more*) to view the **INSTALL** file that comes with the hello program, like so:

```
[master@server hello-2.10]$ less  INSTALL
```

Exit the pager by typing **q** when you are done reading the file.

Configuring the Package

You both want this relationship to work and possibly last longer than the previous ones. So this is a good time to establish guidelines and expectations.

Most packages ship with an auto-configuration script; it is safe to assume they do, unless their documentation says otherwise. These scripts are typically named **configure** (or **config**), and they can accept parameters. A handful of stock parameters are available across all **configure** scripts, but the interesting stuff occurs on a program-by-program basis. Each package will have a few features that can be enabled or disabled, or that have special values set at compile time, and they must be set up via **configure**.

To see what **configure** options come with a package, simply run

```
[master@server hello-2.10]$ ./configure  --help
```

Yes, those are two hyphens (**--**) before the word "help."

NOTE One commonly available option is **--prefix**. This option allows you to set the base directory where the package gets installed. By default, most packages default to using **/usr/local**. Each component in the package will install into the appropriate directory in **/usr/local**.

If you are happy with the default options that the **configure** script offers, type the following:

```
[master@server hello-2.10]$ ./configure
checking whether build environment is sane... yes
checking for gawk... gawk
checking for gcc... gcc
...<OUTPUT TRUNCATED>...
config.status: creating po/Makefile
```

With all of the options you want set up, a run of the **configure** script will create a special type of file called a *makefile.* Makefiles are the foundation of the compilation phase. Generally, if **configure** fails, you will not get a makefile. Make sure that the **configure** command did indeed complete without any errors.

Compiling the Package

Although this stage does not quite fit anywhere in our dating model, you might consider it as being similar to that period when you are so blindly in love that everything just flies by and a lot of things are just inexplicable.

To compile the hello package, all you need to do is run **make**, like so:

```
[master@server hello-2.10]$  make
```

The **make** tool reads the directives in the makefile(s) created by the **configure** script. These files tell **make** which files to compile and the order in which to compile them—which is crucial, since there could be hundreds of source files. Depending on the complexity of the application, the speed of your system, the available memory, and how busy it is doing other things, the compilation process could take a while to complete, so don't be surprised.

As **make** is working, it will display each command it is running and all the parameters associated with it. This output is usually the invocation of the compiler and all the parameters passed to the compiler—it's pretty tedious and uninteresting stuff that most people just gloss over!

If the compile goes through smoothly, you won't see any error messages. Most compiler error messages are clear and distinct, so don't worry about possibly missing an error. If you do see an error, don't panic. Most error messages don't reflect a problem with the program itself, but usually with the system in some way or another. Typically, these messages are the result of inappropriate file permissions or needed files, libraries, or executables that cannot be found.

In general, slow down and read the error message. Even if the format is a little odd, it might explain what is wrong in plain English, thereby allowing you to fix it quickly. If the error is still confusing, look at the documentation that came with the package to see if there is a mailing list or e-mail address you can contact for help. Most developers are more than happy to provide help, but you need to remember to be nice and to the point; in other words, don't start an e-mail with a rant about why the software is terrible!

Installing the Package

You've done almost everything else. You've found your partner, you've studied them, you've even compiled them. Now it's time to move them in with you—again!

Unlike the compile stage, the installation stage typically goes smoothly. Also unlike the previous steps, the install stage/step needs to be performed with elevated (superuser) privileges if the program is to be installed systemwide. In most cases, once the compile completes successfully, all that you need to do is run the following:

```
[master@server hello-2.10]$ sudo make install
```

This will install the package into the location specified by the default prefix (or the **--prefix**) argument that was used with the **configure** script earlier.

It will start the installation script (which is usually embedded in the makefile). Because **make** displays each command as it is executing it, you will see a lot of text fly by. Don't worry about it—it's perfectly normal. Unless you see an error message, the package will be safely installed.

If you do see an error message, it is most likely because of permissions problems. Look at the last file it was trying to install before failure, and then go check on all the permissions required to place a file there. You might need to use the **chmod**, **chown**, and **chgrp** commands for this step.

CAUTION If the software being installed is meant to be used and available systemwide, then the **make install** stage is almost always the stage that needs to be performed by the superuser (the root user). Accordingly, most install instructions will require you to become root before performing this step.

If, on the other hand, a regular user is compiling and installing a software package for their own personal use into a directory for which that user has full permissions (for example, by specifying **--prefix=/home/<USERNAME>/local/**), then there is no need to become root to run the **make install** stage.

Testing the Software

A common mistake administrators make is to go through the process of configuring and compiling, but then, when they install, they do not test the software to make sure that it runs as it should. Testing the software also needs to be done as a regular user, if the software is to be used by non-root users.

In our example, we'll run the **hello** command to verify that the permissions are correct and that users won't have problems running the program.

Assuming that you accepted the default installation prefix for the hello program (the relevant files will be under the **/usr/local** directory), use the full path to the program binary to execute it:

```
[master@server hello-2.10]$  /usr/local/bin/hello
Hello, world!
```

TIP While logged on as the root user, you can use the **su** program to temporarily become a regular user and test the functionality of an application. For example, to test the hello program as the user yyang, type the following:

```
[root@server hello-2.10]# su yyang -c /usr/local/bin/hello
```

That's it—you're done.

Cleanup

Once the package is installed, you can do some cleanup to get rid of all the temporary files created during the installation. Since you have the original source-code tarball, it is okay to get rid of the entire directory from which you compiled the source code. In the case of the hello program, you would get rid of **/usr/local/src/hello-2.10**.

Use the **rm** command to delete the entire source code and build directory for the hello application, like so:

```
[master@server hello-2.10]$  sudo rm -rf /usr/local/src/hello-2.10
```

The **rm** command, especially with the **-rf** parameter, is dangerous! It recursively removes an entire directory without stopping to verify any of the files. It is especially potent when run by the root user—it will shoot first and leave you asking questions later.

Just in case you decide to get rid of the original tarball from which you extracted the program source code, you can do this by running the following:

```
[master@server src]#  rm  -f  /usr/local/src/hello-2.10.tar.gz
```

Installing Snap Apps

You can install a similar hello application as well as numerous other applications that have been packaged and made available at https://snapcraft.io using the bundle approach we alluded to much earlier in this chapter. For example, on newer Ubuntu distros that have the **snapd** program installed, you can search for snaps that match the name **hello-world** by typing this:

```
$ snap find hello-world
```

Once you find the snap that you want, you can install it:

```
$ sudo snap install hello-world
```

You can list the newly installed snap as well as any existing snaps by running this:

```
$ snap list
```

The **hello-world** snap you just installed can be executed as follows:

```
$ /snap/bin/hello-world
```

Common Problems when Building from Source Code

The GNU hello program might not seem like a useful tool, but one valuable thing it provides is the ability to test the compiler on your system. If you've just finished the task of upgrading your compiler, compiling this simple program will provide a sanity check that indeed the compiler is working.

Following are some other problems (and their solutions) you might encounter when building from source.

Problems with Libraries

One problem you might run into is when the program can't find a file of the type *libsomething*.so and terminates for that reason. This file is a *library.* Linux libraries are synonymous with Dynamic Link Libraries (DLLs) in Windows. They are stored in several locations on the Linux system and typically reside in **/usr/lib/**, **/usr/lib64/**, and **/usr/local/lib/**.

If you have installed a software package that provides libraries needed by other programs in a location other than the standard locations, you will have to configure your system or shell to know where to look for those new libraries.

NOTE Linux libraries can be located anywhere on your file system. You'll appreciate the usefulness of this when, for example, you have to use the Network File System (NFS) to share a directory (or, in our case, software) among network clients. You'll find that this design makes it easy for other networked users or clients to run and use software or system libraries residing on the network.

There are two methods for configuring libraries on a Linux system. One is to modify **/etc/ld.so.conf** by adding the path of your new libraries, or you can place a custom configuration file for your application under the **/etc/ld.so.conf.d/** directory. Once this is done, use the **`ldconfig -m`** command to load in the new configuration.

You can also use the **LD_LIBRARY_PATH** environment variable to hold a list of library directories to look for library files. Read the man pages for **ldconfig** and **ld.so** for more information.

Missing Configure Script

Sometimes, you will download a package, unpack it and instantly type **`cd`** to get into its directory and run **`./configure`**. And you will probably be shocked when you see the message "No such file or directory." As stated earlier in the chapter, read the **README** and **INSTALL** files that are packaged with the software. Typically, the authors of the software are courteous enough to provide at least one of these two files. Source code for packages that you obtain directly from version control systems (such as git, svn, mercurial, and so on) may sometimes ship with an **autogen.sh** script, which you can run instead of **configure** or to generate the missing **configure** file if needed.

Broken Source Code

No matter what you do, it is possible that the source code you have is simply broken and the only person who can get it to work or make any sense of it is its original author or some other software developer. You might have already spent countless hours trying to get the application to compile and build before coming to this conclusion and throwing in the towel.

It is also possible that the author of the program has not documented some valuable or relevant information. In cases like this, you could try to see if precompiled binaries for the application already exist for your specific Linux distro in the form of RPMs, .deb files, and so on.

Summary

You've explored the common functions of the popular RPM and DPMS. You used various options to manipulate RPM and .deb packages by querying, installing, and uninstalling sample packages.

You learned about and explored various software management techniques using purely command-line tools. We also discussed a few GUI tools that are used on popular Linux distributions. The GUI tools are similar to the Windows Add/Remove Programs Control Panel applet—just point and click. The chapter also touched on other popular software management systems in Linux distros—namely, DNF, APT, Yum, and snapd.

Using the GNU hello open source program as an example, you went through the steps involved in configuring, compiling, and building software from the raw source code.

As a bonus, you also learned a thing or two about the mechanics of human relationships!

CHAPTER

6 Managing Users and Groups

Linux was designed from the ground up to be a multiuser operating system, but a multiuser operating system would not be much good without users! And this brings us nicely to the subject of this chapter—managing users and groups in GNU/Linux.

On shared computer systems, a system administrator is usually responsible for creating, managing, and deleting user accounts. The ability of a user to access resources on a system is determined by the permissions associated with the user. Associated with each user is some "baggage," which can include files, processes, resources, and other information. When dealing with a multiuser system, a system administrator needs to understand what constitutes a user (and the user's baggage) and a group, how they interact together, and how they impact available system resources.

In this chapter, we will examine the technique of managing users on a single host. We'll begin by exploring the traditional local datastore of user and group information. From there, we'll examine the system tools available to manage the datastore automatically.

What Exactly Constitutes a User?

Under Linux, every file and program must be owned by an *entity* (or user). This entity can be a (human) person, a daemon, a thing, a group, and so on. Each user has a unique identifier called a *user ID (UID).* Each user must also belong to at least one *group*—a collection of users. Users may belong to multiple groups. Groups also have unique identifiers, called *group IDs (GIDs).*

A running program inherits the rights and permissions of the user who invokes it. (An exception to this rule is SetUID and SetGID programs, discussed in more detail later on.)

Each user's rights can be defined in one of two ways: as those of a *normal user* or as a *superuser.* Normal users can access only what they own or have been given permission to access; permission is granted because the user either belongs to the file's group or because the file is accessible to all users. A superuser is allowed to access *all* files and programs in the system. The *root* user is a classic example of a superuser, and these two terms are sometimes used interchangeably.

If you are accustomed to Windows, you can draw some parallels between Windows user management concepts and Linux's user management concepts. Linux UIDs are comparable to Windows SIDs (security identifiers), for example. In contrast to Microsoft Windows, you might find the default GNU/Linux security model maddeningly simplistic: either you're root or you're not. Although it's a little less common, you can also implement finer grained access control through the use of access control lists (ACLs) in Linux, as you can with Windows.

Where User Information Is Kept

If you're already accustomed to user management in Microsoft Windows Server environments, you're definitely familiar with Active Directory (AD), which takes care of the nitty-gritty details (LDAP, SIDs, objects, Kerberos, and so on) of the user and group database. In contrast, Linux takes the path of traditional UNIX and keeps all user information in straight text files. This is beneficial for the simple reason that it allows you to make changes to user information without the need of any complicated or specialized tools, except for a simple text editor!

Folks new to Linux administration, however, need not fear the text editor! Later on, we discuss lots of simple and intuitive user management tools that come with most GNU/Linux distributions. For now, let's examine the text files that store user and group information in Linux.

NOTE This chapter covers the traditional Linux/UNIX methods for storing and managing user information. Chapter 27 of this book discusses some other mechanisms (such as LDAP) for storing and managing users and groups in Linux-based operating systems.

The /etc/passwd File

The **/etc/passwd** file stores the user's login, encrypted password entry, UID, default GID, name (sometimes called GECOS), home directory, and login shell. Each line in the file represents information about a user. The lines are made up of various standard fields, with each field delimited by a colon. A sample entry from a **passwd** file with its various fields is illustrated in Figure 6-1. The fields of the **/etc/passwd** file are discussed in detail in the sections that follow.

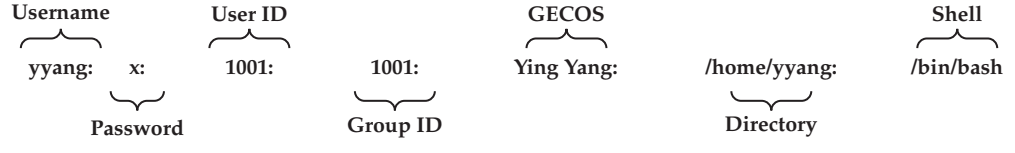


Figure 6-1 Fields of the /etc/passwd file

Username Field

This field is also referred to as the login field or the account field. It stores the name of the user on the system. The username must be a unique string and uniquely identifies a user to the system. Different sites use different conventions for generating human user login names.

A common method is to use the first letter of the user's first name and append the user's last name. This usually works, because the chances are relatively slim that an organization would have several users with the same first and last names. For example, for a user whose first name is "Ying" and whose last name is "Yang," a username of "yyang" can be assigned. Of course, several variations of this method are also used.

Password Field

This field contains the user's encrypted password. On most modern Linux systems, this field contains a letter *x* to indicate that shadow passwords are being used on the system (discussed further on, in this chapter). Because security is important, every regular user or human account on the system should have a *good* and *secure* password.

Passwords and Encryption

Most GNU/Linux distributions use modern encryption algorithms (such as Blowfish, SHA-512, and so on) to encrypt and store passwords on the system. Overall, the stronger and better encryption algorithms have increased the complexity and difficulty involved in cracking passwords. Of course, all of this only works assuming that your end users choose good passwords to begin with!

The process works as follows: When users enter their passwords at a login prompt, the password they enter is encrypted. The encrypted value is then compared against the user's password/shadow entry. If the two encrypted values match, the user is allowed to log into the system.

TIP Choosing good passwords is always a chore. Your users will inevitably ask, "What then, O' Mighty System Administrator, makes a good password?" Here's a good answer: a nonlanguage word (not English, not Spanish, not Yoruba; in fact, not a human-language word), preferably with mixed case, numbers, punctuation, and high entropy—in other words, a string that looks like line noise.

Well, this is all nice and wonderful, but if a password is too hard to remember, most people will quickly defeat its purpose by writing it down and keeping it in an easily accessible location. So better make it memorable! A good technique might be to choose a phrase and then pick the first letter of every word in the phrase. Thus, the phrase "coffee is VERY GOOD for you and me" becomes ciVG4y&m. The phrase is memorable, even if the resulting password isn't! The shadow password configuration file, **/etc/login.defs**, can used to configure and enforce various password parameters.

User ID Field (UID)

This field stores a unique number that the operating system and other applications use to identify the user and determine access privileges. It is the numerical equivalent of the Username field. The UID should be unique for every user.

The UID zero (0) is extra special and usually reserved for the *root* user. Any user who has a UID of 0 has root (administrative) access and thus has the full run (control) of the system. It is considered bad practice to allow any other users or usernames to have a UID of 0.

Different Linux distributions sometimes adopt specific UID numbering schemes. For example, newer Fedora and Red Hat Enterprise Linux (RHEL), openSUSE and Ubuntu distros reserve the UID 65534 for the user "nobody".

Group ID Field (GID)

The next field in the **/etc/passwd** file is the group ID entry. It is the numerical equivalent of the primary group to which the user belongs. This field also plays an important role in determining user access privileges. It should be noted that in addition to a primary group, a user can belong to other groups as well (more on this later in the section "The /etc/group File").

GECOS

This field can store various pieces of information for a user. It can act as a placeholder for the user description, full name (first and last name), telephone number, and so on. This field is optional and as a result can be left blank. It is also possible to store multiple entries in this field by simply separating the different entries with a comma.

NOTE Of historical note, GECOS is an acronym for General Electric Comprehensive Operating System (now referred to as GECOS) and is a carryover from the early days of computing.

Directory

This is usually the user's home directory, but it can also be any arbitrary location on the system. Each user who actually logs into the system needs a place for configuration files that are unique to that user. Along with configuration files, the directory (often referred to as the home directory) also stores the user's personal data such as documents, code, music, photos, and so on.

The *home directory* allows each user to work in an environment that he or she has specifically customized—without having to worry about the personal preferences and customizations of other users.

Startup Scripts

Startup scripts are not quite a part of the information stored in the users' database in Linux, but they nonetheless play an important role in determining and controlling a user's environment. In particular, the startup scripts in GNU/Linux are usually stored under the user's home directory, and hence the need to mention them while we're still on the subject of the directory (home directory) field in the **/etc/passwd** file.

(*continued*)

Each user is allowed to have his or her own configuration files; thus, the system appears to be customized for each particular user (even if other people are logged in at the same time). The customization of each individual user environment is done through the use of shell scripts, initialization files, run control files, and the like. These files can contain a series of commands to be executed by the shell that starts when a user logs in. In the case of the Bash shell, for example, one of its startup files is the **.bashrc** file. (Yes, there is a period in front of the filename—filenames preceded by periods, also called dot files, are hidden from normal directory listings.) The **.bashrc** script is similar in nature to the **autoexec.bat** batch file in the Windows world.

Various Linux software packages use application-specific and customizable options in directories or files that begin with a dot (.) in each user's home directory. Some examples are **.mozilla**, **.kde**, **.local**, **.ssh**, and **.ansible.cfg**. Here are some common dot files that may be present in each user's home directory:

- **.bashrc and .profile** Configuration files for the Bash shell.
- **.tcshrc and .login** Configuration files for tcsh.
- **.xinitrc** This script overrides the default script that gets called when you log into the X Window System.
- **.Xdefaults** This file contains defaults that you can specify for X Window System applications.

When you create a user's account, a set of default dot files are also created for the user; this is mostly for convenience, to help get the user started. The user management tools discussed later in this chapter help you do this automatically. The default or template files that are used to initially populate new user home directories are stored under the **/etc/skel** directory.

For the sake of consistency, most sites place home directories at **/home** and name each user's directory by that user's login name. Thus, for example, if your login name were "yyang," your home directory would be **/home/yyang**. The exception to this is for some special system accounts, such as a root user's account or a system service. The superuser's (root's) home directory in Linux is usually set to **/root**. An example of a special system service account that might need a specific working directory could be a web server whose web pages are served from the **/var/www/** directory.

In Linux, the decision to place home directories under **/home** is strictly arbitrary, but it does make organizational sense. The system really doesn't care where we place home directories, so long as the location for each user is specified in the password file.

Shell

When users log into the system, they expect an environment that can help them be productive. When logged in at the command line, the first program that users encounter is called a *shell.* If you're used to the Windows side of the world, you might equate this with command.com (cmd), Program Manager, PowerShell, or Windows Explorer.

Under GNU/Linux, most shells are text-based. A popular default user shell in Linux is the Bourne Again Shell, or Bash for short. Linux comes with several shells from which to choose—you can see most of them listed in the **/etc/shells** file. Deciding which shell is right for you is kind of like choosing a favorite tea or beer—what's right for you isn't right for everyone, but still, everyone tends to get defensive about their choice!

Interestingly enough, you are not limited only to the list of shells declared in **/etc/shells**. In the strictest of definitions, the password entry for each user doesn't list what shell to run so much as it lists what program to run first for the user. Of course, most users prefer that the first program run be a shell, such as Bash.

The /etc/shadow File

This is the encrypted password file that stores the encrypted password information for user accounts. In addition to storing the encrypted password, the **/etc/shadow** file stores optional password aging or expiration information. The introduction of the **shadow** file came about because of the need to separate encrypted passwords from the **/etc/passwd** file. This was necessary because the ease with which the encrypted passwords could be cracked was growing with the increase in the processing power of commodity computers (home PCs). The idea was to keep the **/etc/passwd** file readable by all users without storing the encrypted passwords in it and then make the **/etc/shadow** file readable only by root or other privileged programs that require access to that information. An example of such a program would be the *login* program.

You might wonder, "Why not just make the regular **/etc/passwd** file readable by root only or other privileged programs?" Well, it isn't that simple. By having the password file open for so many years, the rest of the system software that grew up around it relied on the fact that the password file was always readable by all users. Changing this could cause some software to fail.

Just as in the **/etc/passwd** file, each line in the **/etc/shadow** file represents information about a user. The lines are made up of various standard fields, shown next, with each field delimited by a colon:

- Login name
- Encrypted password
- Days since January 1, 1970, that password was last changed
- Days before password may be changed
- Days after which password must be changed
- Days before password is to expire that user is warned
- Days after password expires that account is disabled

- Days since January 1, 1970, that account is disabled
- A reserved field

A sample entry from the **/etc/shadow** file is shown here for the user account *mmel:*

```
mmel:$1$HEWdPIJ.$qX/RbB.TPGcyerAVDlF4g.:12830:0:99999:7:::
```

UNIX Epoch: January 1, 1970

January 1, 1970, 00:00:00 UTC was chosen as the starting point or origin for keeping time on UNIX systems. That specific instance in time is also known as the UNIX epoch. Time measurements in various computing fields are counted and incremented in seconds from the UNIX epoch. Put simply, it is a count of the seconds that have gone past since January 1, 1970, 00:00:00.

An interesting UNIX time—1000000000—fell on September 9, 2001, at 1:46:40 A.M. (UTC). Another interesting UNIX time—1234567890—fell on February 13, 2009, at 11:31:30 P.M. (UTC). Numerous web sites are dedicated to calculating and displaying the UNIX epoch, but you can quickly obtain the current value by running this command at the shell prompt:

```
# date +%s
```

The /etc/group File

The **/etc/group** file contains a list of groups, with one group per line. Each group entry in the file has four standard fields, each colon-delimited, as in **/etc/passwd** and **/etc/shadow**.

Each user on the system belongs to at least one group, that being the user's default group. Users can then be assigned to additional groups if needed. You will recall that the **/etc/passwd** file contains each user's default group ID (GID). This GID is mapped to the group's name and other members of the group in the **/etc/group** file. The GID should be unique for each group.

The group file must be world-readable so that applications can test for associations between users and groups. Following are the fields of each line in **/etc/group**:

- **Group name** The name of the group
- **Group password** Optional, but if set, allows users who are not part of the group to join
- **Group ID (GID)** The numerical equivalent of the group name
- **Group members** A comma-separated list

A sample group entry in the **/etc/group** file is shown here:

```
bin:x:1:
```

This entry is for the *bin* group. The GID for the group is 1.

User Management Tools

One of the many benefits of having plain-text password database files that have a well-defined format is that it is easy for anyone to write custom management tools. Indeed, many site administrators have already done this to integrate their tools along with the rest of their organization's infrastructure. For example, a new user onboarding process can be easily extended to capture, include, or update extra information such as corporate phone number, e-mail, office location, LDAP address book, and so on. Of course, not everyone wants or is able to write custom tools, which is why Linux comes with several existing tools that do the job for you.

In this section, we discuss user management tools that can be launched from the command-line interface, as well as graphical user interface (GUI) tools. Of course, learning how to use both is the preferred route, since they both have advantages.

Command-Line User Management

You can choose from among several command-line tools to perform the same actions performed by the GUI tools. Some popular command-line tools are **useradd**, **userdel**, **adduser**, **usermod**, **deluser**, **groupadd**, **groupdel**, and **groupmod**. The compelling advantage of using command-line tools for user management, besides speed, is the fact that the tools can usually be incorporated into other automated functions (such as scripts).

useradd

As the name implies, **useradd** allows you to add a single user to the system. Unlike the GUI tools, this tool has no interactive prompts. Instead, all parameters must be specified on the command line.

Here's the syntax for using this tool along with some oft used options:

```
Usage: useradd [options] LOGIN
Options:
  --base-dir BASE_DIR ; --comment COMMENT;   --home-dir HOME_DIR;
  --expiredate EXPIRE_DATE; --gid GROUP; --groups GROUPS;
  --create-home; --no-create-home;  --password PASSWORD; --system;
  --shell SHELL; --uid UID; --key KEY=VALUE; --user-group
```

Take note that most of the options are optional. The **useradd** tool assumes preconfigured defaults in its usage. The only non-optional parameter is the **LOGIN** parameter or the desired username. Also, don't be intimidated by this long list of options! They are all quite easy to use, and some of them are described in Table 6-1.

Option	Description
`-c, --comment`	Allows you to set the user's name in the GECOS field. As with any command-line parameter, if the value includes a space, you will need to add quotes around the text. For example, to set the user's name to Ying Yang, you would have to specify `-c "Ying Yang"`.
`-d, --home-dir`	By default, the user's home directory is **/home/user_name**. When a new user is created, the user's home directory is created along with the user account, so if you want to change the default to another place, you can specify the new location with this parameter.
`-e, --expiredate`	It is possible for an account to expire after a certain date. By default, accounts never expire. To specify a date, use the *YYYY-MM-DD* format. For example, `-e 2021-10-28` means the account will expire on October 28, 2021.
`-f, --inactive`	This option specifies the number of days after a password expires that the account is still usable. A value of `0` (zero) indicates that the account is disabled immediately. A value of `-1` will never allow the account to be disabled, even if the password has expired. (For example, `-f 3` will allow an account to exist for three days after a password has expired.) The default value is `-1`.
`-g, --gid`	Using this option, you can specify the user's default group in the password file. You can use a number or name of the group; however, if you use a name of a group, the group must exist in the **/etc/group** file.
`-G, --groups`	This option allows you to specify additional groups to which the new user will belong. If you use the `-G` option, you must specify at least one additional group. You can, however, specify additional groups by separating the elements in the list with commas. For example, to add a user to the project and admin groups, you would specify `-G project,admin`.
`-m, --create -home [-k skel-dir ]`	By default, the system automatically creates the user's home directory. This option is the explicit command to create the user's home directory. Part of creating the directory is copying default configuration files into it. These files come from the **/etc/skel** directory by default. You can change this by using the secondary option `-k skel-dir`. (You must specify `-m` in order to use `-k`.) For example, to specify the **/etc/adminskel** directory, you would use `-m -k /etc/adminskel`.
`-M, --no-create-home`	If you used the `-m` option, you cannot use `-M`, and vice versa. This option tells the command *not* to create the user's home directory.
`-N, --no-user-group`	Some Linux distros automatically create a new group with the same name as the new user's login as part of the process of adding a user. You can disable this behavior by using this option on such distros.

Table 6-1 Options for the useradd Command (*continued*)

Option	Description
-s, shell	A user's login shell is the first program that runs when a user logs into a system. This is usually a command-line environment, unless you are logging in from the X Window System login screen. By default, this is the Bourne Again Shell (**/bin/bash**), though some folks like to use other shells, such as the Turbo C Shell (**/bin/tcsh**).
-u, --uid	By default, the program will automatically find the next available UID and use it. If, for some reason, you need to force a new user's UID to be a particular value, you can use this option. Remember that UIDs must be unique for all users.
LOGIN or **username**	Finally, the only parameter that *isn't* optional! You must specify the new user's login name.

Table 6-1 Options for the useradd Command

usermod

The **usermod** command allows you to modify an existing user in the system. It works in much the same way as **useradd**. Its usage is summarized here:

```
Usage: usermod [options] LOGIN
Options:
  -c, --comment COMMENT         new value of the GECOS field
  -d, --home HOME_DIR           new home directory for the user account
  -e, --expiredate EXPIRE_DATE  set account expiration date to EXPIRE_DATE
  -f, --inactive INACTIVE       set password inactive after expiration
  -g, --gid GROUP               force use GROUP as new primary group
  -G, --groups GROUPS           new list of supplementary GROUPS
  -a, --append                  append the user to supplemental GROUPS
  -l, --login NEW_LOGIN         new value of the login name
  -L, --lock                    lock the user account
  -m, --move-home               move contents of the home directory
  -o, --non-unique              allow using duplicate (non-unique) UID
  -p, --password PASSWORD       use encrypted password for the new password
  -s, --shell SHELL             new login shell for the user account
  -u, --uid UID                 new UID for the user account
  -U, --unlock                  unlock the user account
```

Every option you specify when using this command results in that particular parameter being modified for the user. All but one of the parameters listed here are identical to the parameters documented for the **useradd** command. The one exception is **-l**.

The **-l** option allows you to change the user's login name. This and the **-u** option are the only options that require special care. Before changing the user's login or UID, you must make sure the user is not logged into the system or running any processes. Changing this information if the user is logged in or running processes will cause unpredictable results.

userdel

The **userdel** command does the exact opposite of **useradd**—it removes existing users. This straightforward command has two commonly used optional parameters and one required parameter listed here:

```
Usage: userdel [options] LOGIN
Options:
 -f, --force       force removal of files,  even if not owned by user
 -r, --remove      remove home directory and mail spool
```

groupadd

The group-related commands are similar to the user commands; however, instead of working on individual users, they work on groups listed in the **/etc/group** file. Note that changing group information does not cause user information to be automatically changed. For example, if you remove a group whose GID is 100 and a user's default group is specified as 100, the user's default group would not be updated to reflect the fact that the group no longer exists.

The **groupadd** command adds groups to the **/etc/group** file. The command-line options for this program are as follows:

```
Usage: groupadd [options] GROUP
```

Table 6-2 describes some common **groupadd** command options.

groupdel

Even more straightforward than **userdel**, the **groupdel** command removes existing groups specified in the **/etc/group** file. The only usage information needed for this command is

```
Usage: groupdel group
```

where **group** is the name of the group to remove.

Option	Description
-g gid	Specifies the GID for the new group as **gid**. This value must be unique, unless the **-o** option is used. By default, this value is automatically chosen by finding the first available value greater than or equal to 1000.
-r, **--system**	The **-r** option tells **groupadd** that the group being added is a system group and should have the first available GID under 999. This is controlled by the value of the variables SYS_GID_MIN and SYS_GID_MAX defined in **/etc/login.defs**.
-f, **--force**	This is the force flag. This will cause **groupadd** to exit without an error when the group about to be added already exists on the system. If that is the case, the group won't be altered (or added again).
GROUP	This parameter is required. It specifies the name of the group to be added.

Table 6-2 Options for the groupadd Command

groupmod

The **groupmod** command allows you to modify the parameters of an existing group. The syntax and options for this command are shown here:

```
Usage: groupmod [options] GROUP
Options:
-g, --gid GID               change the group ID to GID
-n, --new-name NEW_GROUP    change the name to NEW_GROUP
-o, --non-unique            allow to use a duplicate (non-unique) GID
-p, --password PASSWORD     change the password to this (encrypted) PASSWORD
```

The **-g** option allows you to change the GID of the group, and the **-n** option allows you to specify a new name of a group. An existing GROUP name is the last parameter.

TIP Debian-based Linux distributions like Ubuntu standardize on a slight variation of the low-level command-line user and group management utilities (**useradd**, **groupadd**, **userdel**, **groupdel**, and so on) used on Red Hat–based distros. User and group management tasks on Debian-based systems are often done via the conversely named tools, such as **adduser**, **addgroup**, **deluser**, **delgroup**, and so on. The utilities **adduser** and **addgroup** present a friendlier and interactive front-end for adding users and groups to the system. These tools consult the **/etc/adduser.conf** configuration file.

GUI User Managers

The obvious advantage to using the GUI tool is ease of use. It is usually just a point-and-click affair. Many of the Linux distributions come with their own GUI user managers. Fedora, CentOS, and RHEL have a utility called **system-config-users**, and openSUSE/SUSE Linux Enterprise (SLE) have a YaST module that can be invoked with **yast2 users**. Ubuntu uses a tool called **Users Account**, which is bundled with the **gnome-control-center** system applet. All these tools allow you to add, edit, and maintain the users on your system. These GUIs work just fine—but you should be prepared to fall back to the command-line tools in situations where you don't have access to the pretty GUI front-ends. Most of these interfaces can be found in the System → Administration menu within the GNOME or KDE desktop environment. They can also be launched directly from the command line.

After installing it, you can launch Fedora's GUI user manager by typing the following:

```
[root@fedora-server ~]#   system-config-users
```

In openSUSE or SLE, to launch the user management YaST module you'd type this:

```
opensuse-server:~ #  yast2  users
```

In Ubuntu, to launch the user management tool (see Figure 6-2), you'd type this:

```
master@ubuntu-server:~$   gnome-control-center    user-accounts
```

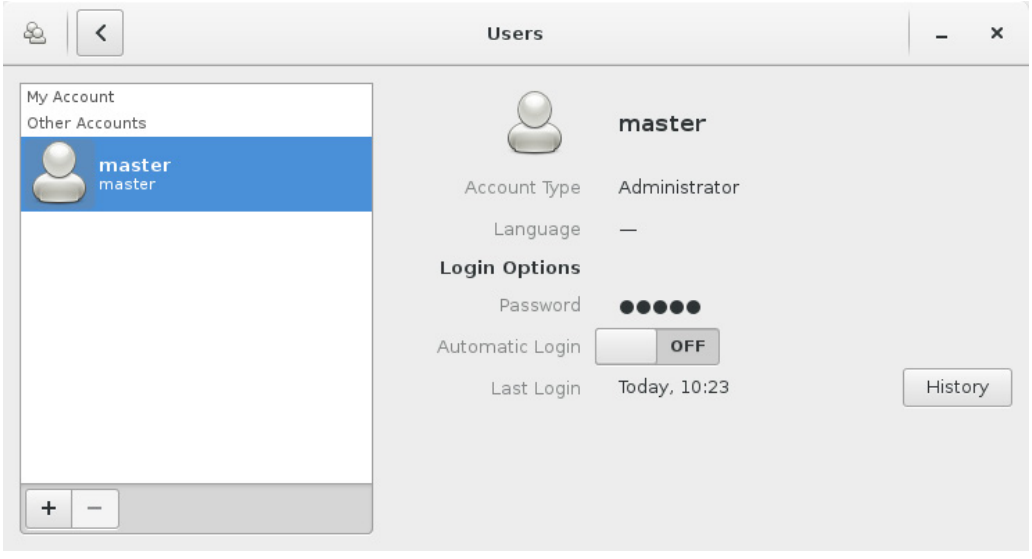


Figure 6-2 Ubuntu/GNOME Users Settings tool

Users and Access Permissions

Linux determines whether a user or group has access to files, programs, or other resources on a system by checking the overall effective permissions on the resource. The traditional permissions model in Linux is simple—it is based on four basic access types or rules and two special access permissions. The following access types and permissions are possible:

- **(r)** Read permission
- **(w)** Write permission
- **(x)** Execute permission
- **(-)** No permission or no access
- **(s)** SetUID or SetGID permission (special permission)
- **(t)** Sticky bit (special permission; useful for preventing users from deleting other users' files or directories)

In addition, these permissions can be applied to three classes of users:

- **Owner** The owner of the file or application
- **Group** The group that owns the file or application
- **Everyone** All users

The elements of this model can be combined in various ways to permit or deny a user (or group) access to any resource on the system. There is, however, a need for an additional type of permission-granting mechanism in Linux. This need arises because every application in Linux must run in the context of a user. This is explained in the next section on SetUID and SetGID programs.

Understanding SetUID and SetGID Programs

Normally, when a program is run by a user, it inherits all of the user's rights (or lack thereof). For example, if a user can't read the **/var/log/messages** file, neither can the program/application that is needed to view the file. Note that this permission can be different from the permissions of the user who owns the program or binary. Consider the **ls** program (which is used to generate directory listings), for example. It is owned by the root user. Its permissions are set so that all users of the system can run the program. Thus, if the user *yyang* runs **ls**, that instance of **ls** is bound by the permissions granted to the user *yyang*, not *root*.

However, there is an exception. Programs can be tagged with what's called a *SetUID bit* (also called a sticky bit), which allows a program to be run with permissions from the program's owner, not the user who is running it. Using **ls** as an example, setting the SetUID bit on it and having the file owned by root means that if the user yyang runs **ls**, that instance of **ls** will run with root permissions, not with yyang's permissions. The *SetGID bit* works the same way, except instead of applying to the file's owner, it is applied to the file's group setting.

To enable the SetUID bit or the SetGID bit, you use the **chmod** command. To make a program SetUID, prefix whatever permission value you are about to assign it with a **4**. To make a program SetGID, prefix whatever permission you are about to assign it with a **2**.

For example, to make **/bin/ls** a SetUID program (which is a very bad idea, by the way), you would use this command:

```
[master@server ~]$  sudo   chmod 4755 /bin/ls
```

The following variation of the **chmod** command also adds the user sticky bit:

```
[master@server ~]$  sudo   chmod u+s /bin/ls
```

To undo the effect of the previous command, type this:

```
[master@server ~]$  sudo   chmod 755 /bin/ls
```

The following variation of the **chmod** command removes the user sticky bit:

```
[master@server ~]$  sudo   chmod  u-s  /bin/ls
```

To make **/bin/ls** a SetGID program (which is also bad idea), you would type:

```
[master@server ~]$  sudo   chmod g+s /bin/ls
```

To remove the SetGID attribute from the **/bin/ls** program, type:

```
[master@server ~]$  sudo   chmod g-s /bin/ls
```

Sticky Bit

The sticky bit is a special permission that can be used to prevent users from deleting other users' files or directories. The sticky bit can be set by the owner of a file or directory, and the results cannot be overridden by any other users except, of course, for the superuser (root). The sticky bit is especially useful for managing the deletion of files in shared or public folders.

Let's create a sample directory and file under the **/tmp** directory:

```
[master@server ~]$   mkdir  /tmp/chapter-6  && \
touch  /tmp/chapter-6/sticky-file.txt
```

To view what the new directory looks like without the sticky bit set, type the following:

```
[master@server ~]$   ls -ld /tmp/chapter-6/
drwxrwxr-x. 2 master master 60 Oct 24 16:59 /tmp/chapter-6/
```

Now set the sticky bit on the directory created:

```
 [master@server ~]$   chmod +t /tmp/chapter-6/
```

View the new directory permissions by typing this:

```
[master@server ~]$  ls  -ld  /tmp/chapter-6/
drwxrwxr-t. 2 master master 60 Oct 24 16:59 /tmp/chapter-6/
```

With the sticky bit set, another user will not be able to maliciously delete files created by the *master* user under the **/tmp/chapter-6** folder.

You can remove the sticky bit on the folder as follows:

```
[master@server ~]$   chmod  -t /tmp/chapter-6/
```

Pluggable Authentication Modules

Pluggable Authentication Modules (PAM) is a system of libraries that allows the use of a centralized authentication mechanism on Linux/UNIX systems. Besides providing a common authentication scheme on a system, the use of PAM allows for a lot of flexibility and control over authentication for application developers, as well as for system administrators.

Traditionally, programs that grant users access to system resources have performed the user authentication through some built-in mechanism. Although this worked great for a long time, the approach was not very scalable, and more sophisticated methods were required. This led to a number of ugly hacks to abstract the authentication mechanism, eventually leading to a Linux implementation of PAM.

The idea behind PAM is that instead of applications reading the password file, they would simply ask PAM to perform the authentication. PAM could then use whatever authentication mechanism the system administrator wanted. For many environments, the mechanism of choice is still a simple password file. And why not? It does what we want. Most users understand the need for it, and it's a well-tested method to get the job done.

In this section, we discuss the use of PAM under the Fedora distribution. Note that although the placement of files may not be exactly the same in other distributions, the underlying configuration files and concepts still apply.

How PAM Works

PAM is to other Linux programs what a Dynamic Link Library (DLL) is to a Windows application—it is just a library. When programs need to perform authentication on some user, they call a function that exists in the PAM library. PAM provides a library of functions that an application can use to request that a user be authenticated.

When invoked, PAM checks the configuration file for that application. If it finds no application-specific configuration file, it falls back to a default configuration file. This configuration file tells the library what types of checks need to be done to authenticate the user. Based on this, the appropriate module is called upon. Fedora, RHEL, and CentOS distros store these modules in the **/lib64/security** or **/lib/security** directory.

This module can check any number of things. It can simply check the **/etc/passwd** file or the **/etc/shadow** file, or it can perform a more complex check, such as calling on an LDAP server. Once the module has made the determination, an "authenticated" or "not authenticated" message is passed back to the calling application.

If this seems like a lot of steps for what should be a simple check, you're almost correct. Each module here is small and does its work quickly. From a user's point of view, there should be no noticeable difference in performance between an application that uses PAM and one that does not. From a system administrator's and developer's point of view, the flexibility this scheme offers is incredible.

PAM's Files and Their Locations

On a Fedora or Red Hat–type system, PAM puts configuration files in certain places. These file locations and their definitions are listed in Table 6-3.

Looking at the list of file locations in Table 6-3, you might ask why PAM needs so many different configuration files. "One configuration file per application? That seems crazy!" Well, maybe not. The reason PAM allows this is that not all applications are created equal. For instance, a Post Office Protocol (POP) mail server that uses the Dovecot mail server might

File Location	Definition
/lib64/security or **/lib/security (32 bit)**	Dynamically loaded authentication modules called by the actual PAM library.
/etc/security	Configuration files for the modules located in **/lib64/security** (or **/lib/security**).
/etc/pam.d	Configuration files for each application that uses PAM. If an application that uses PAM does not have a specific configuration file, the default is automatically used.

Table 6-3 Important PAM Directories

want to allow all of a site's users to fetch mail, but the login program might want to allow only certain users to log into the console. To accommodate this, PAM needs a configuration file for POP mail that is different from the configuration file for the login program.

Configuring PAM

The configuration files that we will be discussing here are located in the **/etc/pam.d** directory. If you want to change the configuration files that apply to specific modules in the **/etc/security** directory, you should consult the documentation that came with the module. Remember that PAM is just a framework, and specific modules can be written by anyone!

PAM configuration files are intrinsically interesting because of their "stackable" nature. Every line of a configuration file is evaluated during the authentication process (with the exceptions shown next). Each line specifies a module that performs some authentication task and returns either a success flag or failure flag. A summary of the results is returned to the application program calling PAM.

NOTE By "failure," we do not mean the program did not work. Rather, we mean that when some process was undertaken to verify whether a user could do something, the return value was "NO." PAM uses the terms "success" and "failure" to represent this information that is passed back to the calling application.

Each PAM configuration file in the **/etc/pam.d/** directory consists of lines that have the syntax/format

```
module_type   control_flag  module_path   arguments
```

where **`module_type`** represents one of four types of modules: **auth**, **account**, **session**, or **password**. Comments must begin with the hash (#) character. Table 6-4 lists these module types and their functions.

The **`control_flag`** allows you to specify how you want to deal with the success or failure of a particular authentication module. Some common control flags are described in Table 6-5.

Module Type	Function
auth	Instructs the application or program to prompt the user for a password and then grants both user and group privileges. It is used for authentication.
account	Performs no authentication, but determines access from other factors, such as time of day or location of the user. It is used for authorization. For example, the root login can be given only console access this way.
session	Specifies what, if any, actions need to be performed before or after a user is authenticated. (For example, creating a log entry of successful or failed attempts).
password	These modules allow users to change their passwords (if appropriate).

Table 6-4 PAM Module Types

Control Flag	Description
required	If this flag is specified, the module *must* succeed in authenticating the individual. If it fails, the returned summary value must be **failure**.
requisite	This flag is similar to **required**; however, if **requisite** fails authentication, modules listed after it in the configuration file are not called, and a failure is immediately returned to the application. This allows you to require that certain conditions hold true before even a login attempt is accepted (for example, the user must be on the local area network and cannot be attempting to log in over the Internet).
sufficient	If a **sufficient** module returns a success and there are no more required or sufficient control flags in the configuration file, PAM returns a success to the calling application.
optional	This flag allows PAM to continue checking other modules, even if this one has failed. In other words, the result of this module is ignored. For example, you might use this flag when a user is allowed to log in even if a particular module has failed.
include	This flag is used for including all lines or directives from another configuration file specified as an argument. It is used as a way of chaining or stacking together the directives in different PAM configuration files.
substack	This is used for including all lines (stacks) of a given type from the specified PAM configuration file. It differs from the **include** control flag, because a failure in the specified **substack** does not mean that checks of the rest of the module stack will be automatically skipped. Evaluation of the current module stack will continue.

Table 6-5 PAM Control Flags

The **module_path** specifies the relative module filename or the full path to the module that performs the authentication task. The modules are usually stored under the **/lib64/security/**(or **/lib/security/**) directory.

The final entry in a PAM configuration line is **arguments**, which represents the parameters passed to the authentication module. Although the parameters are specific to each module, some generic options can be applied to all modules. Some of the more common arguments are described in Table 6-6.

A Sample PAM Configuration File

Let's examine a sample PAM configuration file, **/etc/pam.d/login**:

```
1. #%PAM-1.0
2. auth       substack     system-auth
3. auth       include      postlogin
4. account    required     pam_nologin.so
5. account    include      system-auth
6. password   include      system-auth
7. # pam_selinux.so close should be the first session rule
8. session    required     pam_selinux.so close
9. session    required     pam_loginuid.so
```

Please note that line numbers have been added to aid readability of the file.

Argument	Description
debug	Sends debugging information to the system logs.
no_warn	Does not give warning messages to the calling application.
use_first_pass	Does not prompt the user for a password a second time. Instead, the password that was entered in the preceding **auth** module should be reused for the user authentication. (This option applies to the **auth** and **password** modules only.)
try_first_pass	This option is similar to `use_first_pass` because the user is not prompted for a password the second time. However, if the existing password causes the module to return a failure, the user is then prompted for a password again.
use_mapped_pass	This argument instructs the module to take the cleartext authentication token entered by a previous module and use it to generate an encryption/decryption key with which to safely store or retrieve the authentication token required for this module.
expose_account	This argument allows a module to be less discreet about account information—as deemed fit by the system administrator.
nullok	This argument allows the called PAM module to allow blank (null) passwords.

Table 6-6 PAM Configuration Arguments

The first line begins with a hash (#) symbol and is therefore a comment. Thus, we can ignore it:

```
#%PAM-1.0
```

Let's go on to line 2:

```
auth       substack     system-auth
```

The **module_type** is **auth**, and this means that it deals with validating the user's authentication credentials (for example, username and password). The **control_flag** is set to **substack**, so this line will cause a jump to the specified PAM configuration file, **/etc/pam.d/system-auth**. Evaluation will continue in the manner specified in the **system-auth** file, and control will eventually be returned to the login PAM module. There are no arguments on this line.

The rest of the lines in the file are presented before their explanation, starting with line 3:

```
auth       include      postlogin
```

Similar to the second line, line 3 wants to validate the user via authentication.

The **control_flag** is set to **include**. This will include all lines with the type **auth** from the **/etc/pam.d/postlogin** configuration file. There are no arguments on this line.

```
account    required     pam_nologin.so
```

In line 4, the **module_type** is **account**, and this means that it is used to restrict/permit access of an account to a system service/resource based on various factors.

The **control_flag** is **required**. This means that if this module fails, PAM returns a failure result to the calling application, but evaluation of the next module in the stack will continue.

pam_nologin.so (**/lib64/security/pam_nologin.so**) is the name of the actual PAM module to call. The **pam_nologin.so** module checks for the **/etc/nologin** file. If it is present, only root is allowed to log in; others are turned away with an error message. If the file does not exist, it always returns a success.

```
account    include      system-auth
```

In line 5, the **module_type** is **account**. The **control_flag** is set to **include**. This will include all lines with the type **account** from the **/etc/pam.d/system-auth** configuration file. There are no arguments on this line.

```
password   include      system-auth
```

In line 6, the **module_type** is **password**. The **control_flag** is set to **include**. This will include all lines with the type **password** from the **/etc/pam.d/system-auth** configuration file. There are no arguments on this line.

```
# pam_selinux.so close should be the first session rule
```

Line 7 is commented out. We can thus ignore it.

```
session    required     pam_selinux.so close
```

In line 8, the **module_type** is **session**, which is used for specifying what, if any, actions need to be performed before or after a user is logged in. The **control_flag** is set to **required**, and **pam_selinux.so** (**/lib64/security/pam_selinux.so**) is the name of the actual PAM module to call. The **pam_selinux.so** module is used for setting the default security context. It is mostly popular on Red Hat–based distributions. The **close** argument in this case is passed to the **pam_selinux.so** module.

```
session    required     pam_loginuid.so
```

In line 9, the **module_type** is **session**. The **control_flag** is set to **required**. **pam_loginuid.so** is the name of the actual PAM module to call.

The same concepts apply to the other configuration files under the **/etc/pam.d** directory. If you need more information about what a particular PAM module does or about the arguments it accepts, you can consult the man page for the module. For example, to find out more about the **pam_loginuid.so** module, you would issue this command:

```
# man pam_loginuid
```

The "Other" File

As mentioned earlier, if PAM cannot find a configuration file that is specific to an application, it will use a generic configuration file instead. This generic configuration file is called **/etc/pam.d/other**. By default, the "other" configuration file is set to a paranoid setting so that all authentication attempts are logged and then promptly denied. It is recommended you keep it that way.

D'oh! I Can't Log In!

Don't worry—screwing up a setting in a PAM configuration file happens to everyone. Consider it part of learning the ropes. First thing to do: Don't panic. Like most configuration errors under Linux, you can fix things by booting into single-user mode (Chapter 8) and fixing the errant file.

If you've inadvertently broken your login configuration file and need to bring it back to a sane state, here is a safe setting you can put in:

```
auth          required      pam_unix.so
account       required      pam_unix.so
password      required      pam_unix.so
session       required      pam_unix.so
```

This setting will give Linux the default behavior of simply looking into the **/etc/passwd** or **/etc/shadow** file for a password. This should be good enough to get you back in, where you can make the changes you meant to make!

NOTE The **pam_unix.so** module is a standard Linux/UNIX authentication module. According to the module's man page, it uses standard calls from the system's libraries to retrieve and set account information as well as authentication. Usually, this is obtained from the **/etc/passwd** file and from the **/etc/shadow** file as well if **shadow** is enabled.

Debugging PAM

Like many other GNU/Linux services, PAM makes excellent use of the system log files (see Chapter 9). If things are not working the way you expect them to, begin by looking at the tail end of the log files and see if PAM is spelling out what happened. More than likely, it is. You should then be able to use this information to change your settings and fix the problem. On Red Hat–like distros, the main system log file to monitor is the **/var/log/messages** file, and on Debian-like distros, the equivalent log file is the **/var/log/syslog** file.

A Grand Tour

In this section, you'll see real-world usage of the utilities discussed so far. We take a step-by-step approach to creating, modifying, and removing users and groups. Some new and relevant commands are also introduced and used.

Creating Users with useradd

On our sample Fedora server, we will add new user accounts and assign passwords with the **useradd** and **passwd** commands, as follows:

1. Create a new user whose full name is "Ying Yang" with the login name (account name) of *yyang*. Type the following:

   ```
   [master@server ~]$  sudo useradd -c "Ying Yang" yyang
   ```

   The user will be created with the usual Fedora default attributes. The entry in the **/etc/passwd** file will be

   ```
   yyang:x:1001:1001:Ying Yang:/home/yyang:/bin/bash
   ```

   From this entry, you can tell these things about the Fedora (and RHEL) default new user values:

   - The UID number is the same as the GID number. The value is 1001 in this example.
   - The default shell for new users is the Bash shell (**/bin/bash**).
   - A home directory is automatically created for all new users (for example, **/home/yyang**).

2. Use the **passwd** command to create a new password for the username yyang. Set the password to be 19gan19, and repeat the same password when prompted. Type the following:

   ```
   [master@server ~]$ sudo   passwd  yyang
   Changing password for user yyang.
   ...<OUTPUT TRUNCATED>...
   ```

NOTE Several PAM modules may be called whenever the **passwd** command is called. For example, on Red Hat–like distros, one of the modules is responsible for checking that the selected password meets the password strength policies on the system—policies such as the password length, complexity, and so on. The module called in the previous example is **pam_pwquality.so**. To learn more about this particular module, see its manual page (**man pam_pwquality**).

3. Create another user account called *mmellow* for the user, with a full name of "Mel Mellow," but this time, change the default Fedora behavior of creating a group with the same name as the username. This is accomplished by specifying the **-N** option, which will cause the user to be added to the default "users" group. Type this:

   ```
   [master@server ~]$ sudo useradd -N -c "Mel Mellow"  mmellow
   ```

4. Use the **id** command to examine the properties of the user mmellow:

   ```
   [master@server ~]$   id  mmellow
   ```

5. Again, use the **passwd** command to create a new password for the account mmellow. Set the password to be *2owl!78*, and repeat the same password when prompted:

   ```
   [master@server ~]$ sudo  passwd  mmellow
   ```

6. Create the final user account, called *bogususer*. But this time, specify the user's shell to be the tcsh shell, and let the user's default primary group be the system "games" group:

```
[master@server ~]$ sudo useradd -s /bin/tcsh -g games bogususer
```

7. Examine the **/etc/passwd** file for the entry for the bogususer user:

```
[master@server ~]$  grep  bogususer  /etc/passwd
bogususer:x:1003:20::/home/bogususer:/bin/tcsh
```

From this entry, you can tell the following: The UID is 1003, the GID is 20, the home directory is **/home/bogususer**, and the user shell is **/bin/tcsh.**

Creating Groups with groupadd

Next, we are going to create a couple of different classes of groups: system and nonsystem (Regular). Follow these steps:

1. Create a new group called research:

```
[master@server ~]$  sudo groupadd   research
```

2. Examine the entry for the research group in the **/etc/group** file:

```
[master@server ~]$   grep  research  /etc/group
research:x:1002:
```

This output shows that the group ID for the research group is 1002.

3. Create another group called sales:

```
[master@server ~]$ sudo   groupadd  sales
```

4. Create the final group called bogus, and force this group to be a system group (that is, the GID will be lower than 999). Type the following:

```
[master@server ~]$ sudo   groupadd -r bogus
```

5. Examine the entry for the bogus group in the **/etc/group** file:

```
[master@server ~]$   grep  bogus  /etc/group
bogus:x:973:
```

The output shows that the group ID for the bogus group is 973.

Modifying User Attributes with usermod

Now try using **usermod** to change the user and group IDs for a couple of accounts.

1. Use the **usermod** command to change the user ID (UID) of the bogususer to 1600:

```
[master@server ~]$  sudo usermod -u 1600  bogususer
```

2. Use the **id** command to view your changes:

```
[master@server ~]$   id bogususer
uid=1600(bogususer) gid=20(games) groups=20(games)
```

3. Use the **usermod** command to change the primary GID of the bogususer account to that of the bogus group (GID = 973) and also to set an expiry date of 12-12-2021 for the account:

```
[master@server ~]$ sudo usermod -g 973  -e  2024-12-12  bogususer
```

4. View your changes with the **id** command:

```
[master@server ~]$   id  bogususer
uid=1600(bogususer) gid=973(bogus) groups=973(bogus)
```

5. Use the **chage** command to view the new account expiration information for the user:

```
[master@server ~]$  sudo chage  -l  bogususer
Account expires                         : Dec 12, 2024
...<OUTPUT TRUNCATED>...
```

TIP On Red Hat–like distros such as Fedora, CentOS, and RHEL, you can grant regular users superuser privileges (root) or sudo access by adding them to the *wheel* group. After proper authentication, this allows a user with sudo privileges to execute a command as the superuser or another user, as specified by the system security policy. For example, to add the master user to the wheel group and thus grant sudo rights, you would run the following:

```
#  usermod master -a -G wheel
```

The check box asking if you want to allow the newly created user to administer the system during the initial operating system installation (Chapter 2) was essentially adding the user to the wheel group!

Modifying Group Attributes with groupmod

Now try using the **groupmod** command:

1. Use the **groupmod** command to rename the bogus group as bogusgroup:

```
[master@server ~]$ sudo    groupmod  -n  bogusgroup  bogus
```

2. Again use the **groupmod** command to change the GID of the newly renamed bogusgroup to 1600:

```
[master@server ~]$ sudo    groupmod  -g  1600  bogusgroup
```

3. View your changes to the bogusgroup in the **/etc/group** file:

```
[master@server ~]$ grep  bogusgroup  /etc/group
```

Deleting Users and Groups with userdel and groupdel

Let's try using the **userdel** and **groupdel** commands to delete users and groups, respectively:

1. Use the **userdel** command to delete the user bogususer that you created previously. At the shell prompt, type this:

```
[master@server ~]$ sudo userdel  -r  bogususer
```

2. Use the **groupdel** command to delete the bogusgroup group:

```
[master@server ~]$ sudo groupdel bogusgroup
```

Notice that the bogusgroup entry in the **/etc/group** file is removed.

NOTE When you run the **userdel** command with only the user's login specified on the command line (for example, **userdel bogususer**), all of the entries in the **/etc/passwd** and **/etc/shadow** files, as well as references in the **/etc/group** file, are automatically removed. But if you use the optional **-r** parameter, all of the files owned by the user in that user's home directory are removed as well.

TIP The virtual machine images used in popular cloud environments, such as Amazon Web Services (AWS), Google Compute Platform (GCP), and Linode, tend to standardized on certain username and/or authentication schemes. For example, AWS often has a user account named *ec2-user* in its VM images, and GCP usually has user accounts named after the GCP login name (google handle) or the VM's operating system name (suse, ubuntu, fedora, and so on). For Microsoft's Azure, you usually have the choice of specifying the username at the time of launching the VM instance. Native FOSS VM bootstrapping tools such as **virt-builder** usually have mechanisms for allowing you to specify or create user accounts (and credentials) at the time of image compilation.

Summary

This chapter discussed the ins and outs of user and group management under GNU/Linux. Much of what you read here also applies to numerous GNU/Linux distributions. which makes administering users in heterogeneous environments much easier.

Among other things, the following main points were covered in this chapter:

- Each user gets a unique UID, and each group gets a unique GID.
- The **/etc/passwd** file maps UIDs to usernames.
- GNU/Linux handles encrypted passwords in multiple ways.
- GNU/Linux includes a multitude of tools to help you administer users.
- Linux can store user and group information in various databases that have a standard format and structure.
- Pluggable Authentication Modules (PAM) is Linux's generic way of handling multiple authentication mechanisms.

These concepts may be pretty foreign to an administrator coming from a Microsoft Windows environment and can be a little tricky at first. Not to worry, though—the GNU/Linux security model is quite straightforward, and as such you will quickly get comfortable with how it all works.

This page intentionally left blank

CHAPTER

7 Booting and Shutting Down

As the complexity in modern-day operating systems has grown, so has the complexity in starting up and shutting down processes. Not only does the hardware need to be initialized and the core operating system need to be brought up and shut down, but an impressive list of services and processes must also be started and stopped at the right moment.

In this chapter, we discuss the bootstrapping of the Linux operating system with GRUB. We then step through the processes of starting up and shutting down the Linux environment. We discuss the scripts that automate parts of these processes, as well as modifications that may sometimes be desirable in the scripts. We finish up with coverage of a few odds and ends that pertain to booting up and shutting down.

CAUTION Apply a liberal dose of common sense in following the practical exercises in this chapter on a real/production system. As you experiment with modifying startup and shutdown scripts, bear in mind that it is possible to bring your system to a nonfunctional state that cannot be recovered by mere rebooting. Don't mess with a production system; if you must, make sure you back up all the files you want to change—and most importantly, have your bootable rescue media handy.

Boot Loaders

For any operating system to boot on standard PC hardware, you need what is called a *boot loader*. The boot loader is the first software program that runs when a computer starts. It is responsible for handing over control of the system to the operating system.

Typically, the boot loader will reside in the Master Boot Record (MBR) of the disk, and it knows how to get the operating system up and running. The de facto boot loader on Linux distros is called GRUB, and it can be configured to also recognize and boot other operating systems. GRUB currently comes in two versions—the older GRUB Legacy and the newer and more common GRUB version 2 (GRUB 2).

GRUB Legacy

Most modern Linux distributions use GRUB as the default boot loader during installation, including Fedora, Red Hat Enterprise Linux (RHEL), openSUSE, Debian, CentOS, Ubuntu, and a host of others. GRUB aims to be compliant with the Multiboot Specification and offers many features.

The GRUB Legacy boot process happens in stages. Each stage is taken care of by special GRUB image files, with each preceding stage helping the next stage along. Two of the stages (stage 1 and 2) are essential, and the other stages (for example, stage 1.5) are optional and dependent on the particular system setup.

On RPM-based distros, you can check your GRUB version by running the following:

```
# rpm -qa | grep -i grub
```

On Debian-based systems, you can check your version of GRUB by running this:

```
# dpkg -l | grep -i grub
```

If the output shows GRUB version 1.9*, 2.*, or higher, then you are running GRUB 2 or newer. If the output shows version 0.99 or earlier, you are running a legacy version of GRUB.

Conventions Used in GRUB

GRUB has its own special way of referring to devices (optical disc drives, floppy drives, hard disk drives, and so on). The device name has to be enclosed in parentheses: ( ). GRUB starts numbering its devices and partitions from *0*, not from *1*. Therefore, GRUB would refer to the first enumerated hard drive on the controller as (hd0), where "hd" means "hard disk" drive and the number 0 means it is the first enumerated hard disk.

NOTE A simple-to-use script that can help complete various GRUB-related tasks or repairs is the **grub-install** script (see `man grub-install`).

Keep in mind that, typically, GRUB needs to be installed only once. Any further modifications are stored in a text configuration file, and such changes don't need to be rewritten to the MBR or partition boot sector every time.

CAUTION Back up the MBR! Before you proceed with the exercises that follow, it is a good idea to make a backup of your current "known good" MBR. It is easy to do this using the `dd` command. Since the MBR of a PC's hard disk (sda, vda, sdb ...) resides in the first 512 bytes of the disk, you can easily copy the first 512 bytes to a file by typing the following:

```
# dd if=/dev/sda of=/tmp/COPY_OF_MBR bs=512 count=1
1+0 records in
1+0 records out
512 bytes (512 B) copied, 0.0261237 s, 19.6 kB/s
```

This command will save the MBR into a file called **COPY_OF_MBR** under the **/tmp** directory.

GRUB 2

GRUB 2 is the successor to the GRUB Legacy boot loader. Some widely used and deployed older versions of popular Linux distros still use and standardize on GRUB Legacy, but many mainstream distros have adopted GRUB 2. It is reasonable to assume that at some point in the near future GRUB Legacy will no longer be supported and everybody will have to move to GRUB 2 or to something else, if something better comes along.

The main features of GRUB 2, as well as some differences when compared with GRUB Legacy, are listed in Table 7-1.

Configuring GRUB 2

The primary configuration file in GRUB 2 is **grub.cfg**. On most modern EFI-based systems, **grub.cfg** is stored on the special EFI boot partition **(/boot/efi)**. On our sample EFI-based Fedora server, the exact location of the file is **/boot/efi/EFI/fedora/grub.cfg**. On other distros such as Ubuntu, the file might be located at **/boot/efi/EFI/ubuntu/grub.cfg**, for example. On traditional BIOS-based systems, the file is usually located at **/boot/grub/grub.cfg** (Ubuntu) or **/boot/grub2/grub.cfg** (Fedora).

GRUB 2 Feature	Description
Configuration files	The primary configuration file for GRUB 2 is now named **grub.cfg**. This is different from GRUB Legacy's configuration file, which is named **menu.lst** .**grub.cfg** is not meant to be edited directly. Its content is automatically generated. Multiple files (scripts) are used for configuring GRUB's menu, and some of these files are stored under the **/etc/grub.d/** directory, such as the following: • **00_header** Sets the default values for some GRUB variables, such as graphics mode, default selection, time-outs, and so on. • **10_linux** Helps to find all the kernels on the root device of the current operating system, and automatically creates associated GRUB entries for all the kernels it finds. • **30_os-prober** Automatically probes for other operating systems that might be installed on the system. This is especially useful in dual-boot systems (Windows running with Linux, for example). • **40_custom** This is where users can edit and store custom menu entries and directives.
Partition numbers	Partition numbers in GRUB 2 device names start at *1*, not *0* (zero). However, the physical device name/numbering convention remains the same as in GRUB Legacy; that is, it still starts from 0. So, for example, a GRUB 2 directive that reads (hd0,1) refers to the first (1) partition on the first drive (hd0). In contrast, the GRUB Legacy equivalent of this would be (hd0,0).
File system	GRUB 2 natively supports many more file systems than GRUB Legacy.
Image files	GRUB 2 no longer uses the Stage 1, Stage 1.5, and Stage 2 files. Most of the functions served by the stage files have been replaced by the **core.img** file, which is generated dynamically from the kernel image and some other modules.

Table 7-1 GRUB 2 Features

Regardless of its location, this file is *automagically* generated and updated by invoking the **grub2-mkconfig** command. The **grub2-mkconfig** command parses the various scripts (see Table 7-1) under the **/etc/grub.d/** directory to create **grub.cfg**. Entries in **grub.cfg** are a good indication for what will end up being displayed in the operating system boot menu.

The following shows an excerpt from a sample **grub.cfg** file on a Fedora-based system (note that line numbers 1–16 have been added to the output to aid readability):

```
1   ### BEGIN /etc/grub.d/10_linux ###
2      menuentry 'Fedora ...' --class fedora
--class gnu-linux --class gnu --class os
--unrestricted $menuentry_id_option 'gnulinux-5.*' {
3         load_video
4         set gfxpayload=keep
5         insmod gzio
6         insmod part_gpt
7         insmod xfs
8         set root='hd0,gpt2'
9         if [ x$feature_platform_search_hint = xy ]; then
10           search --no-floppy --fs-uuid --set=root  0e1b*021d
11         else
12           search --no-floppy --fs-uuid --set=root 0e1b*021d
13         fi
14         linux   /vmlinuz-5.*.x86_64 root=/dev/mapper/fedora-root  ro
          rd.lvm.lv=fedora/root
          rd.lvm.lv=fedora/swap rhgb quiet
15         initrd  /initramfs-5.*.x86_64.img
16 }
```

The entries in the preceding sample configuration file for GRUB are discussed here:

- **Line 1** All lines that begin with a pound sign (#) are comments and are ignored.
- **Line 2: `menuentry`** This directive defines the beginning stanza for a GRUB menu entry/title. In this example, the specific title looks like `'Fedora...'`. This title is what will be displayed in the GRUB menu at system boot. The directive accepts different options, such as class, users, unrestricted, hotkey, and so on. The **`--class`** options can be used to group menu entries into classes.

  The **`menuentry`** line ends with an opening brace ({), which signifies the start of all the directives that pertain to this particular GRUB 2 **`menuentry`**.
- **Line 3: `load_video`** This refers to a function defined earlier in the configuration file that is not shown in our excerpt. This particular function loads several dynamic GRUB video-related modules such as **vga**, **efi_uga**, **ieee1275_fb**, and so on.
- **Line 4: `set`** Sets the environment variable **`gfxpayload`** value to **`keep`**.
- **Lines 5–7: `insmod`** These lines insert the dynamic GRUB modules **gzio**, **part_gpt**, and **xfs**, respectively.
- **Line 8: `set root='hd0,gpt2'`** Sets the GRUB root device name (usually **/boot**). In this example, the root device is defined as the second partition (**gpt2**) on the first hard disk (hd0).

- **Lines 9–13: `if`, `then`, `else`, `fi`** This block makes use of the built-in scripting capabilities of GRUB 2 using conditionals. The block essentially tests the return value of the statements or conditions specified between the starting **`if`** reserved command and the ending **`fi`** command. Here, the GRUB 2 **`search`** command is being executed based on the return value of the tests.
- **Line 14: `linux`** This command loads the specified Linux kernel image file (**/vmlinuz-5.*.x86_64**) in 32-bit mode. Other variations of this are **`linux16`**, **`linuxefi`**, and so on. Specific directives may also be specified in the BootLoaderSpec files stored under **/boot/loader/entries/**. The other arguments after the kernel filename are passed verbatim to the kernel as boot parameters/arguments. One example of a boot parameter is **`rd.lvm.lv`**, which activates the specified logical volumes (LVs). Another example is the **`quiet`** parameter, which disables most of the verbose log messages as the system boots.
- **Line 15: `initrd`** Loads the specified initial ramdisk file for the Linux kernel image to be booted in 32-bit mode. Other variations of this are **`initrd16`**, **`initrdefi`**, and so on. Specific directives may also be specified in the BootLoaderSpec files stored under **/boot/loader/entries/**. The initial ramdisk image provides a way to preload certain basic modules that may be necessary for accessing and mounting the root file system.
- **Line 16: }** This closing brace signifies the end of this particular GRUB 2 menu entry.

NOTE The path names are relative to the **/boot** directory, so, for example, instead of specifying the path to the kernel to be **/boot/vmlinuz-5.*.x86_64**, GRUB's configuration file references this path as **/vmlinuz-5.*.x86_64**.

Initial RAM Disk (initrd)

You might be wondering about the **`initrd`** option found in the GRUB configuration file. It is used for preloading modules or drivers. The initial random access memory (RAM) disk is a special device or an abstraction of RAM. It is initialized by the boot loader before the actual kernel kicks in. One sample problem solved by **`initrd`** happens when a file system module is needed to allow access to the file system in order to load the other necessary modules. For example, your boot partition might be formatted with some exotic file system, such as the B-tree file system (Btrfs) or ReiserFS, for which the kernel has no built-in drivers and whose modules/drivers reside on the disk. This is a classic chicken-and-egg problem! You can't access the file system because you don't have the file system modules, and you can't get to file system modules because they reside on the file system!

The solution is to provide the kernel with a RAM-based structure (image) that contains the necessary loadable modules to get to the rest of the modules. This image is executed and resides in RAM, and as a result it does not need immediate access to the on-disk file system. The word **`initrd`** is sometimes used interchangeably with **`initramfs`**, which uses a slightly different mechanism to solve the same problem. **`initramfs`** is the newer and preferred mechanism of the two, and it's the one you will find in use on most modern Linux-based distributions.

Adding a New GRUB 2 Menu Entry

In this section, you learn how to manually add a new entry to the GRUB 2 boot menu. We will be adding a semi-useless entry just for testing purposes. In other words, the new entry will not do anything useful—we're adding it for illustration purposes only. In Chapter 9, we will use the same concepts to add a more useful GRUB 2 menu entry that will boot the Linux kernel we build from scratch.

In the following steps, you will create a new entry that does nothing more than change the foreground and background colors of GRUB's boot menu:

1. While logged in as the root user, change your current working directory to the /etc/grub/d/ directory:

```
fedora-server:~ #  cd /etc/grub.d/
```

2. Create a new GRUB 2–formatted script named **99_custom.sh** that will be used to add your new entry to the GRUB menu. Using any text editor you are comfortable with (the **vim** editor is used in this example), type the following text in the file:

```
#!/bin/sh -e
echo "Starting new GRUB2 script creation..."
cat << EOF
menuentry "Change Background Color" {
      set  color_normal=white/green
}
EOF
```

3. Once the script has been successfully created, make the script executable:

```
fedora-server:~ #  chmod +x  99_custom.sh
```

4. Run **grub2-mkconfig**, which will parse the commands in the new script you just created and create an appropriate updated **/boot/efi/EFI/fedora/grub.cfg** configuration:

```
fedora-server:~ #  grub2-mkconfig -o /boot/efi/EFI/fedora/grub.cfg
```

   On a BIOS based or non-EFI system, you would instead need to send the output **grub2-mkconfig** to the **/boot/grub2/grub.cfg** (Fedora) or **/boot/grub/grub.cfg** (Ubuntu) file.

5. Use the **grep** command to verify that the new **menuentry** was added to **grub.cfg**:

```
fedora-server:~ # grep -A 4 "99_custom.sh" /boot/efi/EFI/fedora/grub.cfg
```

6. Once you've confirmed the entry is in the **grub.cfg** file, reboot the system.

When the system reboots, you can test your changes by following the next steps while at the initial GRUB screen:

1. After the GRUB menu appears, select the Change Background Color option and press ENTER.

2. The color of the menu should change to the color you specified in the **99_custom.sh** file using the **`color_normal`** directive. Note that your selection will *not* actually boot the system. You will have to select a valid **`menuentry`** to boot the OS into Linux.

TIP If you don't want a specific menu entry to be automatically created in a system using GRUB 2 as the boot loader, you have to delete or disable the corresponding script in the **/etc/grub.d/** directory that creates the entry. For example, if you don't want to see entries for other, non-native operating systems such as Microsoft Windows in your boot menu, you need to delete **/etc/grub.d/30_os-prober** or alternatively make it non-executable and then run the **`grub2-mkconfig`** command to regenerate **grub.cfg**, like so:

```
# chmod -x /etc/grub.d/30_os-prober
# grub2-mkconfig -o /boot/efi/EFI/fedora/grub.cfg
```


USB GRUB 2 Boot Disk

Let's create a bootable external USB GRUB 2 disk. This will allow you to boot the system using the external USB device in the off chance that you can't boot the system normally. Here are some sample situations where a bootable GRUB 2 USB disk can come in handy:

- Your system does not currently have a boot loader installed.
- The boot loader for the system is damaged or corrupted, possibly by another operating system.
- You want to create a multiboot USB medium for storing and booting different operating systems that come in ISO (CD/DVD-ROM) image format.

The general idea behind using a USB GRUB boot disk is that it is assumed that you currently have a system with an unbootable, corrupt, or unwanted boot loader—and since the system cannot be booted by itself from the hard disk, you need another medium with which to bootstrap the system. For this, you can use a GRUB 2 USB disk, a GRUB CD, or even a GRUB floppy disk (good luck with finding a system with a floppy disk drive!).

Assuming we are ready to lose the entire contents of the target USB device and that the current block device for the USB drive is **/dev/sdb**, we can carry out the following procedures:

1. Plug in a reasonably sized USB drive into the appropriate port on the system and then use either the **`fdisk`**, **`lsusb`**, **`dmesg`**, or **`blkid`** command to verify the block device file that Linux has associated with your USB drive. We use the **`blkid`** command here:

```
[root@server ~]#   blkid
/dev/sdb1: UUID="176-ef7-459-a4e6-304" TYPE="ext4
```

2. Format the drive with the **ext4** file system (this will wipe the drive clean):

```
[root@server ~]# mkfs.ext4   /dev/sdb1
```

3. Mount the USB device to an existing and available mount point (for example, **/media/sda1**):

```
[root@server ~]#  mount  /dev/sdb1   /media/sdb1/
```

4. Use the **grub2-install** utility to install the boot loader on the USB device:

```
[root@server ~]# grub2-install  --boot-directory=/media/sdb1/   /dev/sdb
```

5. If you don't want to use GRUB 2 interactively from the GRUB 2 prompt during system boot, you will have to store a suitable GRUB 2 configuration file on the USB device.

   You can alternatively duplicate the exact boot configuration on the current system by copying over the current **grub.cfg** file to the USB device or creating a custom configuration file from scratch.

   Here we will duplicate the GRUB 2 boot configuration on the current system and write the **grub.cfg** file to the USB device mount point:

```
[root@server ~]# grub2-mkconfig   -o  /media/sdb1/grub/grub.cfg
```

6. View the files and directories created on the USB device by the previous steps and then **umount** the USB device when done:

```
[root@server ~]# ls  /media/sdb1/grub/  &&   umount   /dev/sdb1
```

That's it! You can reboot the system at your leisure and try selecting the USB device as the boot medium (adjust your UEFI or BIOS settings if necessary) to make sure that you have indeed created a bootable external USB device.

Bootstrapping

This section assumes you are already familiar with the boot processes of other operating systems and thus already know the boot cycle of your hardware. We cover the process of bootstrapping the operating system. We'll begin with the Linux boot loader (usually GRUB for PCs).

Kernel Loading

Once GRUB has started and you have selected Linux as the operating system to boot, the first thing to get loaded is the kernel. Keep in mind that no operating system exists in memory at this point, and PCs have no easy way to access all of their memory. Thus, the kernel must load completely into the first megabyte of available RAM. To accomplish this, the kernel is compressed. The head of the file contains the code necessary to bring the CPU into protected mode (thereby removing the memory restriction) and decompress the remainder of the kernel.

Kernel Execution

With the kernel in memory, it can begin executing. Something to remember is that the kernel is nothing but a program (albeit a very sophisticated and smart one) that needs to be executed. The kernel knows only whatever functionality is built into it, which means any parts of the kernel compiled as modules are useless at this point. At the very minimum, the kernel must have enough code to set up its virtual memory subsystem and access the root file system (usually the ext3, ext4, Btrfs, or XFS file system). Once the kernel has started, a hardware probe determines what device drivers should be initialized. From here, the kernel can *mount* the root file system. This is akin to what a Microsoft Windows system will do to recognize and access its C drive. The kernel mounts the root file system and starts a program called **init**, which is discussed in the next section.

SysV init Runlevel	systemd Targets	Description
0	poweroff.target, runlevel0.target	Halt the system
1	rescue.target, runlevel1.target	Enter single-user mode
2	runlevel2.target, multi-user.target	Multiuser mode, traditionally without Network File System (NFS)
3	multi-user.target, runlevel3.target	Full multiuser mode (normal)
4	multi-user.target, runlevel4.target	Unused or user defined
5	graphical.target, runlevel5.target	Same as runlevel 3, except using an X Window (graphical) System login rather than a text-based login
6	reboot.target, runlevel6.target	Reboot the system
emergency	emergency.target	Emergency shell

Table 7-2 SysV init Runlevels and systemd Targets

The init Process

On the traditional System V (SysV) style of Linux distros, the **`init`** process is the first non-kernel process that is started; therefore, it always gets the process ID number of 1. **`init`** reads its configuration file, **/etc/inittab**, and determines the *runlevel* where it should start. Essentially, a runlevel dictates the system's behavior. Each level (designated by an integer between 0 and 6) serves a specific purpose. A runlevel of **`initdefault`** is selected if it exists; otherwise, you are prompted to supply a runlevel value.

Most modern Linux distros have substituted the functionality previously provided by SysV **`init`** with a new startup manager called **`systemd`**. The notion of runlevels is different in **`systemd`**, and instead are referred to as *targets*. Chapter 9 discusses **`systemd`** in greater detail. Table 7-2 shows the different runlevels in the traditional SysV world as well as their equivalent in the **`systemd`** world.

Systemd Scripts

On **`systemd`**-based distros, the configuration files that control *what* is started, *how* it is started, and *when* it is started are referred to as unit files. These files are stored in certain locations on the file system. For example, **/usr/lib/systemd/system/** holds any relevant unit files that come with installed packages, and **/etc/systemd/system/** often holds unit files installed (implicitly or explicitly) by the system administrator.

These unit files are collectively responsible for starting, managing, or stopping various services and processes at a fixed point during or after the boot process. In many cases, the order in which these units are executed makes a difference. For example, you can't start services that rely on a configured network interface without first enabling and configuring the network interface!

Let's peer into the **/usr/lib/systemd/system** directory of our sample Fedora server:

```
[master@fedora-server ~]$ ls  -l /usr/lib/systemd/system
-rw-r--r--. 1 root root  521 Dec 17 23:32 network.target
-rw-r--r--. 1 root root  704 Dec 17 23:32 tmp.mount
-rw-r--r--. 1 root root  322 Oct 31 12:36 crond.service
...<OUTPUT TRUNCATED>...
```

As you can see from the preceding output, there are lots of unit files! The files are named **tmp.mount**, **basic.target**, **network.target**, **crond.server**, **blah.socket**, and so on. You will learn more about the relevance of the filename extensions (*.target, *.service, *.mount, and so on) in Chapter 9. Thankfully, not all of the unit files are active all of the time, and they also all don't need to be manually managed by the system administrator. The only unit files that are active are the ones that have been explicitly enabled by the system administrator or that have been implicitly enabled by the system via a software package or system configuration utility.

For any given enabled (active) unit file, a corresponding symbolic link for the unit is created under the unit configuration directory.

Okay, we are beginning to talk in circles, so let's break off into a real example that will hopefully make things clearer. We'll create our very own service (aka **systemd** service unit) in the next section.

Writing Your Own rc Script

In the course of administering a Linux system and keeping it running, at some point you will need to modify the behavior of a process or service as it pertains to the system bootup or shutdown. **systemd** makes this easy to do on modern Linux distros.

Let's step through the process using an example to see how to create a simple service and integrate it with the automatic system bootup process. You can use this example as a skeleton script, by the way, changing it to add anything you need. Let's assume you are running a **systemd**-enabled Linux server and you want to start a special program that pops up a message every hour and reminds you that you need to take a break from the keyboard—a good idea if you don't want to get carpal tunnel syndrome!

The script to start this program will include the following: a description of the script's purpose (so that you don't forget it a year later) and a simple loop to send an e-mail every hour.

NOTE Lines starting with a pound sign (#) are comments and are not part of the script's actions, except for the first line.

Given these parameters, let's begin creating the script.

Creating the carpald.sh Script

First, we'll create the script that will perform the actual function we want. The script is unsophisticated, but it will serve our purpose here. A description of what the script does is embedded in its comment fields.

1. Launch any text editor of your choice and type the following text:

```
#!/bin/sh
#
#Description: This simple script will send a mail to any e-mail address
#specified in ADDR variable every hour, reminding the user to take a
#break from the computer to avoid the carpal tunnel syndrome. The script
#has such little intelligence that it will always send an e-mail as long
#as the system is up and running - even when the user is fast asleep!!
#So don't forget to disable it after the fact.
#Author: w.s.
#
set -e
ADDR=root@localhost
while true
do
         sleep 1h
         echo "Get up and take a break NOW !!" | \
         mailx -s "Carpal Tunnel Warning" $ADDR
done
```

2. Save the text of the script into a file called **carpald.sh**.
3. You next need to make the script executable. Type the following:

```
[master@fedora-server ~]$   chmod 755 carpald.sh
```

4. Copy or move the script over to a common and central directory location; we'll use the **/usr/local/bin/** directory:

```
[master@fedora-server ~]$ sudo  mv  carpald.sh  /usr/local/bin/
```

5. Notice that the **carpald.sh** script references a **`mailx`** command (second-to-last line). Therefore, you need to have the **`mailx`** utility available or installed on the system so it can be used within the script.

   On our sample Fedora/CentOS/RHEL system, you can use **`dnf`** to install **`mailx`**—if you don't already have it installed. Type the following:

```
[master@fedora-server ~]$ sudo dnf -y --nogpgcheck install mailx
```

   If you don't have **`mailx`** on a Debian-based system like Ubuntu, you can use the **`apt`** program to install it by running this:

```
master@ubuntu-server:~$ sudo  apt -y install mailutils
```

NOTE Our demo **carpald.sh** script will need a mechanism to actually deliver e-mails to the recipient e-mail address (root@localhost). This involves either setting up a local mail server or configuring the system to use an external mail server or some other mechanism to send e-mails. Because this is just a proof of concept, we are not ready to go *whole hog* just yet. So, for now, kindly ignore any "message not sent" errors that you might see in the system logs! Chapter 20 covers installing and setting up a fully functioning mail server. If you don't trust us or simply can't wait to test the script, you can quickly install the **`postfix`** package using **`dnf`**, **`yum`**, or **`apt`**.

Creating the carpald Service Unit

Here you will create the actual startup script that's executed during system startup and shutdown. The file you create here is called **carpald**. The file is **`systemd`**-enabled, which means that it will make use of **`systemd`** domain-specific language/syntax. This is a useful and time-saving functionality.

1. Launch any text editor of your choice and type the following text:

```
[Unit]
Description=The Carpal Notice Daemon
After=syslog.target
[Service]
ExecStart=/usr/local/bin/carpald.sh
Type=simple
[Install]
WantedBy=multi-user.target
```

   A few comments about the preceding sample **carpald.service** unit file. The fields in square brackets, like **[Unit]**, **[Service]**, and **[Install]**, mark the start of a new section within the file:

   - The [**Unit**] section contains generic parameters affect various **`systemd`** objects, such as **Description**, **After**, and so on. Here, we specify a simple string that describes our custom carpald service, and we also specify that this unit can be started after the **syslog.target**.
   - The [**Service**] section contains parameters that are more specific to the particular unit. For one thing, it specifies that this is a service type of unit. The **`ExecStart`** parameter is set to the path of the **carpald.sh** shell script that we created earlier.

     The **`Type`** parameter specifies how the service should notify the init system, after starting up. Some sample values for **`Type`** are **simple**, **forking**, and **exec dbus**.
   - Finally, the [**Install**] section contains directives that specify the circumstances (aka triggers) under which this unit is started. The value of **`WantedBy`** is set to **multi-user.target** here, meaning that our **carpald.service** unit will be automatically started whenever the **multi-user.target** (or runlevel) is requested.

2. Save the text of the script into a file called **carpald.service**.

3. Copy or move the script over to the **/etc/systemd/system/** directory where **systemd** will find it:

```
[master@fedora-server ~]$ sudo mv -f carpald.service  /etc/systemd/system
```

4. Now tell **systemd** about the existence of this new unit file by reloading the **systemd** daemon:

```
[master@fedora-server ~]$ sudo systemctl daemon-reload
```

This might all appear rather elaborate, but the good news is that you only need to set this up once and then pretty much forget about it. As long as the script/daemon/service/process is doing what it is designed to do, and when it is supposed to, there should be no need to babysit it. Specifically, in our example the script will automatically run during startup and shutdown—thanks to **systemd**. The overhead upfront is well worth the long-term benefits of avoiding carpal tunnel syndrome!

We have set up the "nuts and bolts" around the **carpald** infrastructure, but it's not a terribly useful until we actually get it to run. We'll use the **service** command to manually start the daemon now:

1. Use the **service** command to view the status of the **carpald.sh** program:

```
[master@fedora-server ~]$  sudo service carpald status
```

2. We don't want to wait for the next system reboot to test the **carpald** script, so let's manually start the **carpald** program using the **service** command:

```
[master@fedora-server ~]$  sudo  service carpald start
```

TIP As long as the e-mail subsystem of the server is running (see Chapter 20), you should see a mail message from the **carpald.sh** script after about an hour. You can use the **mail** program from the command line by typing the following:

```
server:~ # mail
Mail Version **.   Type ? for help.
"/var/mail/root": 1 message 1 new
>N  1 root   Sun Aug 23 18:55   19/589   "Carpal Tunnel Warning"
&
```

Type **q** at the ampersand (&) prompt to quit the **mail** program.

3. Now stop the **carpald** service unit by typing the following:

```
[master@fedora-server ~]$  sudo service carpald stop
```

You can similarly use the native **systemctl** command to stop the **carpald** daemon by running the following:

```
[master@fedora-server ~]$  sudo systemctl stop  carpald.service
```

Enabling and Disabling Services

At times, you might find that you simply don't need a particular service to be started at boot time. This is especially important if you are configuring the system as a server and need only specific services and nothing more. Depending on the specific Linux distro, you have various tools (**chkconfig**, **systemctl**, **rc-**, and so on) at your disposal for enabling or disabling services. In the following section, we'll use the newer Swiss army knife–like **systemctl** utility, which is found on most modern Linux distros running **systemd**.

Enabling a Service

Let's configure carpald service to automatically always start up as part of the system boot process,

1. Once the specified **systemd** target (**multi-user.target**) is reached. Type the following:

```
[master@fedora-server ~]$  sudo systemctl enable carpald.service
Created symlink
/etc/systemd/system/multi-user.target.wants/carpald.service
/etc/systemd/system/carpald.service.
```

   This automatically creates a symbolic link for the **carpald.service** unit under the folder of the target specified in the **[Install]** section in the **WantedBy** parameter. The target folder in this example is **/etc/systemd/system/multi-user.target.wants/**.

2. Use **systemctl** to view whether the **carpald** service is configured to automatically start in the current environment by typing this:

```
[master@fedora-server ~]$  sudo systemctl is-enabled carpald.service
enabled
```

Disabling a Service

To disable a service or **systemd** unit completely, you must, at a minimum, know the name of the service. You can then use the **systemctl** tool to turn it off, thereby preventing it from starting in all targets (runlevels).

1. To disable our "wrist-saving" **carpald.sh** program from automatic startup, you could type this:

```
[master@fedora-server ~]$  sudo  systemctl disable carpald.service
```

2. Check **carpald.service** unit again to see its new startup state:

```
[master@fedora-server ~]$  sudo systemctl is-enabled carpald.service
```

3. To remove the **carpald.sh** program permanently from under **systemd**'s knowledge and control, stop it, delete its unit file, and then reload **systemd**. Enter the following:

```
[master@fedora-server ~]$ sudo systemctl stop  carpald.service
[master@fedora-server ~]$ sudo rm -f  /etc/systemd/system/carpald.service
[master@fedora-server ~]$ sudo systemctl daemon-reload
```

4. We are done with our sample **carpald.sh** script, and we can delete it from the system for good by typing this:

```
[master@fedora-server ~]$ sudo  rm -f /usr/local/bin/carpald.sh
```

And that's how services start up and shut down automatically in Linux. Now go take a break.

Odds and Ends of Booting and Shutting Down

Most Linux administrators do not like to shut down their Linux servers. It spoils their "uptime," which is a thing of pride for Linux system admins. Thus, when a Linux box has to be rebooted, it is usually for unavoidable reasons. Perhaps something bad has happened or the kernel has been upgraded.

Thankfully, Linux does an excellent job of self-recovery, even during reboots. It is rare to have to deal with a system that will not boot correctly, but that is not to say that it will never happen—and that's what this section is all about.

fsck!

Making sure that data on a system's hard disk is in a consistent state is an important function. This function is partly controlled by various subsystems/programs and another file called the **/etc/fstab** file. The File System Check (**`fsck`**) tool is automatically run as necessary on every boot, as specified by certain flags in the **/etc/fstab** file. The specific flag is the **`passno`** flag in the sixth field of the each entry in **/etc/fstab** (see **man fstab**). The purpose of the **`fsck`** program is similar to that of Windows ScanDisk/chkdsk: to check and repair any damage on the file system before continuing the boot process. Because of its critical nature, **`fsck`** is traditionally scheduled to run very early in the boot sequence. On **`systemd`**-enabled systems, the checks are managed by the **`systemd-fsck@.service`** and the **`systemd-fsck-root.service`** unit files.

If the file system is found to be in an unclean state or if a certain number of times has passed in between checks, the system will automatically initiate a file system check at the next startup. Of course, you can always manually initiate as an **`fsck`** check as necessary (and it isn't uncommon for the system administrator to be cursing throughout the process).

If **`fsck`** does need to run, don't panic. It is unlikely you'll have any problems. However, if something does arise, **`fsck`** will prompt you with information about the problem and ask whether you want to perform a repair. In general, you'll find that answering "yes" is the right thing to do.

Virtually all modern Linux distributions use what is called a "journaling file system," and this makes it easier and quicker to recover from any file system inconsistencies that might arise from unclean shutdowns and other minor software errors. Examples of file systems with this journaling capability are ext4, Btrfs, ext3, ReiserFS, JFS, and XFS.

If your storage partitions or volumes are formatted with any of the journaling-capable file systems (such as XFS, ext4, ext3, Btrfs, or ReiserFS), you will notice that recovering from unclean system resets will be much quicker and easier. The only trade-off with running a journaled file system is the slight overhead involved in keeping the journal, and even this depends on the method by which the file system implements its journaling.

Booting into Single-User ("Recovery") Mode

Under Windows, the concept of "Recovery Mode" was borrowed from a long-time UNIX feature of booting into single-user mode. What this means for you in the Linux world is that if something gets broken in the startup scripts that affect the booting process of a host, it is possible for you to boot into this mode, fix the problem, and then allow the system to boot into complete multiuser mode (normal behavior).

If you are using the GRUB 2 boot loader, these are the steps:

1. Select the GRUB entry that you want to boot from the GRUB menu. The entry for the default or most recently installed kernel version will be highlighted by default in the boot menu. Press the E key.
2. You will next be presented with a submenu with various directives (directives from the file **/boot/grub/grub.cfg**, **/boot/grub2/grub.cfg**, **/boot/efi/EFI/fedora/grub.cfg**, **/boot/efi/EFI/ubuntu/grub.cfg**, and so on).
3. Use the ARROW keys on your keyboard to navigate to the line that specifies the Linux kernel to be loaded as well as the parameters to pass to the kernel. On our sample Fedora system, this is the line that begins with **`linux`** or **`linux16`**. Go to the end of that line, leave a space, and then add the keyword **`single`** (or the letter **`s`**) to the end of the line.

TIP Red Hat–based systems like Fedora accept the **`rd.break`** kernel command-line parameter, which can be used for password recovery purposes.

4. Press CTRL-X to boot the system with the temporary changes you just made. The system should boot the kernel into single-user mode.
5. Depending on the system configuration, you will either be prompted for the root password or simply be given a shell prompt. If prompted for a password, type the root password and press ENTER, and you will get the shell prompt.
6. You should now effectively be in the systemd **`rescue.target`** mode (aka runlevel 1). In this mode, you'll find that almost all the services that are normally started are not running. This may include network configuration. So if you need to change the IP address, gateway, netmask, or any network-related configuration file, you can. This is also a good time to run **`fsck`** manually on any partitions that could not be automatically checked and recovered. (The **`fsck`** program will tell you which partitions are misbehaving, if any.)

TIP In the single-user mode of many Linux distributions, only the root partition will be automatically mounted for you. If you need to access any other partitions, you will need to consult **/etc/fstab** and mount them yourself using the **`mount`** command.

7. Once you have made any changes you need to make, simply press CTRL-D. This will exit single-user mode and continue with the booting process, or you can just issue the **`reboot`** command to reboot the system.

Summary

This chapter looked at the various core concepts involved with starting up and shutting down a typical Linux system. We started our exploration with the almighty boot loader. We looked at GRUB in particular as a sample boot loader/manager, because it is the boot loader of choice among the popular Linux distributions. Next, we explored how things (or services) typically get started and stopped in GNU/Linux, and how Linux decides what to start and stop, and at which runlevel (or *target* in **`systemd`** parlance) it is supposed to do this. We even wrote a little shell program, as a demonstration, that helps us to avoid carpal tunnel syndrome. We then went ahead and configured the system to start up the program automatically during system startup!

This page intentionally left blank

CHAPTER

8 File Systems

File systems provide a means of organizing data on a storage medium. They serve as a nice abstraction layer above the nitty-gritty details of sectors, cylinders, and integrated circuits (ICs) of physical disks. This chapter discusses the composition and management of these abstraction layers supported by Linux. We'll pay particular attention to the native Linux file systems—the extended file system family.

This chapter will also cover the many aspects of managing disks. This includes creating partitions and volumes, establishing file systems, automating the process by which the file systems are mounted at boot time, and dealing with them when things go wrong. We will also touch on Logical Volume Management (LVM) concepts.

NOTE Before beginning your study of this chapter, you should be familiar with files, directories, permissions, and ownership in the Linux environment. If you haven't yet read Chapter 4, you should read that chapter before continuing.

The Makeup of File Systems

Let's begin by going over the structure of file systems under Linux to build a proper foundation for the various concepts discussed later on.

i-Nodes

The most fundamental building block of many Linux/UNIX file systems is the *i-node*. An i-node is a control structure that points either to other i-nodes or to data blocks.

The control information in the i-node includes the file's owner, permissions, size, time of last access, creation time, group ID, and other information. The i-node *does not* provide the file's name, however. Directories themselves are special instances of files. This means each directory gets an i-node, and the i-node points to data blocks containing information (filenames and i-nodes) about the files in the directory. Figure 8-1 illustrates the organization of i-nodes and data blocks in the older ext2 file system.

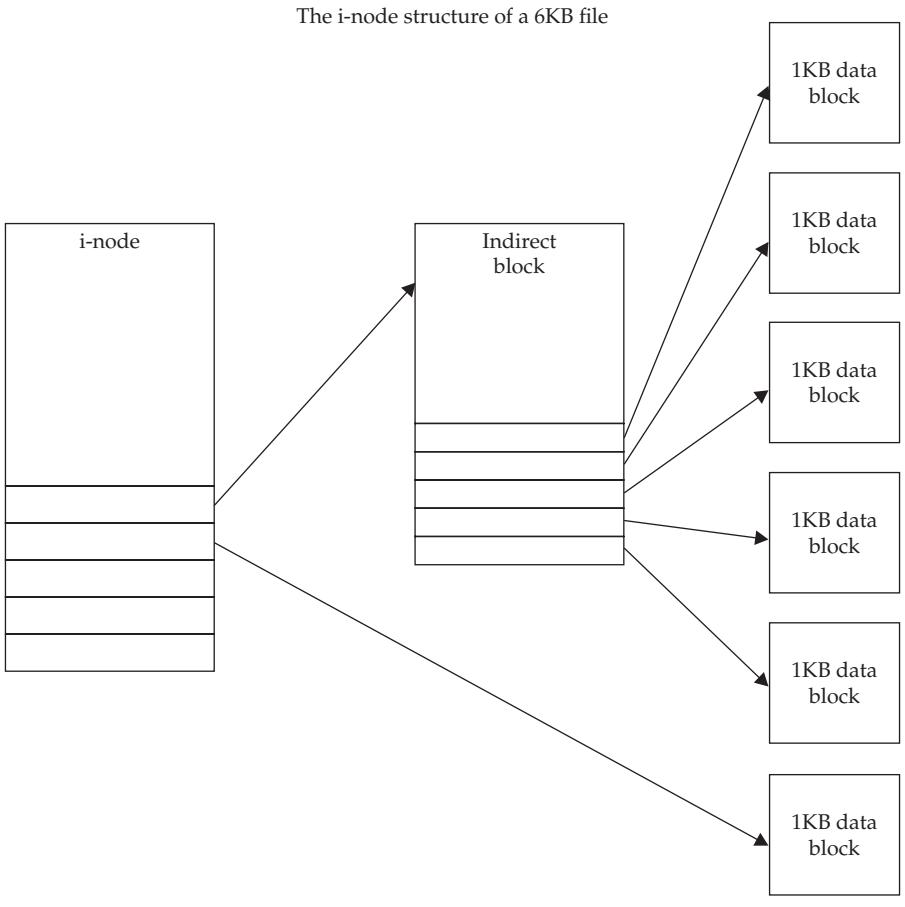


Figure 8-1 The i-nodes and data blocks in the ext2 file system

As you can see in Figure 8-1, the i-nodes are used to provide *indirection* so that more data blocks can be pointed to—which is why each i-node does not contain the filename. Only one i-node works as a representative for the entire file; thus, it would be a waste of space if every i-node contained filename information. Take, for example, a 6GB disk that contains 1,079,304 i-nodes. If every i-node also required 256 bytes to store the filename, a total of about 33MB would be wasted in storing filenames, even if they weren't being used!

Each indirect block, in turn, can point to other indirect blocks if necessary. With up to three layers of indirection, it is possible to store very large files on a Linux file system.

Blocks

Data on an ext* file system is organized into blocks. A block is a sequence of bits or bytes, and it is the smallest addressable unit in a storage device. Depending on the block size, a block might contain only a part of a single file or an entire file. Blocks are in turn grouped into block groups.

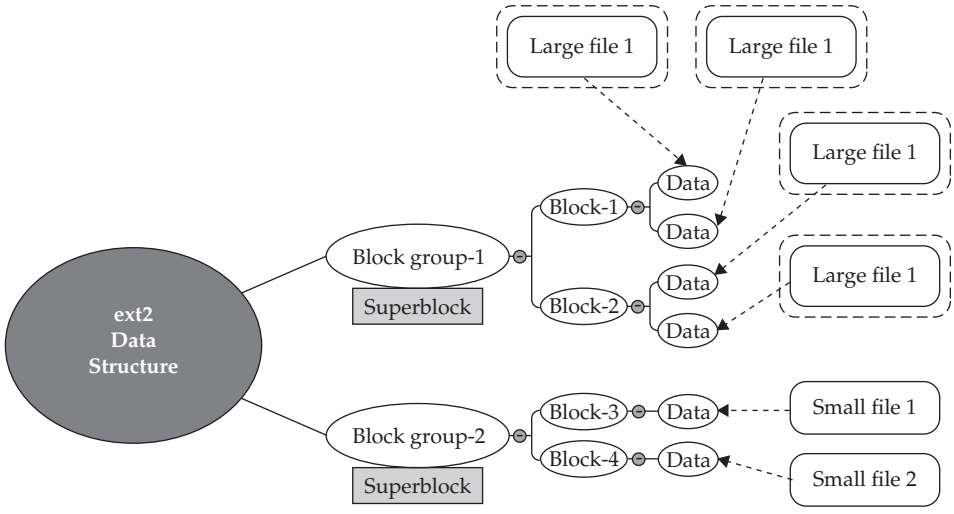


Figure 8-2 Data structure on ext2 file systems

Among other things, the block group contains a copy of the superblock, the block group descriptor table, the block bitmap, an i-node table, and of course the actual data blocks. The relationship among the different structures in an ext2 file system is shown in Figure 8-2.

Superblocks

The first piece of information read from a disk is its *superblock*. This small data structure reveals several key pieces of information, including the disk's geometry, the amount of available space, and, most importantly, the location of the first i-node. Without a superblock, an on-disk file system is useless.

Something as important as the superblock is not left to chance. Multiple copies of this data structure are scattered all over the disk to provide backup in case the first one is damaged. Under Linux's ext2 file system, a superblock is placed after every *group* of blocks, and it contains i-nodes and data. One group consists of 8192 blocks; thus, the first redundant superblock is at 8193, the second at 16,385, and so on.

ext4

The fourth extended file system (ext4) is the successor of the ext2/ext3 file system. It is the default file system used in most Linux distributions. The ext4 file system offers several improvements and features, discussed next.

Journals

Journaling file systems work by first creating an entry of sorts in a log (or journal) of changes that are about to be made before actually committing the changes to disk. Once this transaction has been committed to disk, the file system goes ahead and modifies the actual data or metadata. This results in an all-or-nothing situation—that is, either all or none of the file system

changes get done. Traditional file systems (such as ext2) must instead search through the directory structure, find the right place on disk to lay out the data, and then lay out the data. (Linux can also cache the whole process, including the directory updates, thereby making the process appear faster to the user.)

One of the benefits of using a journaling-type file system is the greater assurance that data integrity will be preserved, and in the unavoidable situations where problems arise, speed, ease of recovery, and likelihood of success are vastly increased. One such unavoidable situation is a system crash. In this case, you might not need to run the file system checker or file system consistency checker (**`fsck`**). Other benefits of using journaling-type file systems are that system reboots are simplified, disk fragmentation is reduced, and I/O operations can be accelerated (depending on the journaling method used).

Btrfs, XFS, and ext4 are popular Linux file systems that implement journaling.

Extents

Unlike ext3/ext2, the ext4 file system does not use the indirect block mapping approach. Instead, it uses the concept of *extents*. An extent is a way of representing contiguous physical blocks of storage on a file system. An extent provides information about the range or magnitude over which a data file extends on the physical storage. So instead of each block carrying a marker to indicate the data file to which it belongs, a single extent (or a few extents) can be used to state that the next *X* number of blocks belong to a specific data file.

Online Defragmentation

As data grows, shrinks, and is moved around, it can become fragmented with time. Fragmentation can cause the mechanical components of a physical storage device to work harder than necessary, which in turn leads to increased wear and tear on the device.

Traditionally, the process of undoing file fragmentation is to defragment the file system offline. "Offline" in this instance means to run the defragmenting when no possibility exists that the files are being accessed or used. ext4 supports online defragmentation of individual files or an entire file system.

Larger File System and File Size

The older ext3 file system is able to support maximum file system sizes of 16TB as well as maximum individual file sizes of up to 2TB. The ext4 system, on the other hand, is able to support maximum file system sizes of 1EB (exabyte) as well as maximum individual file sizes of up to 16TB each.

Btrfs

The B-tree file system (Btrfs) is a next-generation Linux file system aimed at solving any enterprise scalability issues that the current Linux file systems may have. Btrfs is fondly pronounced "Butter FS." As of this writing, Btrfs is available for use in different Linux distributions. In addition to all the advanced features supported by ext4, Btrfs supports (or plans to support) several additional features, including the following:

- Dynamic i-node allocation and transparent compression
- Online file system checking (**`fsck`**-ing)

- Built-in RAID functions such as mirroring and stripping
- Online defragmentation, support for snapshots, and support for sub-volumes
- Support for the online addition and removal of block devices
- Improved storage utilization via support for data deduplication

XFS

XFS is a journaled 64-bit file system that's been around for a while. It has since been ported to the Linux kernel. XFS was recently introduced as the default file system on some Red Hat–based distros, like RHEL and CentOS. XFS is considered a *Big Iron* file system, meaning it is enterprise grade, high performance, reliable, scalable, well tested, and so on. Its features include:

- Quick recovery and support for extended attributes
- Support for file systems as large as 8 exbibytes (8EiB or ~8 million terabytes)
- Support for a maximum file size of 8 exbibytes (8EiB or ~8 million terabytes)
- Support for as close to raw input/output performance that the underlying hardware can provide
- Online defragmentation and online resizing

TIP With so many choices for file systems, it might be a little daunting figuring out which one to go with for which use case or work load! No need to fear, though, because during the server OS installation process, you will find that the default file system supplied by the distribution vendor will suffice for most general use cases, so you can go about your merry business without giving it another thought.

Managing File Systems

Once the file systems have been created, deployed, and added to the backup cycle, they do tend to take care of themselves for the most part. What makes them tricky to manage are the administrative issues, such as users who refuse to do housekeeping on their personal home directories and other cumbersome nontechnical issues.

In the following sections, we'll go over the technical issues involved in managing file systems—that is, the process of mounting and unmounting partitions, dealing with the **/etc/fstab** file, and performing file system recovery with the **`fsck`** tool.

Mounting and Unmounting Local Disks

Partitions or volumes need to be mounted so that their contents can be accessed. In actuality, the file system on a partition or volume is mounted so that it appears as just another subdirectory on the system. This helps to promote the illusion of one large directory tree structure, even though several different file systems might be in use. This characteristic is especially helpful to the administrator, who can relocate data stored on a physical partition to a new location (possibly a different partition) under the directory tree, with the system users being none the wiser.

The file system management process begins with the root directory. This partition is also fondly called *slash* and likewise symbolized by a forward slash character (/). The partition containing the kernel and core directory structure is mounted at boot time. It is possible and usual for the physical partition that houses the Linux kernel to be stored on a separate file system, such as **/boot**. It is also possible for the root file system (/) to house both the kernel and other required utilities and configuration files needed to bring the system up to single-user mode.

As the boot scripts run, additional file systems are mounted, adding to the structure of the root file system. The mount process overlays a single subdirectory with the directory tree of the partition it is trying to mount. For example, let's say that **/dev/sda2** is the root partition. It includes the directory **/usr**, which contains no files. The partition **/dev/sda3** contains all the files that you want in **/usr**, so you mount **/dev/sda3** to the directory **/usr**. Users can now simply change directories to **/usr** to see all the files from that partition. The user doesn't need to know that **/usr** is actually a separate partition.

NOTE In this and other chapters, we might inadvertently say that a partition or volume is being mounted at such and such a directory. Please note that it is actually the file system on the partition that is being mounted.

Keep in mind that when a new directory is mounted, the **`mount`** process hides all the contents of the previously mounted directory. So in our **/usr** example, if the root partition did have files in **/usr** before mounting **/dev/sda3**, those **/usr** files would no longer be visible. (They're not erased, of course, because once **/dev/sda3** is unmounted, the **/usr** files would become visible again.)

Using the mount Command

Like many command-line tools, the **`mount`** command has a plethora of options, most of which you won't be using in daily work. You can get full details on these options from the **`mount`** man page. In this section, we'll explore the most common uses of the command.

The structure of the **`mount`** command is as follows:

```
mount [options] device directory
```

The **`mount`** ***`options`*** can be any of those shown in Table 8-1.

Issuing the **`mount`** command without any options will list all the currently mounted file systems:

```
[master@fedora-server ~]$  mount
/dev/mapper/fedora-root on / type xfs (rw,relatime,attr2,noquota)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
...<OUTPUT TRUNCATED>...
tmpfs on /tmp type tmpfs (rw,nosuid,nodev)
```

Assuming that a directory named **/bogus-directory** exists, the following **`mount`** command will mount the **/dev/sda3** partition onto the **/bogus-directory** directory in read-only mode:

```
[master@fedora-server ~]$ sudo  mount -o ro /dev/sda3 /bogus-directory
```

Option	Description
`-a`	Mounts all the file systems listed in **/etc/fstab** (this file is examined later in this section).
`-t fstype`	Specifies the type of file system being mounted. Linux has support for file systems such as ext2, ext3, ext4, Btrfs, XFS, File Allocation Table (FAT), Virtual File Allocation Table (VFAT), NTFS, ReiserFS, and numerous others. The **`mount`** command usually detects the correct file system on its own.
`remount`	The **`remount`** option is used for remounting already-mounted file systems. It is commonly used for changing the **`mount`** flags for a file system. For example, it can be used for changing a file system that is mounted as read-only into a writable file system, without unmounting it.
`-o options`	Specifies options applying to this **`mount`** process that are specific to the file system type (options for mounting network file systems may not apply to mounting local file systems). Some of the more often used options are read-only (ro), read-write (rw), noatime, noauto, nosuid, and so on.

Table 8-1 Options Available for the mount Command

Unmounting File Systems

To unmount a file system, use the **`umount`** command (note that the command is *not* **`unmount`** with an *n*). Here's the syntax for the command:

```
umount [-f] directory
```

Here, ***directory*** is the directory to be unmounted. Here's an example:

```
[master@fedora-server ~]$ sudo  umount  /bogus-directory
```

This unmounts the partition mounted on the **/bogus-directory** directory.

When the File System Is in Use There's a catch to using **`umount`**: If the file system is in use (that is, someone or something is currently accessing the contents of the file system via reading or writing), you won't be able to unmount that file system. To get around this, you can do any of the following:

- You can use the **`lsof`** or **`fuser`** program to determine which processes are keeping the files open and then kill them off or ask the process owners to stop what they're doing. If you choose to kill the processes, make sure you understand the repercussions of doing so—in other words, be *extra* careful before killing unfamiliar processes.
- You can use the **`-f`** option with **`umount`** to force the unmount process. It is especially useful for Network File System–type file systems that are no longer available.

- Use the lazy **unmount**, specified with the **-l** option. This option almost always works, even when others fail. It detaches the file system from the file system hierarchy immediately, and it cleans up all references to the file system as soon as the file system stops being busy.
- The safest and most proper alternative is to bring the system down to single-user mode and then unmount the file system or initiate a simple reboot. In reality, of course, you don't always have the luxury of being able to do this on production systems.

The /etc/fstab File

The **/etc/fstab** file is a configuration file that **mount** can use. This file contains a list of all partitions known to the system. During the boot process, this list is read and the items in it are automatically mounted with the options specified therein.

Here's the format of entries in a sample **/etc/fstab** file:

```
/dev/device  /dir/to/mount   fstype   Parameters   fs_freq   fs_passno
```

Following are sample entries from an **/etc/fstab** file (line numbers have been added to the output to aid readability):

```
1. /dev/mapper/fedora-root   /         xfs    defaults     0 0
2. UUID=24f22094-3178-4eca   /boot     ext4    defaults    1 2
3. /dev/mapper/fedora-swap    swap     swap    defaults    0 0
```

Let's take a look at some of the entries in the **/etc/fstab** file that haven't yet been discussed.

Line 1 The first entry in our sample **/etc/fstab** file is the entry for the root volume. The first column shows the device that houses the file system—the **/dev/mapper/fedora-root** logical volume (more on volumes later in the section "Volume Management").

The second column shows the mount point—the / (slash or root) directory.

The third column shows the file system type—**XFS** in this case.

The fourth column shows the options with which the file system should be mounted—only the default options are required in this case.

The fifth field is used by the **dump** utility (a simple backup tool) to determine which file systems need to be backed up. And the sixth and final field is used by the **fsck** program to determine whether the file system needs to be checked and also to determine the order in which the checks are done.

Line 2 The next entry in our sample file is the **/boot** mount point. The first field of this entry shows the device—in this case, it points to the device identified by its Universally Unique Identifier (UUID). In the case of the **/boot** mount point, you might notice that the field for the device looks a little different from the usual **/dev/<path-to-device>** convention. The use of a UUID to identify devices/partitions helps to ensure that they are correctly and uniquely identified under any circumstances—such as when a new disk is added or an existing disk is removed or when changing the drive controller or bus to which the drive is attached, and so on.

Some Linux distributions may instead opt to use labels to identify the physical device in the first field of the **/etc/fstab** file. The use of labels helps to hide the actual device (partition)

from which the file system is being mounted. When labels are used, the device is replaced with a token that looks like the following: **LABEL=/boot**. During the initial installation, the partitioning program of the installer automatically set the label on the partition. Upon bootup, the system scans the partition tables and looks for these labels and does the right thing. Labels are also useful for transient external media such as flash drives, USB hard drives, and so on.

The other fields mean basically the same thing as the field for the root mount point discussed previously.

TIP The command-line utility **blkid** can be used to display different attributes of the storage devices attached to a system. One such attribute is the UUID of the volumes. For example, running blkid without any options will print a variety of information, including the UUID of each block device on the system:

```
# blkid
/dev/vda1: UUID="a7c00b-40ee" TYPE="ext4" PARTUUID="69dab7-01"
/dev/vda2: UUID="tICG-1oRC" TYPE="LVM2_member" PARTUUID="6dab7-02"
```


Line 3 This is the entry for the system swap partition, where virtual memory resides. In Linux, the virtual memory can be kept on a separate partition from the root partition. Keeping the swap space on a separate partition helps to improve performance. Also, because the swap partition doesn't need to be backed up or checked with **fsck** at boot time, the last two parameters in its entry are zeros. See the man page on **mkswap** for additional information.

Using fsck

The `fsck` tool (short for File System Check) is used to diagnose and repair file systems that might have become damaged in the course of daily operations. Such repairs may be necessary after a system crash in which the system did not get a chance to fully flush all of its internal buffers to disk. (The fact that this tool's name—**fsck**—bears a striking resemblance to one of the expressions often uttered by system administrators after a system crash, coupled with the fact that this tool can be used as a part of the recovery process, is *strictly* coincidental!)

Usually, the system runs the **fsck** tool automatically during the boot process as it deems necessary. If it detects a file system that was not cleanly unmounted, it runs the utility. A file system check will also be run once the system detects that a check has not been performed after a predetermined threshold, such as a number of mounts or an amount of time passed between mounts. Linux will do its best to automatically repair any problems it runs across! The robust nature of the Linux file systems helps in dire situations. However, when things get out of hand, you might get this message:

```
*** An error occurred during the file system check.
*** Dropping you to a shell; the system will reboot
*** when you leave the shell.
```

At this point, you need to run **fsck** by hand and answer its prompts yourself.

If you do find that a file system is not behaving as it should (spurious errors in log messages are an excellent hint of this type of anomaly), you may want to run **fsck** yourself on a running system. The only downside is that the file system in question must be unmounted in order for this to work, which might sometimes require the system to be offline.

The name **fsck** isn't the actual title for the repair tool; it's actually just a wrapper. The **fsck** wrapper tries to determine what kind of file system needs to be repaired and then runs the appropriate repair tool, passing any parameters that were passed to **fsck**. For the ext4 file system, the actual tool is **fsck.ext4**; for the VFAT file system, the tool is **fsck.vfat**; and for a XFS file system, the utility is called **fsck.xfs**. So, for example, when a system crash occurs on an ext4-formatted partition, you might choose to call **fsck.ext4** directly rather than relying on the wrapper tool **fsck** to call it for you automatically.

To run **fsck** on the **/dev/mapper/fedora-home** file system mounted at the **/home** directory, you would carry out the following steps.

Assuming that the **/home** file system is not currently being used or accessed by any process or user, first unmount the file system:

```
[master@fedora-server ~]$ sudo umount  /home
```

Since we know that this particular file system type is ext4, we can call the correct utility (**fsck.ext4**) directly or simply use the **fsck** utility:

```
[master@fedora-server ~]$ sudo fsck  /dev/mapper/fedora-home
e2fsck 1.* (17-May-2028)
/dev/mapper/fedora-home: clean, 246/3203072 files, 257426/12804096 blocks
```

This output shows that the file system is marked clean.

To forcefully check the file system and answer yes to all questions in spite of what your OS thinks, type this:

```
[master@fedora-server ~]$ sudo fsck.ext4 -f -y /dev/mapper/fedora-home
```

What If I Still Get Errors?

First, relax.

The **fsck** utility rarely finds problems that it cannot correct by itself. When it does ask for human intervention, telling **fsck** to execute its default suggestion is often enough. Very rarely does a single pass of **fsck** not clear up all problems.

On the rare occasions when a second run is needed, it *should not* turn up any more errors. If it does, you are most likely facing a hardware failure. Remember to start with the obvious: Check for reliable power and well-connected and good quality cables; for mechanical drives, make sure that there are no clicking sounds; and so on.

And when all else fails, and **fsck** doesn't want to fix the issue, it will often give you a *hint* as to what's wrong. You can then use this hint to perform a search on the Internet and see what other people have done to resolve the same issue.

The lost+found Directory

Another rare situation occurs when **fsck** finds file segments that it cannot rejoin with the original file. In those cases, it will place the fragment in the partition's **lost+found** directory. This directory is located where the partition is mounted, so if **/dev/mapper/fedora-home** is mounted on **/home**, for example, then **/home/lost+found** correlates to the **lost+found**

directory for that particular file system. Anything can go into a **lost+found** directory—file fragments, directories, and even special files. At the very least, **lost+found** tells you whether anything became dislocated. Again, such errors are extraordinarily rare.

Adding a New Disk

On systems sporting a PC hardware architecture, the process of adding a disk under Linux is relatively easy. Assuming you are adding a disk that is of similar type to your existing disks—for example, adding a SATA disk to a system that already has SATA drives—the system should automatically detect the new disk at boot time. All that remains is partitioning it and creating file system(s) on it.

If you are adding a new type of disk (such as a SAS [Serial Attached SCSI] disk on a system that has only SATA drives), you may need to ensure that your kernel supports the new hardware. This support can either be built directly into the kernel or be available as a loadable module (driver). Note that the kernels of most Linux distributions come with support for many popular disk/storage controllers.

Once the disk is in place, simply boot the system, and you're ready to go. If you aren't sure about whether the system can see the new disk, run the **dmesg** command and view what the kernel has detected. Here's an example:

```
[root@server ~]# dmesg | egrep -i "hd|sd|disk|vd"
```

Overview of Partitions

For the sake of clarity, and in case you need to know what a partition is and how it works, let's briefly review this subject. Disks typically need to be *partitioned* before use. Partitions divide the disk into segments, and each segment acts as a complete disk by itself. Once a partition is filled with data, the data cannot automatically overflow onto another partition.

Various things can be done with a partitioned disk, such as installing an OS into a single partition that spans the entire disk, installing several different OSs into their own separate partitions in what is commonly called a "dual-boot" configuration, and using the different partitions to separate and restrict certain system functions into their own work areas.

This last example is especially relevant on a multiuser system, where the content of users' home directories should not be allowed to overgrow and disrupt important OS functions.

Traditional Disk and Partition Naming Conventions

Modern Linux distributions use the libATA library to provide support within the Linux kernel for various storage devices as well as host controllers. Under Linux, each disk is given its own device name. The device files are stored under the **/dev** directory.

Hard disks start with the name **sd*X*,** where *X* can range from *a* through *z,* with each letter representing a physical block device. For example, in a system with two hard disks, the first hard disk would be **/dev/sda** and the second hard disk would be **/dev/sdb**. Depending on the implementation/driver, *virtual* block devices start with names like **vd*X.***

When partitions are created, corresponding device files are created. They take the form of **/dev/sd*XY*** (or **/dev/vd*XY***), where *X* is the device letter (as described in the preceding paragraph) and *Y* is the partition number.

Thus, the first partition on the **/dev/sda** disk is **/dev/sda1**, the second partition would be **/dev/sda2**, the second partition on the third disk would be **/dev/sdc2**, and so on.

Some standard devices are automatically created during system installation, and others are created as they are connected to the system.

Volume Management

You may have noticed earlier that we use the terms "partition" and "volume" interchangeably in parts of the text. Although they are not exactly the same, they are similar in a conceptual way.

Volume management is a new approach to dealing with disks and partitions: Instead of viewing a disk or storage entity along partition boundaries, the boundaries are no longer present and everything is now seen as volumes. (That made perfect sense, didn't it? Don't worry if it didn't; this is a tricky concept. Let's try this again with more detail.)

This new approach to dealing with partitions is called *Logical Volume Management (LVM)* in Linux. It offers several benefits and removes the restrictions, constraints, and limitations that the concept of partitions imposes. Following are some of the benefits:

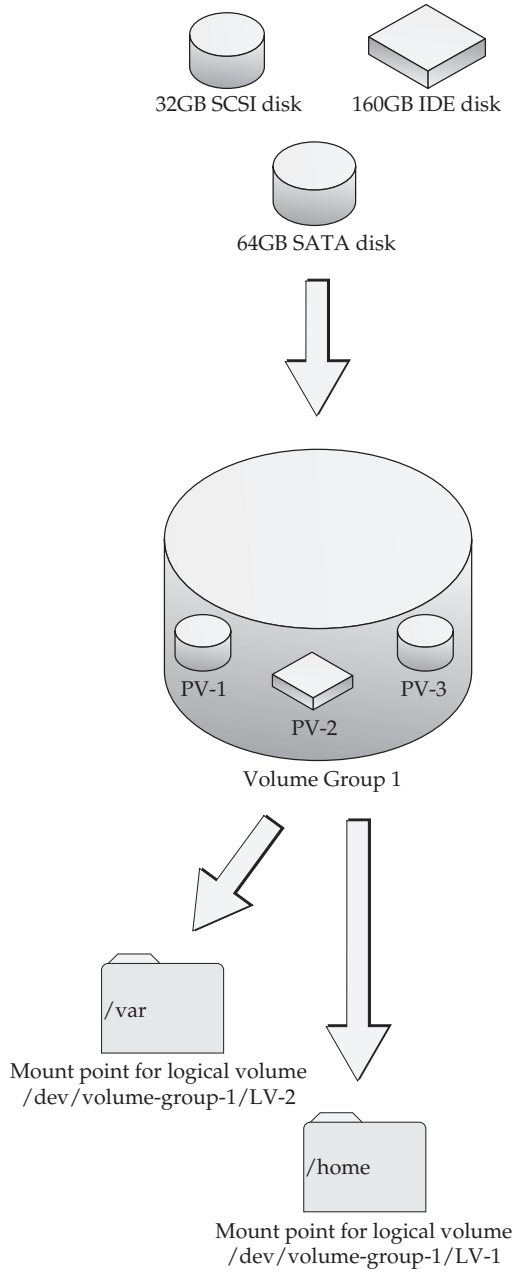
- Greater flexibility for disk partitioning
- Easier online resizing of volumes
- Easier to increase storage space by simply adding new disks to the storage pool
- Use of snapshots

Following are some important volume management terms:

- **Physical volume (PV)** This typically refers to the physical hard disk(s) or another physical storage entity, such as a Redundant Array of Inexpensive Disks (RAID) array or iSCSI LUN. Only a single storage entity (for example, one partition) can exist in a PV.
- **Volume group (VG)** Volume groups are used to house one or more physical volumes and logical volumes into a single administrative unit. A volume group is created out of physical volumes. VGs are simply a collection of PVs; however, VGs are not mountable. They are more like virtual raw disks.
- **Logical volume (LV)** This is perhaps the trickiest LVM concept to grasp, because logical volumes (LVs) are the equivalent of disk partitions in a non-LVM world. The LV appears as a standard block device. We put file systems on the LV, and the LV gets mounted. The LV gets `fsck`-ed if necessary.

  LVs are created out of the space available in VGs. To the administrator, an LV appears as one contiguous partition, independent of the actual PVs from which it is derived.
- **Extents** Two kinds of extents can be used: physical extents and logical extents. Physical volumes (PVs) are said to be divided into chunks, or units of data, called "physical extents." Logical volumes (LVs) are said to be divided into chunks, or units of data, called "logical extents."

The following illustration shows the relationship between disks, physical volumes (PVs), volume groups (VGs), and logical volumes (LVs) in LVM:



STEP A:
Prepare the storage pool. This pool consists of three different hard disks.

STEP B:
Create a volume group by assigning disks from your storage pool (Step A) to a volume group. Assuming we use physical extents (PEs) that are 32MB each, then:
SCSI disk = PV-1 = 32GB = 1000 PE
IDE disk = PV-2 = 160GB = 5000 PE
SATA disk = PV-3 = 64GB = 2000 PE

Therefore, there are a total of 87,000 PEs (or ~224GB) available to the volume group.

STEP C:
Create logical volumes (LVs) by using free space from the parent volume group from Step B.

1. Create an LV called LV-1, format it, and mount its file system on the/**home** directory. The LV has been created with 3125 logical extents (or 100GB).

2. Create another LV called LV-2, format it, and mount it at the/**var** directory; it has been created with 62 logical extents (or 2GB).

Creating Partitions and Logical Volumes

During the OS installation process, you probably used a "pretty" tool with a nice GUI front-end to create partitions. The GUI tools available across the various Linux distributions vary greatly in looks and ease of use. Two command-line tools that can be used to perform most partitioning tasks, and that have a unified look and feel regardless of the Linux flavor, are the venerable **`parted`** and **`fdisk`** utilities.

Although **`fdisk`** is small and somewhat awkward, it's a reliable command-line partitioning tool.

`parted`, on the other hand, is much more user-friendly and has a lot more built-in functionalities than other tools have. In fact, a lot of the GUI partitioning tools call the **`parted`** program in their back-end. Therefore, you should be familiar with foundational tools such as **`parted`** and **`fdisk`**. Other powerful command-line utilities for managing partitions are **`sfdisk`** and **`cfdisk`**.

During the installation of the OS, as covered in Chapter 2, you would have probably ended up with some free unallocated space on the disk *if* you accepted the default partitioning scheme. We will now use that free space to demonstrate some LVM concepts by walking through the steps required to create a logical volume.

In particular, we will create a 20GB-sized logical volume that will house the contents of our current **/var** directory. Because a separate **/var** volume was *not* created during the OS installation, the contents of the **/var** directory are currently stored under the volume that holds the root (/) tree. The general idea is that because the **/var** directory is typically used to hold frequently changing and growing data (such as log files), it is prudent to put its content on its own separate file system.

The steps we'll follow can be summarized this way:

1. Examine the current disk partition layout using the **`parted`** utility.
2. Examine the current LVM layout using the **`pvdisplay`**, **`vgdisplay`**, and **`lvdisplay`** utilities.
3. Determine how much unallocated space we have on our existing volume group(s).
4. Finally, create a new logical volume within the volume group, format the volume, and assign mount points to the logical volume.

CAUTION The process of creating partitions is potentially irrevocably destructive to the data already on the disk. Before creating, changing, or removing partitions on any disk, you must be sure of what you are doing and the consequences.

Beyond the *bother* of having to physically insert a new disk into the server chassis, the actual steps of making the disk available/usable to the OS comprise a simple but methodical process from start to finish. We've interspersed the *process* in the following sections with some *extra steps,* along with some notes and explanations.

Some common and handy LVM utilities used are listed and described in Table 8-2.

LVM Command	Description
lvcreate	Creates a new logical volume in a volume group by allocating logical extents from the free physical extent pool of that volume group
lvdisplay	Displays the attributes of a logical volume, such as read/write status, size, and snapshot information
pvcreate	Initializes a physical volume for use with the LVM system
pvdisplay	Displays the attributes of physical volumes, such as size and PE size
vgcreate	Creates new volume groups from block devices created using the **pvcreate** command
vgextend	Adds one or more physical volumes to an existing volume group to extend its size
vgdisplay	Displays the attributes of volume groups

Table 8-2 LVM Utilities

Examining Disk/Partition Layout

Let's examine the current partition or disk layout of the main system disk, **/dev/sda**, by following these steps:

1. Begin by running the **parted** utility with the device name as a parameter:

```
[master@fedora-server ~]#   sudo parted /dev/sda
Welcome to GNU Parted! Type 'help' to view a list of commands.
(parted)
```

   You will be presented with a simple **parted** prompt: (parted).

2. Print the partition table again while at the **parted** shell. Type **print** at the prompt to print the current partition table:

```
(parted)  print
Model: SATA HARDDISK
Disk /dev/sda: 107GB
Sector size (logical/physical): 512B/512B
Partition Table: gpt
Disk Flags:
Number  Start   End     Size    File system   Name                  Flags
 1      1049kB  211MB   210MB   fat16         EFI System Partition boot, esp
 2      211MB   1285MB  1074MB  xfs
 3      1285MB  107GB   106GB                                       lvm
```

   A few facts are worthy of note regarding this output:

   - The total disk size is approximately 107GB.
   - The partition table type is the GUID Partition Table (GPT) type. Three partitions are currently defined on our sample system: 1, 2, and 3 (**/dev/sda1**, **/dev/sda2**, and **/dev/sda3,** respectively).

- Partition 1 (**/dev/sda1**) is marked with a boot flag (*). This means that it is a bootable partition. Specifically, it is the special required **EFI** partition used on UEFI-based systems.
- Partition 2 (**/dev/sda2**) is our traditional xfs-formatted **/boot** partition.
- Partition 3 (**/dev/sda3**) is the last partition spanning the rest of the disk and is marked with the **`lvm`** flag.
- From the partitioning scheme we chose during the OS installation, we can deduce that partition 1 (**/dev/sda1)** houses the **/boot/efi** file system and partition 2 is **/boot**, while partition 3 (**/dev/sda3)** houses everything else (see the output of the **`df`** command for reference).
- The last partition (3, or **/dev/sda3**) ends smack dab on the 1074MB (107GB) boundary. Therefore, there isn't any more room to create additional partitions!

3. We are done admiring the disk layout, so type **`quit`** at the **`(parted)`** prompt and press ENTER:

```
(parted) quit
```

You will be returned back to your regular command shell (Bash in our case).

TIP Assuming you had free usable space on the disk or even extra/additional disks connected to the system, you can use a combination of various **`parted`** subcommands, such as **`mkpart`**, **`set`**, and so on, to create new partitions.

Keep in mind that in some very rare cases, you may need to reboot the system or unplug and reinsert the newly partitioned block device in order to allow the Linux kernel to recognize or use newly created partitions. Running utilities like **`partprobe`**, **`partx`**, **`blockdev`**, **`hdparm`** with the right options can help inform the OS of partition table changes without rebooting.

Exploring Physical Volume(s)

We will use the **`pvdisplay`** command in the following procedures to examine the current physical volume on the system.

1. Make sure you are still logged into the system as a user with superuser privileges.
2. Let's view the current physical volumes defined on the system. Type **`pvdisplay`** at the prompt, like so:

```
[master@fedora-server ~]$ sudo   pvdisplay
  PV Name               /dev/sda3
  VG Name               fedora
  PV Size               98.80 GiB / not usable 2.00 MiB
...<OUTPUT TRUNCATED>…
```

Take note of the physical volume name field (**`PV Name`**). Our sample output shows that the **/dev/sda3** partition is currently initialized as a physical volume.

Exploring Volume Group(s)

We will examine/explore any volume group (VG) defined on the system.

Use the **vgdisplay** command to view the current volume groups that might exist on your system:

```
[master@fedora-server ~]$ sudo  vgdisplay
  VG Name               fedora
  Format                lvm2
 ...<OUTPUT TRUNCATED>...
  VG Size               98.80 GiB
  PE Size               4.00 MiB
  Total PE              25293
  Alloc PE / Size       5865 / 22.91 GiB
  Free  PE / Size       19428 / 75.89 GiB
  VG UUID               o7zuzN-jyCY-k1Zi-mgfT-MrjZ-sz6E-Cbmxeo
```

From the preceding output, we can see the following:

- The volume group name (VG Name) is **fedora**.
- The current size of the VG is 98.80 GiB
- The physical extent size is 4.00 MiB, and there are a total of 25,293 PEs.
- There are 19,428 physical extent (or ~75.89 GiB) free in the VG.

With the all free, usable extents/space that we've uncovered, we'll next go on to carving out a logical volume (LV) in the following section.

Creating a Logical Volume

Now that we've unraveled the free space on the VG, we can go ahead and create the logical volume (LV) for our future **/var** file system.

1. First view the current LVs on the system:

```
[master@fedora-server ~]$ sudo  lvdisplay
 --- Logical volume ---
  LV Path                /dev/fedora/root
  LV Name                root
  VG Name                fedora
 ...<OUTPUT TRUNCATED>...
  --- Logical volume ---
  LV Path                /dev/fedora/swap
  LV Name                swap
  VG Name                fedora
  ...<OUTPUT TRUNCATED>...
```

The preceding output shows the current LVs:

- **/dev/fedora/root**
- **/dev/fedora/swap**

2. With the background information that we now have, we will create an LV using the same naming convention (that is, name of mount point) currently used on the system. We will create a third LV of size 20GB called **var** on the **fedora** VG.

   The full path to the LV will be **/dev/fedora/var**. Type the following:

```
[master@fedora-server ~]$ sudo lvcreate  -L 20G  --name var fedora
Logical volume "var" created
```

NOTE You can actually name your LV any way you want. We named ours **var** for consistency only. We could have replaced **var** with another name, such as "my-volume" or "LogVol03" if we wanted to. The value for the **--name** (**-n**) option determines the name of the LV.

Also, the **-L** option specifies the size in human-readable (gigabyte or megabytes) units. We could have also specified the size in megabytes by using an option such as **-L 19030M**.

3. View the LV you created by typing the following:

```
[master@fedora-server ~]$ sudo  lvdisplay /dev/fedora/var
  LV Path                /dev/fedora/var
  LV Name                var
  VG Name                fedora
  LV Size                20.00 GiB
 ...<OUTPUT TRUNCATED>...
```

TIP You can install a GUI tool named **blivet-gui** on Fedora, RHEL, and CentOS Linux distributions that can greatly simplify the entire management of an LVM system.

The openSUSE Linux distribution also includes a very capable GUI tool for managing disks, partitions, and the LVM. Issue the command **yast2 disk** to launch the utility. GNOME desktop environments usually have a nicely integrated GUI tool named **gnome-disks** that you can also use for storage/disk management.

Creating File Systems

After creating any volume(s), you next need to put file system(s) on them to make them actually useful. (If you're accustomed to Microsoft Windows, this is akin to formatting the disk once you've partitioned it.)

The type of file system you want to create will determine the particular utility you should use. In this project, we want to create an XFS-type file system; therefore, we'll use the **mkfs.xfs** utility. As indicated earlier in this chapter, XFS is considered a highly performant and production-ready file system; therefore, you should be able to use it for most production type workloads. Many command-line parameters are available for the **mkfs.xfs** tool, but we'll use it in its simplest form here.

Following are the steps for creating a file system:

1. The only command-line parameter you'll usually have to specify is the name of the partition (or volume) onto which the file system should go. To create a file system on **/dev/fedora/var**, issue the following command:

```
 [master@fedora-server ~]$ sudo mkfs.xfs  /dev/fedora/var
meta-data=/dev/fedora/var isize=512  agcount=4, agsize=1310720 blks
...<OUTPUT TRUNCATED>...
```

Once the preceding command runs to completion, the file system will be created.

We will next begin the process of trying to relocate the contents of the current **/var** directory to its own separate (and new) file system.

2. Create a temporary folder that will be used as the mount point for the new file system. Create it under the root folder:

```
[master@fedora-server ~]$ sudo  mkdir /new_var
```

3. Mount the new **var** logical volume at the /**new_var** directory:

```
[master@fedora-server ~]$ sudo  mount /dev/fedora/var /new_var/
```

4. Copy the content of the current **/var** directory to the /**new_var** directory:

```
[master@fedora-server ~]$ sudo  cp  -vrp /var/*   /new_var/
```

5. Now you can rename the current **/var** directory to /**old_var**:

```
[master@fedora-server ~]$ sudo mv  /var /old_var
```

6. Create a new and empty **/var** directory:

```
[master@fedora-server ~]$ sudo  mkdir /var
```

7. To avoid taking the system down to single-user mode to perform the following sensitive steps, we will type the following:

```
[master@fedora-server ~]$ sudo  mount --bind /new_var/  /var
```

This step temporarily remounts the /**new_var** directory to the /**var** directory where the system actually expects it to be. This is done by using the **`bind`** option with the **`mount`** utility. This is useful until we are good and ready to reboot the system.

TIP The **`bind`** option can also be useful on systems running the NFS service. This is because the **rpc_pipefs** pseudo-file system is often automatically mounted under a subfolder in the **/var** directory (**/var/lib/nfs/rpc_pipefs**). So to get around this, you can use the **`mount`** utility with the **`bind`** option to mount the **rpc_pipefs** pseudo-file system temporarily in a new location so that the NFS service can continue working uninterrupted. The command to do this in our sample scenario would be as follows:

```
# mount --bind /var/lib/nfs/rpc_pipefs \ /new_var/lib/nfs/rpc_pipefs
```

8. This step usually is optional, but it may be necessary in certain Linux distros (such as Fedora, RHEL, and CentOS) that have SELinux enabled to restore the security contexts for the new **/var** folder so that the daemons that need it can use it:

```
[master@fedora-server ~]$ sudo restorecon -R /var
```

9. We need to create an entry for the new file system in the **/etc/fstab** file. To do so, we must edit the **/etc/fstab** file so that our changes can take effect the next time the system is rebooted. Open the file for editing with any text editor of your choice and add the following entry into the file:

```
/dev/mapper/fedora-var   /var   xfs  defaults  1   2
```

TIP You can also use the **echo** and **tee** commands to append the preceding text to the end of the file. The command is:

```
$ echo "/dev/mapper/fedora-var  /var  xfs  defaults  1 2" | \
sudo tee -a /etc/fstab
```

10. This is a good time to reboot the system:

```
[master@fedora-server ~]$ sudo  shutdown  -r now
```

11. Hopefully, the system came back up fine. After the system boots, delete the **/old_var** and **/new_var** folders using the **rm** command.

TIP While still on the subject of file systems, this might be a good time to increase or expand the size of the root volume group to fill up any remaining space. The current size was chosen by default during our initial OS install in Chapter 2. We'll use the **lvresize** command to do this on our demo system, as follows:

```
$ sudo lvresize  --resizefs --extents +100%FREE  /dev/fedora/root
```

Summary

In this chapter, we discussed some de facto Linux file systems such as the extended file system family (ext2, ext3, ext4), XFS, and Btrfs. We covered the process of administering your file systems, and we touched on various storage administrative tasks.

We also went through the process of moving a sensitive system directory (**/var**) onto its own separate file system (XFS). The exercise detailed what you might need to do while managing a Linux server in the real world. With this information, you're armed with what you need to manage basic file system issues on a production-grade Linux-based server in a variety of environments.

CHAPTER

9 Core System Services

Regardless of distribution, network configuration, and overall system design, every Linux-based system ships with some core services. These services include **systemd**, **init**, the logging daemon, **cron**, and others. The functions performed by these services might seem simple, but they are very fundamental. Without them, a great deal of Linux's power and utility would be missed.

This chapter will discuss each of the core services and their configuration, in addition to another useful system service called **xinetd**. You'll find that the sections covering these simple services are not terribly long, but don't neglect this material. Take some time to get familiar with these services. Many creative solutions have been realized through the use of these services. Hopefully, this chapter will inspire a few more.

systemd

At its core, **systemd** is a system and startup manager. **systemd** is the result of an incredibly ambitious project that has reengineered the way services and other bootup procedures have traditionally worked on GNU/Linux systems. In its early days, the project faced some resistance from the Free and Open Source Software (FOSS) community due to the usual growing pains associated with most new major applications. The backlash against systemd was significant because the project is opinionated and changes a lot of the traditional ways that folks are used to managing or administering systems. That was in the past and systemd has since matured and has been adopted by and incorporated into most of the mainstream Linux distributions. It is now the de facto system and service manager! It has effectively replaced traditional tools and subsystems such as **init**, **telinit**, **inittab**, **SysV**, **upstart**, and so on. In fact, you will find that most of the traditional system startup subsystems and tools are now actually just symbolic links to the **systemd** binary and its minions!

The **systemd** project's web page (www.freedesktop.org/wiki/Software/systemd) offers the following description:

> "systemd is a suite of basic building blocks for a Linux system. It provides a system and service manager that runs as PID 1 and starts the rest of the system. systemd provides aggressive parallelization capabilities, uses socket and D-Bus activation for starting services, offers on-demand starting of daemons, keeps track of processes using Linux control groups, supports snapshotting and restoring of the system state, maintains mount and automount points and implements an elaborate transactional dependency-based service control logic. systemd supports SysV and LSB init scripts and works as a replacement for sysvinit."

The following sections take apart the official description for **systemd** and try to explain each component.

systemd's Role

As is expected of a startup manager, **systemd** manages various system startup and shutdown functions; it also manages the startup and shutdown of services (daemons) on Linux-based operating systems. **systemd** goes a little further by also playing the role of a babysitter of sorts to services of which it is aware. This means that **systemd** can monitor the services throughout their lifetime and automatically restart them, gather statistics about them, or report on them if necessary.

Because for the longest time the traditional way of managing system services has been through the use of startup shell scripts (for example, System V **init**), **systemd** provides compatibility and support for some existing System V and Linux Standard Base (LSB) specification **init** scripts.

The systemd Edge

One of the advantages that **systemd** brings to service/system management in Linux is its so-called "aggressive parallelization" capabilities. Put simply, this means that **systemd** can start several system services in parallel or concurrently. It does away with the traditional approach of starting services sequentially based on the numbering of the corresponding run control (**rc**) script. This parallelization equates to quicker startup times for Linux systems.

systemd also no longer uses traditional shell scripts to store the configuration information for services. The often tedious-to-read shell scripts have been replaced by simpler configuration files.

In addition, **systemd** records the start and exit time, the PID, and the exit status of every process it spawns and supervises. This is useful for troubleshooting daemons or other services.

How systemd Works

To do its work, **systemd** uses various Linux concepts and subsystems. Some of these are described next.

Control Groups (cgroups)

cgroups is a kernel-provided facility that allows processes to be arranged hierarchically and labeled individually. **systemd** places every process that it starts in a control group named

after its service, and this allows it to keep track of processes and allows **systemd** to have greater knowledge and control of a service throughout its life span. For example, **systemd** can safely end or kill a process as well as any child processes it might have spawned.

Socket Activation

The benefits to using **systemd** come from its proper and inherent understanding of the interdependence among system services—that is, it knows what various system services require from each other. As it turns out, most startup services or daemons actually need only the socket(s) provided by certain services and not the high-level services themselves. Because **systemd** knows this, it ensures that any needed sockets are made available very early on during system startup. It thus avoids the need to start a service first that provides a service as well as a socket. (If this is still a little confusing, see the sidebar "Human Digestive System vs. systemd" for an analogy.)

TIP The two main types of Linux sockets are the file-system-related **AF_UNIX** and **AF_LOCAL** sockets and the networking-related **AF_INET** sockets.

The **AF_UNIX** or **AF_LOCAL** socket family is used for communicating between processes on the same machine efficiently. The **AF_INET** sockets, on the other hand, provide interprocess communication (IPC) between processes that run on the same machine as well as between processes that run on different machines.

Human Digestive System vs. systemd

The following steps summarize how the normal human digestive system works to keep us alive (for example, a person needs the nutrients obtained from food to survive):

1. A person obtains food, opens their mouth, and puts the food into their mouth.
2. The person chews the food.
3. The food travels through various digestive tracts—for example, the esophagus, stomach, and small and large intestines—and then gets mixed with digestive juices and such and is then digested.
4. Through the digestive process, the food is broken down into chemicals that are useful to the body. These chemicals are the nutrients.
5. The nutrients are then absorbed by the body and transported throughout the body for use.

Conversely, **systemd** would distill the preceding tedious five-step procedure to get essential nutrients for the body into two steps:

1. Obtain the food and extract the raw nutrients from it.
2. Inject the raw nutrients directly into the human blood stream intravenously!

Units

The objects that **systemd** manages are called *units,* and they form the building blocks of **systemd**. These objects can include services or daemons, devices, file system entities such as mount points, and so on. Units are named the same as their configuration files, and the configurations files are normally stored under various directories collectively called the Unit File Load Path. Standard (package or vendor created) unit configuration files are stored under the **/usr/lib/systemd/system/** directory. Any new (administrator created) unit files as well as any needed customizations to existing unit files should be copied over to the **/etc/systemd/system/** folder for actual use. Transient or ephemeral units created during system runtime are stored under the **/run/systemd/system/** directory, which is itself a temporary file system structure.

The following types of units exist:

- **Service units** These units include traditional system daemons or services. These daemons can be started, stopped, restarted, and reloaded. Here's an example of a service unit:

  ```
  /etc/systemd/system/foobar.service
  ```

- **Socket units** These units consist of local and network sockets that are used for interprocess communication in a system. They play a very important role in the socket-based activation feature that helps reduce interservice dependencies. Here's an example:

  ```
  /etc/systemd/system/foobar.socket
  ```

- **Device units** These units allow **systemd** to see and use kernel devices. Here's an example of a device unit:

  ```
  /etc/systemd/system/foobar.device
  ```

- **Mount units** These units are used for mounting and unmounting file systems. An example of a mount unit is shown here:

  ```
  /etc/systemd/system/foobar.mount
  ```

- **Target units** **systemd** uses targets instead of runlevels. Target units are used for the logical grouping of units. They don't actually do anything by themselves, but instead reference other units, thereby allowing the control of groups of units together. Here's an example:

  ```
  /etc/systemd/system/foobar.target
  ```

- **Timer units** These units are used for triggering activation of other units based on timers. Here's an example:

  ```
  /etc/systemd/system/foobar.timer
  ```

- **Snapshot units** These units are used to save the state of the set of **systemd** units temporarily. Here's an example:

  ```
  /etc/systemd/system/foobar.snapshot
  ```

TIP You can use the **systemctl** command to view, list, and manage units of specific types. For example, to view all the active target units, type:

```
# systemctl list-units --type=target
```

To view all the active and inactive mount units, type:

```
# systemctl list-units --all --type=mount
```

To view all the active and inactive units of every type, enter:

```
# systemctl list-units –all
```

To see if a service unit called NAME is configured to automatically start, type:

```
# systemctl is-enabled NAME.service
```

To enable a service unit called NAME to automatically start at system boot, type:

```
# systemctl enable NAME.service
```

To immediately start a service unit called NAME, type:

```
# systemctl start NAME.service
```

To enable a service unit called NAME and immediately start it, type:

```
# systemctl enable –-now NAME.service
```

To immediately stop a running service unit called NAME, type:

```
# systemctl stop NAME.service
```

To disable a unit service called NAME from automatically starting at system boot, type:

```
# systemctl disable NAME.service
```


Legacy Notes: init and upstart

We are putting these notes here not just for legacy or historical reasons, but also for practical reasons. There are still tons of Linux systems out there that run and depend on these traditional toolsets and subsystems (**init**, **telinit**, **upstart**, SysV, and so on). These Linux distros will be maintained and supported by their vendors for a foreseeably long time into the future. You will no doubt encounter these systems in your journeys as a system administrator.

Additionally, you will find that, for good technical reasons, the newer solutions *borrow* sound first-principle concepts from the legacy toolsets and whenever possible strive very hard to maintain backward compatibility with the older toolsets. Therefore, being a little familiar with the old and the new in this context is invaluable.

The init Daemon The **init** process used to be the grand patron of all processes. It is always the first process that gets started in any Linux/UNIX-based system. **init** is executed by the kernel and is responsible for starting all other processes initially on a system. The process identification (PID) for **init** is always 1. Should **init** ever fail, the rest of the system will likely follow suit.

The **init** daemon as it was traditionally known has been largely replaced on most new Linux distributions by different solutions. One solution is an upstart named **upstart** (pun intended), found on older Debian/Ubuntu distros.

The **init** process serves several roles: It is responsible for completing the initialization routines on a system, thus making it the ultimate parent process. And because **init**

(*continued*)

never dies, the system can always be sure of its presence and, if necessary, can make reference to it. The need to refer to **init** usually happens when a process dies before all of its spawned child processes have completed. Such orphaned processes are called zombie processes. The orphaned child processes automatically inherit **init** as their parent process and this forms the basis of another one of init's roles—*the zombie reaper*. A quick execution of the **`ps -ef`** command will show a number of processes that will have a parent process ID (PPID) of 1.

The /etc/inittab File Although very rarely used anymore, the **/etc/inittab** file contains all the information **init** needs for starting runlevels.

Here is the format of each line in this file:

```
id:runlevels:action:process
```

The telinit Command The mysterious force that tells **init** when to change runlevels is actually the **`telinit`** command. This command takes two command-line parameters. One is the desired runlevel that **init** needs to know about, and the other is **-t *sec***, where ***sec*** is the number of seconds to wait before telling **init**. Whether **init** actually changes runlevels is its decision. Obviously, it usually does, or this command wouldn't be terribly useful.

Under most UNIX implementations (including Linux), the **`telinit`** command is really just a symbolic link to the **init** program. Because of this, some folks prefer running **init** with the runlevel they want rather than using **`telinit`**. And on **systemd**-enabled distros, **`telinit`** is a symbolic link to the **systemctl** program.

It is extremely rare that you'll ever have to run the **`telinit`** command yourself. Usually, this is all handled for you by the startup and shutdown scripts.

upstart According to its documentation, "upstart is an event-based replacement for the init daemon which handles starting of tasks and services during boot, stopping them during shutdown, and supervising them while the system is running." This same description of **upstart** pretty much describes the function of the traditional **init** daemon, except that **upstart** tries to achieve its stated objectives in a more elegant and robust manner.

Another stated objective of **upstart** is to achieve complete backward compatibility with **init** (System V **init**). And **upstart** does indeed do a good job of handling this backward compatibility with **init** very well and transparently!

As previously mentioned, **upstart** is a replacement for the **init** daemon. **upstart** works using the notion of *jobs* (or tasks) and *events.*

On Debian-based distros such as Ubuntu that use **upstart**, jobs are created and placed under the **/etc/init/** directory. The name of the job is the filename under this directory—without the .conf extension.

(*continued*)

For example, the job definition that automatically sets the hostname for the system might be defined in a file named **/etc/init/hostname.conf**. The contents of the file look like this:

```
# This task is run on startup


# It  sets the system hostname from /etc/hostname.
description     "set system hostname"
start on startup
task
exec hostname -b -F /etc/hostname
```

Without going into too much detail, this job definition can be explained as follows:

- The **start** stanza specifies that the job be run during the occurrence of an event. The event in this case is the system **startup**. The startup event starts as early as possible during the system initialization.
- The **task** stanza specifies that this is a short-lived job. It is useful for jobs that run and finish completely when a certain event happens.
- The **exec** stanza specifies the path to a binary on the file system and optional arguments to pass to it. The binary in this case is the **hostname** command.

You can query the status of any job by using the **status** command. Here's an example that queries the status of our example **hostname** job:

```
root@ubuntu-server:~#  status hostname
```

The **initctl** command can be used to display a listing of all jobs and their states. This example lists all jobs and their states:

```
root@ubuntu-server:~#  initctl  list
```

xinetd

xinetd is an acronym for "extended Internet services daemon," and it is the more modern incarnation of the older **inetd**. The **xinetd** program can be considered a daemon process itself. You probably know that daemons are special programs that, after starting, voluntarily release control of the terminal from which they started. The main mechanism by which daemons can interface with the rest of the system is via IPC channels, by sending messages to the systemwide logging facility or by appending to a file on disk.

Strictly speaking, a Linux system can run effectively without the presence **xinetd**, but some other daemons rely solely on the functionality it provides. So if a daemon's author or authors decide that their daemon requires or relies on **xinetd (**or **similar)**, then you need it—no two ways about it! The **xinetd** program helps to start *other* programs that provide Internet services. It's a simple philosophy: Not all server processes (including those that accept new connections) are called upon so often that they each require an individual program to be running in memory all the time. **xinetd** can function as a "super-server" to other network-server-related processes, such as Telnet, FTP, TFTP, and so on. One of the main reason for the existence of a super-server is to conserve system resources. So instead of needing to maintain potentially dozens of services loaded in memory waiting to *perhaps* be used, **xinetd** listens on their behalf.

Stated another way, rather than having various programs or daemons *always* automatically start up during system initialization and remain unused until a connection request arrives, **xinetd** instead stands in the gap for those programs and listens on their normal service ports. As a result, when **xinetd** hears a service request meant for one of the services it manages, it starts or spins up the appropriate service.

A secondary benefit of x**inetd** is that it serves as a sort of network programing abstraction layer so that third-party daemon developers don't have to explicitly code network connectivity functions into their applications. This is possible because **xinetd** can transparently handle the network logic aspects and pass incoming network streams onto the third-party process as its standard input (**`STDIN`**). Likewise, any of the process's output (**`STDOUT`**) is also sent back to the host that has connected to the process—over the **xinetd** abstraction layer.

In comparison, to the older inetd daemon, **xinetd** uses a new configuration file format and offers a lot of additional features. For this reason, any (really old) application that relies on **inetd** might require some minor manual configuration adjustments to make to them work seamlessly under **xinetd.**

As a rule of thumb, low-volume services (such as TFTP) are usually best run through **xinetd**, whereas higher-volume services (such as web servers) are better run as stand-alone processes that are always in memory, ready to handle requests.

In this section, we cover the newer **xinetd** daemon.

NOTE Your Linux distribution might not have the **xinetd** software installed out of the box. The **xinetd** package can be installed with **`dnf`** (or **`yum`**) on a Fedora distro (or RHEL or CentOS) by running the following:

```
# sudo   dnf -y install xinetd
```

On Debian-based distros such as Ubuntu, **xinetd** can be installed using APT via the following:

```
# sudo  apt-get install xinetd
```

The /etc/xinetd.conf File

The **/etc/xinetd.conf** file consists of a series of blocks that take the format

```
blockname
{
   attribute = value
}
```

where ***blockname*** is the name of the block that is being defined, ***attribute*** (or ***variable***) is the name of an attribute being defined within the context of the block, and ***value*** is the value assigned to the ***attribute***. Every block can have multiple attributes defined within.

One special block is called **defaults**. Whatever attributes are defined within this block section will be inherited by all service configurations—unless explicitly overridden in the service configuration.

An exception to the block format is the **includedir** directive, which tells **xinetd** to read all the files in a directory and consider them part of the **/etc/xinetd.conf** file. Any line that begins with a pound sign (#) is the start of a comment. The stock **/etc/xinetd.conf** file that ships with Fedora/CentOS/RHEL looks like this:

```
# This is the master xinetd configuration file.
...<SOME OUTPUT TRUNCATED>...
defaults
{
      log_type       = SYSLOG daemon info
      log_on_failure = HOST
      log_on_success = PID HOST DURATION EXIT
      cps            = 50 10
      instances      = 50
      per_source     = 10
      v6only         = no
      groups         = yes
      umask          = 002
}
includedir /etc/xinetd.d
```

The first line of our sample **xinetd.conf** file is a comment explaining what the file is and what it does.

After the comments, you see the first block: **defaults**. The first attribute that is defined in this block is **log_type**, which is set to the value of **SYSLOG daemon info**. Nine attributes in total are defined in this block, the last one being **umask**. Since this block is titled **defaults**, the variables that are set within it will apply to all future blocks that are defined. We'll list the attributes and values and discuss their meaning in detail in the next section.

Finally, the last line of the file specifies that the **/etc/xinetd.d** directory must be examined for other files that contain more configuration information. This will cause **xinetd** to read all the files in that directory and parse them as if they were part of the **/etc/xinetd.conf** file.

Attributes and Their Meanings

Table 9-1 lists some of the attribute/variable names that are supported in the **/etc/xinetd.conf file**.

Variable	Description
`id`	This attribute is used to identify a service uniquely. This is useful because services exist that can use different protocols and that need to be described with different entries in the configuration file. By default, the service ID is the same as the service name.
`type`	Any combination of the following values may be used: `RPC` if this is a Remote Procedure Call (RPC) service, `INTERNAL` if this service is provided by **xinetd**, or `UNLISTED` if this is a service not listed in the **/etc/services** file.
`disable`	This is either `yes` or `no`. A `yes` value means that although the service is defined, it is not available for use.
`socket_type`	Valid values for this variable are `stream`, which indicates that this service is a stream-based service; `dgram`, which indicates that this service is a datagram; and `raw`, which indicates that this service uses raw IP datagrams. The `stream` value refers to connection-oriented TCP data streams (for example, Telnet and FTP). The `dgram` value refers to datagram (User Datagram Protocol [UDP]) streams (for example, the Trivial File Transfer Protocol [TFTP] service is a datagram-based protocol). Other protocols outside the scope of TCP/IP do exist, but you'll rarely encounter them.
`protocol`	Determines protocol type (either TCP or UDP) for the connection.
`wait`	If this is set to `yes`, it means that the service is single-threaded and only one connection will be processed at a time. If this is set to `no`, it means that the service is multithreaded and multiple connections will be allowed by running the appropriate service daemon multiple times.
`user`	Specifies the username under which this service will run. The username must exist in the **/etc/passwd** file.
`group`	Specifies the group name under which this service will run. The group must exist in the **/etc/group** file.
`instances`	Specifies the maximum number of concurrent connections this service is allowed to handle. The default is no limit (specified by `UNLIMITED`).
`server`	The name of the program to run when this service is connected.
`server_args`	The arguments passed to the server. In contrast to **inetd**, the name of the server should not be included in `server_args`.
`only_from`	Specifies the networks from which a valid connection may arrive. (This is the built-in TCP Wrapper functionality.) You can specify this in one of three ways: as a numeric address, a hostname, or a network address with netmask. The numeric address can take the form of a complete IP address to indicate a specific host (such as 192.168.1.1). However, if any of the ending octets are zeros, the address will be treated like a network where all of the octets that are zero are wildcards (for instance, 192.168.1.0 means any host that starts with the numbers 192.168.1). Alternatively, you can specify the number of bits in the netmask after a slash (for example, 192.168.1.0/24)

Table 9-1 xinetd Configuration File Attributes (*continued*)

Variable	Description
`no_access`	The opposite of **`only_from`** in that instead of specifying the addresses from which a connection is valid, this variable specifies the addresses from which a connection is invalid. It can take the same type of parameters as **`only_from`**.

`log_type`	Determines where logging information for that service will go. There are two valid values: **`SYSLOG`** and **`FILE`**. If **`SYSLOG`** is specified, you must specify to which **syslog** facility to log as well (see "The Logging Daemon," later in this chapter, for more information on facilities). For example, you can specify this:
	`log_type = SYSLOG local0`
	Optionally, you can include the log level as well:
	`log_type = SYSLOG local0 info`
	If **`FILE`** is specified, you must specify which filename to log. Optionally, you can also specify the soft limit on the file size—where an extra log message indicating that the file has gotten too large will be generated. If the soft limit is specified, a hard limit can also be specified. At the hard limit, no additional logging will be done. If the hard limit is not explicitly defined, it is set to be 1 percent higher than the soft limit. Here's an example of the **`FILE`** option:
	`log_type = FILE /var/log/mylog`

`log_on_success`	Specifies which information is logged on a connection success. The options include **`PID`** to log the process ID of the service that processed the request, **`HOST`** to specify the remote host connecting to the service, **`USERID`** to log the remote username (if available), **`EXIT`** to log the exit status or termination signal of the process, and **`DURATION`** to log the length of the connection.
`port`	Specifies the network port under which the service will run. If the service is listed in **/etc/services**, this port number must equal the value specified there.
`interface`	Allows a service to bind to a specific interface and be available only there. The value is the IP address of the interface to which you want this service to be bound. An example of this is binding less-secure services (such as Telnet) to an internal and physically secure interface on a firewall and not allowing the external, more vulnerable interface outside the firewall.
`cps`	The first argument specifies the maximum number of connections per second this service is allowed to handle. If the rate exceeds this value, the service is temporarily disabled for the second argument number of seconds. For example, `cps = 50 10` will disable a service for 10 seconds if the connection rate ever exceeds 50 connections per second.

Table 9-1 xinetd Configuration File Attributes

Examples: A Simple (echo) Service Entry

Using the **echo** service (provided by the **xinetd** package) as an example, let's take a look at one of the simplest entries possible with **xinetd**:

```
# This is the configuration for the tcp/stream echo service.
service echo
{
      disable           = yes
      id          = echo-stream
      type        = INTERNAL
      wait        = no
      socket_type = stream
}
```

As you can see, the entry is self-explanatory. The first line that begins with the pound character (#) indicates that this is a comment field and is thus ignored. The service name is **echo**, and because of the **socket_type** (that is, **stream**), we know this is a TCP service. The ***wait*** variable tells us that multiple **echo** processes can be running concurrently. Finally, the ***type*** attribute with a value of **INTERNAL** shows that this is a service provided by **xinetd**—in other words, it is not provided by any external package or daemon.

With our understanding of this **xinetd** service entry, let's try to enable and disable a sample **xinetd**-based service.

Enabling/Disabling the Echo Service

If you want a secure system, chances are you will run it with only a few services—some people don't even run **xinetd** at all! It takes just a few steps to enable or disable a service. For example, to enable a service, you would first enable it in the **xinetd** configuration file, restart the **xinetd** service, and finally test things out to make sure you have the behavior you expect. To disable a service, just do the opposite.

NOTE The service we will be exploring is the **echo** service. This service is internal to **xinetd** (that is, it is not provided by any external daemon).

Let's step through enabling the **echo** service:

1. Use any plain-text editor to edit the file **/etc/xinetd.d/echo-stream** and change the variable **disable** to **no**, as shown in the following file excerpt:

```
# This is the configuration for the tcp/stream echo service.
service echo
{
  disable               = no
  id                    = echo-stream
  type                  = INTERNAL
  wait                  = no
  socket_type           = stream
}
```

TIP On an Ubuntu-based system, the configuration file for the **echo** service is **/etc/xinetd.d/echo**. The Ubuntu distro goes further to combine the UDP and TCP versions of the **echo** service in one file. Fedora, on the other hand, sorts the UDP and TCP versions of the **echo** service into two separate files: **/etc/xinetd.d/echo-dgram and /etc/xinetd.d/echo-stream**.

2. Save your changes to the file and then exit the editor.
3. Restart the **xinetd** service. On most Linux distros, you can type the following:

```
[master@server ~]$ sudo service xinetd restart
```

On modern **systemd**-enabled distros, you can alternatively restart the **xinetd** service using the **systemctl** utility, like this:

```
[master@fedora-server ~]$ sudo  systemctl restart xinetd.service
```

4. Telnet to the port (port 7) of the **echo** service and see if the service is indeed running:

```
[master@fedora-server ~]$ telnet localhost 7
Connected to localhost.
Escape character is '^]'.
Hello World
Hello World
```

Your output should be similar to the preceding, if the **echo** service has been enabled.

You can type any character on your keyboard at the Telnet prompt and watch the character get echoed (repeated) back to you. We typed "Hello World" in the example. (As you can see, the **echo** service is one of those terribly useful and lifesaving services that users and system administrators *cannot* do without!)

Simultaneously, press CTRL-] on your keyboard to break the connection, and then type **quit** to exit Telnet and return back a shell prompt.

This exercise walked you through enabling a service by directly editing its **xinetd** configuration file. It is a simple process to enable or disable a service. However, you should actually go back and make sure that the service is indeed disabled (if that is what you want) by testing it—because it is always better to be safe than sorry. An example of being *sorry* is "thinking" that you have disabled the insecure Telnet service when it is in fact still running!

TIP You can quickly enable or disable a service that runs under **xinetd** by using the **chkconfig** utility, which is readily available (or installable) on older GNU/Linux distributions. For example, to disable the **echo-stream** service that you manually enabled, just issue the following command:

```
# chkconfig echo-stream off
```

The Logging Daemon

With so much going on at any one time, especially with services that are disconnected from a terminal window, a standard mechanism by which special events and messages can be logged is required. Linux distributions have traditionally used the **syslogd** (**sysklogd**) daemon to provide this service. However, modern Linux distros have standardized on the **rsyslog**

package as well as the logging component of **systemd**, referred to as **systemd-journald** (**journald**), for logging-related functions.

Regardless of the software used, the idea remains the same, and the end result (to gather system logs) is mostly the same; the main differences between the new approaches are in the additional feature sets offered. In this section, we will be concentrating on **rsyslog**. There will be occasional references to **syslogd** when appropriate. Managing and configuring **rsyslog** is similar to the way it is done in **syslogd**. The **rsyslog** daemon maintains backward compatibility with the traditional **syslog** daemon but offers a plethora of new features as well.

rsyslogd

The **rsyslog** daemon provides a standardized means of performing logging. Many other platforms' systems employ a compatible daemon, thus providing a means for cross-platform logging over the network. This is especially valuable in a large heterogeneous environment where it's necessary to centralize the collection of log entries to gain an accurate picture of what's going on. You could equate this system of logging facilities to the Event Viewer functionality in Windows.

rsyslogd can send its output to various destinations: straight text files (usually stored in the **/var/log** directory), Structured Query Language (SQL) databases, other hosts, and more. Each log entry consists of a single line containing the date, time, hostname, process name, PID, and the message from that process. A system-wide function in the standard C library provides an easy mechanism for generating log messages. If you don't feel like writing code but want to generate entries in the logs, you have the option of using the `logger` command.

Invoking rsyslogd

If you do find a need to either start **rsyslogd** manually or modify the script that starts it up at boot, you'll need to be aware of **rsyslogd**'s command-line parameters, shown in Table 9-2.

Log Message Classifications

A basic understanding of how log messages are classified in the traditional **syslog** daemon way is also useful in helping you understand the configuration file format for **rsyslogd.**

Each message has a *facility* and a *priority.* The facility tells you from which subsystem the message originated, and the priority tells you the message's importance. These two values are separated by a period. Both values have string equivalents, making them easier to remember. The combination of the facility and priority makes up the "selector" part of a rule in the configuration file. Common log facilities and priorities are listed in Tables 9-3 and 9-4, respectively.

NOTE The priority levels are in the order of severity according to **syslogd**. Thus, `debug` is not considered severe at all, and `emerg` is the most crucial. For example, the combination facility-and-priority string `mail.crit` indicates there is a critical error in the mail subsystem (for example, it has run out of disk space). **syslogd** considers this message more important than `mail.info`, which may simply note the arrival of another message.

Parameter	Description
`-d`	Debug mode. Normally, at startup, **rsyslogd** detaches itself from the current terminal and starts running in the background. With the `-d` option, **rsyslogd** retains control of the terminal and prints debugging information as messages are logged. It's extremely unlikely that you'll need this option.
`-f config`	Specifies a configuration file as an alternative to the default **/etc/rsyslog.conf**.
`-h`	By default, **rsyslogd** does not forward messages sent to it that were destined for another host. This option will allow the daemon to forward logs received remotely to other forwarding hosts that have been configured.
`-l hostlist`	This option lets you list the hosts for which only the simple hostname should be logged and not the fully qualified domain name (FQDN). You can list multiple hosts, as long as they are separated by a colon, like so: `-l ubuntu-serverA:serverB`
`-m interval`	By default, **rsyslogd** generates a log entry every 20 minutes as a "just so you know I'm running" message. This is for systems that might not be busy. (If you're watching the system log and don't see a single message in more than 20 minutes, you'll know for a fact that something has gone wrong.) By specifying a numeric value for `interval`, you can indicate the number of minutes **rsyslogd** should wait before generating another message. Setting a value of zero for this option turns it off completely.
`-s domainlist`	If you are receiving **rsyslogd** entries that show the entire FQDN, you can have **rsyslogd** strip off the domain name and leave just the hostname. Simply list the domain names to remove in a colon-separated list as the parameter to the `-s` option. Here's an example: `-s example.com:domain.com`

Table 9-2 rsyslogd Command-Line Parameters

In addition to the priority levels in Table 9-4, **rsyslogd** understands wildcards. Thus, you can define a whole class of messages; for instance, `mail.*` refers to all messages related to the mail subsystem.

rsyslogd Configuration File

The **/etc/rsyslog.conf** file contains the configuration information that **rsyslogd** needs to run. The default configuration file that ships with most systems is sufficient for most standard needs. But you may find that you have to tweak the file a little if you want to do any additional fancy things with your logs—such as sending local log messages to remote logging machines that can accept them, logging to a database, reformatting logs, indexing the logs into a SIEM engine, and so on.

Facility	Description
`auth`	Authentication messages.
`authpriv`	Essentially the same as `auth`.
`cron`	Messages generated by the `cron` subsystem.
`daemon`	Generic classification for service daemons.
`kern`	Kernel messages.
`lpr`	Printer subsystem messages.
`mail`	Mail subsystem messages.
`mark`	Obsolete, but some books still discuss it; **syslogd** simply ignores it.
`news`	Messages through the Network News Transfer Protocol (NNTP) subsystem.
`security`	Same thing as `auth`; should not be used.
`syslog`	Internal messages from **syslog** itself.
`user`	Generic messages from user programs.
`uucp`	Messages from the UUCP (UNIX-to-UNIX copy) subsystem.
`local0-local9`	Generic facility levels whose importance can be decided based on your needs.

Table 9-3 Logging Facilities

Format of /etc/rsyslog.conf

rsyslogd's configuration relies heavily on the concepts of templates. To understand the syntax of **rsyslogd**'s configuration file, let's begin by explaining a few key points:

- Templates define the format of log messages. They can also be used for dynamic filename generation. Templates must be defined before they are used in rules. A template is made of several parts: the template directive, a descriptive name, the template text, and possibly other options.
- Any entry in the **/etc/rsyslog.conf** file that begins with a dollar (**$**) sign is a directive.

Priority	Priority Numeric Equivalent	Description
`debug`	7	Debugging statements
`info`	6	Miscellaneous information
`notice`	5	Important statements, but not necessarily bad news
`warn` or `warning`	4	Potentially dangerous situation
`err` or `error`	3	An error condition
`crit`	2	Critical situation
`alert`	1	A message indicating an important occurrence
`emerg` or `panic`	0	An emergency situation

Table 9-4 Logging Priority Levels

- Log message properties refer to well-defined fields in any log message. Examples of common message properties are shown in Table 9-5.
- The percentage sign (**%**) is used to enclose log message properties.
- Properties can be modified by the use of property replacers.
- Any entry that begins with a pound sign **(#)** is a comment and is ignored. Empty lines are also ignored.

rsyslogd Templates

As mentioned, **rsyslogd** relies on the use of templates, and the templates define the format of logged messages. The use of templates is what allows **rsyslogd** to produce log files that are formatted similarly to the traditional format that is produced by the older **syslogd**. Templates that support the **syslogd** log message format are hard-coded into **rsyslogd** and are used by default.

A sample template directive that supports the use of the traditional **syslogd** message format is shown here:

```
template(name="RSYSLOG_TraditionalFileFormat" type="string"
  string="%TIMESTAMP% %HOSTNAME%  %syslogtag%  %msg%  RANDOM TEXT \n"  <OPTION>)
```

The various fields and options of this sample template are explained in the following list. Table 9-5 describes some commonly used message properties.

- **template** This directive implies that the line is a template definition.
- **name** This is a unique and descriptive template name. The template name in this example is **RSYSLOG_TraditionalFileFormat**, and it refers to a built-in template that mimics the traditional syslog log format. Template names beginning with "**RSYSLOG_**" are reserved for **rsyslog** use.

Property Name (propname)	Description
msg	The MSG part of the message; the actual log message
rawmsg	The message exactly as it was received from the socket
HOSTNAME	Hostname from the message
FROMHOST	Hostname of the system from which the message was received (might not necessarily be the original sender)
syslogtag	TAG from the message
PRI-text	The priority (PRI) part of the message in a textual form
syslogfacility	The facility from the message in numerical form
syslogfacility-text	The facility from the message in text form
syslogseverity	The severity from the message in numerical form
syslogseverity-text	Severity from the message in text form
timereported	Timestamp from the message
MSGID	The contents of the MSGID field

Table 9-5 rsyslog's Message Property Names

- **type** This parameter is used to indicate the template type being used. Different types enable different ways to specify the template contents. The **string** type is used for textual content. Other possible types are list, subtree, and plugin.
- **string** This is the actual content of the string. It makes use of predefined properties (that is, property replacers) that are enclosed between two percentage (%) symbols. The entire **string** in the example is made up of the following message properties:
  - **%TIMESTAMP%** This represents when the message was received.
  - **%HOSTNAME%** This specifies the **`HOSTNAME`** property.
  - **%syslogtag%** This specifies the **`syslogtag`** property.
  - **%msg%** This is represents the *actual* message (MSG) part of the message.
  - **RANDOM TEXT** This can be any random/relevant string or text appended to the message.
  - **\n** The backslash is an escape character. Here, the **`\n`** implies a new line.
- **<options>** This entry is optional. It specifies options influencing the template as a whole. Example options are **sql**, **json**, and **stdsql**.

rsyslogd Rules

Rules can be created in **rsyslog.conf** using different filter criteria. Table 9-6 shows the currently available filter criteria.

Filter Type	Description	Syntax	Example
Facility based and Priority based	Use traditional facility and priority selectors	*`facility.priority action_field`*	`mail.info /var/log/messages`
Property based	Allow filtering on **rsyslog**-defined properties such as **`HOSTNAME`**, **`syslogtag`**, **`programname`**, and **`msg`**	*`:property, [!]compare_operation, "value"`*	`:hostname, isequal, "fedora-server"`
Expression based	Use **rsyslog**'s built-in scripting language called RainerScript to build filters	*`if EXPRESSION`* `then` *`ACTION else ACTION`*	`if $msg contains 'look' then { action(type="omfile" file="/var/log/look.log") else action(type="omfile" file="/var/log/nolook.log") }`

Table 9-6 rsyslog Filter Criteria

Let's delve into the traditional facility- and priority-based filter type rules a little more. This type of filter rule in the **rsyslog.conf** file is broken down into a selector field (facility.priority), an action field (or target field), and an optional template name. Specifying a template name after the last semicolon will assign the respective action to that template. Whenever a template name is missing, a hard-coded template is used instead. It is, of course, important that you make sure that the desired template is defined before referencing it.

Here is the format for each line in the configuration file:

```
selector_field   action_field ; <optional_template_name>
```

Here's an example:

```
mail.info     /var/log/messages;  RSYSLOG_TraditionalFileFormat
```

Selector Field The selector field specifies the combination of facilities and priorities. Here's a sample selector field entry:

```
mail.info
```

Here, **`mail`** is the facility and **`info`** is the priority.

Action Field The action field of a rule describes the action to be performed on a message. This action can be something as simple as writing the logs to a file or slightly more complex, such as writing to a database table or forwarding to another host. Here's a sample action field:

```
/var/log/messages
```

This action example indicates that the log messages should be written to the file named **/var/log/messages**.

Other common possible values for the action field are described in Table 9-7.

Action Field	Description
Regular file (e.g., **/var/log/messages**)	A regular file. A full path name to the file should be specified and should begin with a slash (/). This field can also refer to device files, such as .tty files, or the console, such as **/dev/console**.
Named pipe (e.g., **|/tmp/mypipe**)	A named pipe. A pipe symbol (|) must precede the path to the named pipe (First In First Out, or FIFO). This type of file is created with the **`mknod`** command. With **rsyslogd** feeding one side of the pipe, you can run another program that reads the other side of the pipe. This is an effective way to have programs parsing log output.
@loghost or **@@loghost**	A remote host. The at symbol (@) must begin this type of action, followed by the destination host. A single @ sign indicates that the log messages should be sent via traditional UDP. And double at symbols (@@) imply that the logs should be transmitted using TCP instead.

Table 9-7 Action Field Descriptions (*continued*)

Action Field	Description
List of users (e.g., **yyang**, **dude**, **root**)	This type of action indicates that the log messages should be sent to the list of currently logged-on users. The list of users is separated by commas (,). Specifying an asterisk (*) will send the specified logs to all currently logged-on users.
Discard	This action means that the logs should be discarded and no action should be performed on them. This type of action is specified by the tilde character (~) in the action field.
Database table (e.g., **>dbhost,dbname,dbuser, dbpassword;<dbtemplate>**)	This type of action is one of the advanced/new features that **rsyslogd** supports natively. It allows the log messages to be sent directly to a configured database table. This type of location needs to begin with the greater-than symbol (>). The parameters specified after > follow a strict order: The database hostname (*dbhost*) must be given first, followed by a comma, the database name (*dbname*), another comma, the database user (*dbuser*), a comma, and then the database user's password (*dbpassword*). An optional template name (*dbtemplate*) can be specified if a semicolon is specified after the last parameter.

Table 9-7 Action Field Descriptions

Sample /etc/rsyslog.conf File

Following is an excerpt from a sample **rsyslog.conf** file that highlights the facility- and priority-based filter type rules. This example is interspersed with comments that explain what the rules do.

```
# A template definition that resembles traditional syslogd file output
$template myTraditionalFormat,"%timegenerated% %HOSTNAME% %syslogtag%%msg%\n"

# Log all kernel messages to the console.
kern.*   /dev/console

# Log anything(except mail)of level info

# or higher into /var/log/messages file.
# Exclude private authentication messages!
# The rule is using the hard-coded traditional format
#  because a different template name has NOT been defined.
*.info;mail.none;authpriv.none;cron.none      /var/log/messages

# log messages from authpriv facility(sensitive nature)
#  to the /var/log/secure file. But also use the

# template (myTraditionalFormat) defined earlier in
# the file
authpriv.*   /var/log/secure;    myTraditionalFormat
```

```
# Log all the mail messages in one place.
mail.*    -/var/log/maillog

# Send emergency messages to all logged on users
*.emerg      *

# Following is an entry that logs to a database host at the IP address
# 192.168.1.50 into DB named log_database
*.*             >192.168.1.50,log_database,dude,dude_db_password
```


systemd-journald (journald)

systemd-journald is a component of **systemd** (discussed earlier in this chapter). It runs as a system service that collects and stores logging data. Just like **rsyslogd**, it has the ability to receive logging data from various sources: user processes, the kernel, and the standard output (**STDOUT**) and standard error (**STDERR**) of other system services.

journald creates and maintains its logging data differently from the traditional method of using flat files. It natively stores logging data in structured and indexed journals, which lends itself to lots of rich new possibilities and speed improvements.

Most of the mainstream RPM- and Debian-based distributions have integrated **systemd-journald** for use alongside the more traditional logging daemons such as rsyslogd. It is possible to run both solutions concurrently on a system and even feed data from one solution to the other. Even though this is possible, it is generally considered good system administration practice to standardize on a single logging solution on production systems.

journalctl and journalctl Examples

The primary tool for interacting with the journal files collected and maintained by **journald** is called **journalctl**. It serves as the native utility for querying and viewing the contents of logging data collected by **systemd-journald**.

We will walk through some quick **journalctl** uses in the following exercise.

TIP The output for **journalctl** is helpfully colored and bolded when viewed at the terminal. The output lines are colored and bolded according to priorities like this:

Lines of level ERROR (3) and higher (CRIT [2], ALERT [1], and EMERG [0]) are colored red.

Lines of level NOTICE (5) and higher (WARNING [4], CRIT [2], ALERT [1], and EMERG [0]) are highlighted in color and bold.

Lines of level DEBUG (7) and higher (INFO [6]) are displayed normally.

1. Use **journalctl** in its simplest form by running it without any options to view the journal logs:

   ```
   [masterserver ~]$  journalctl
   ```

   Use your keyboard ARROW keys to scroll through the messages or press Q to quit the utility and stop viewing the log file.

2. Display two of the most recent journal logs by typing the following:

```
[masterserver ~]$   journalctl -n 2
```

3. Display a much more detailed (or verbose) version of the previous command:

```
[master@server ~]$   journalctl -o verbose -n 2
```

   This will show some of the metadata collected by **journalctl**.

4. View the log messages in real time as they are being generated and collected:

```
[master@server ~]$   journalctl  -f
```

   Press CTRL-C to quit viewing the live output.

5. View log messages since six days ago:

```
[master@server ~]$   journalctl   --since="6 days ago"
```

6. View journal messages generated between six days ago and four days ago:

```
[master@server ~]$   journalctl   --since="6 days ago" \
 --until="4 days ago"
```

7. It's the year 2082, and it's the 25th day of December on planet *Walenium,* where our server runs, and we want to view log messages only from five days ago:

```
[master@server ~]$  journalctl --since=2082-12-20
```

8. Let's narrow down the date and time range from the previous commands to show the journal entries that were created between 1 P.M. and 1:30 P.M. on the same date:

```
[master@server ~]$ journalctl --since="2082-12-20 13:00" \
--until="2082-12-20 13:30"
```

9. View all the journal entries created since 2082-12-20 at 1 P.M. with a priority warning or higher:

```
[master@server ~]$ journalctl --since="2082-12-20 13:00" -p warning
```

10. View log messages generated since 2082-12-20 at 7 A.M. that exactly match the CRITICAL (2) priority:

```
[master@server ~]$ journalctl  PRIORITY=2  --since="2082-12-20 07:00"
```

11. The user ID for the user master on our sample system is 1000. View all log messages that are associated with this user with UID=1000:

```
[master@server ~]$ journalctl  _UID=1000
```

12. Combine the previous two commands to view messages with critical (6) priority, generated by the user with a UID of 1000 since 2082-12-20 at 7 A.M.:

```
[master@server ~]$ journalctl _UID=1000  PRIORITY=6 \
 --since="2082-12-20 07:00"
```

13. View all system log messages generated by the **systemd** service unit **dbus.service**:

```
[master@server ~]$ journalctl  -u dbus
```

14. View log messages emitted by the NetworkManager program on your server:

```
[master@server ~]$  journalctl  /usr/sbin/NetworkManager
```

15. Check to see if there are any messages caused by the **/dev/sda** or **/dev/vda** hard disk device installed on your server:

```
[master@server ~]$ journalctl  /dev/[vs]da
```

16. Finally, view all the log messages of priority ERROR (3) and higher (that is, worse) from the current boot session of the server:

```
[master@server ~]$ journalctl -b -p err
```


The cron Program

The **cron** program allows any user in the system to schedule a program to run on any date, at any time, or on a particular day of the week, down to the minute. Using **cron** is an extremely efficient way to automate tasks on your system, generate reports on a regular basis, and perform other periodic chores such as backups and e-mail reminders. (A not-so-nice use of **cron** is to have it invoke a system to have you paged when you want to get out of a meeting!)

Like the other services we've discussed in this chapter, **cron** is automatically started by the boot scripts and is most likely already configured for you. A quick check of the process listing should show it quietly running in the background:

```
[master@server ~]$  ps aux | grep crond | grep -v grep
root    723  0.0  0.1 1248  3332 ?    Ss   Dec26   0:02 /usr/sbin/crond -n
```

The **cron** service works by waking up once a minute and checking each user's **crontab** file. This file contains the user's list of events that they want executed at a particular date and time. Any events that match the current date and time are executed.

The crontab File

The tool that allows you to edit entries to be executed by **crond** is **crontab**. Essentially, all it does is verify your permission to modify your **cron** settings and then invoke a text editor so you can make your changes. Once you're done, **crontab** places the file in the right location and brings you back to a prompt.

Whether or not you have appropriate permission is determined by **crontab** by checking the **/etc/cron.allow** and **/etc/cron.deny** files. If either of these files exists, you must be explicitly listed there for your actions to be affected. For example, if the **/etc/cron.allow** file exists, your username must be listed in that file in order for you to be able to edit your **cron** entries. On the other hand, if the only file that exists is **/etc/cron.deny**, unless your username is listed there, you are implicitly allowed to edit your cron settings.

The file listing your **cron** jobs (often referred to as the **crontab** file) is formatted as follows (all values must be listed as integers):

```
Minute Hour Day Month Day_Of_Week Command
```

If you want to have multiple entries for a particular column (for instance, you want a program to run at 4:00 A.M., 12:00 P.M., and 5:00 P.M.), then you need to include each of these time values in a comma-separated list. Be sure not to type any spaces in the list. For the program running at 4:00 A.M., 12:00 P.M., and 5:00 P.M., the *Hour* values list would read **4,12,17**. Newer versions of **cron** allow you to use a shorter notation for the fields. For example, if you want to run a process every two minutes, you just need to put **/2** as the first entry. Notice that **cron** uses military time format.

For the *Day_Of_Week* field, 0 represents Sunday, 1 represents Monday, and so on, all the way up to 6, representing Saturday.

Any entry that has a single asterisk (*) wildcard will match any minute, hour, day, month, or day of the week when used in the corresponding column.

When the dates and times in the file match the current date and time, the command is run as the user who set the **crontab**. Any output generated is e-mailed back to the user. Obviously, this can result in a mailbox full of messages, so it is important to be thrifty with your reporting. A good way to keep a handle on volume is to output only error conditions and have any unavoidable output sent to **/dev/null**.

Let's look at some examples. The following entry runs the **ping** program (**/bin/ping)** with the command-line argument **-c 5 server-B** every four hours:

```
0  0,4,8,12,16,20  *  *  *  /bin/ping -c 5 server-B
```

Here's the same command using the shorthand method:

```
0 */4 * * * /bin/ping -c 5 server-B
```

Here is an entry that runs the custom program **/usr/local/scripts/backup_level_0** at 10:00 P.M. every Friday night:

```
0 22 * * 5 /usr/local/scripts/backup_level_0
```

And finally, here's a script to send out an e-mail at 4:01 A.M. on April 1 (whatever day that might be):

```
1 4 1 4 * /bin/mail mom@domain.com < /home/yyang/joke
```

NOTE When **crond** executes commands, it does so with the **sh** shell. Thus, some environment variables that you might be accustomed to might not work within **cron**. This is why you should always test your **cron** jobs before putting them into production.

Editing the crontab File

Editing or creating a **cron** job is as easy as editing a regular text file. But you should be aware of the fact that the program will, by default, use an editor specified by the **EDITOR** or **VISUAL** environment variable. On most Linux systems, the default editor is **vi**. But you can always

change this default to any editor you are comfortable with by setting the **EDITOR** or **VISUAL** environment variable.

Now that you know the format of the **crontab** configuration file, you need to edit the file. You don't do this by editing the file directly; instead, you use the **crontab** command to edit your **crontab** file:

```
yyang@server:~$  crontab -e
```

To list what is in your current **crontab** file, just give **crontab** the **-l** argument, like so:

```
yyang@server:~$  crontab -l
no crontab for yyang
```

According to this output, the user *yyang* does not currently have anything in **cron**.

TIP In it's true Jack-of-all-trades fashion, **systemd** provides an almost functional replacement to the venerable **cron** service. This is done via the **systemd.timer** unit object. This object stores information about a *timer construct,* which is controlled and managed by **systemd**.

So, for example, to create an ad-hoc **.timer** unit job that will execute a sample **ping** command in 45 seconds, you would type the following:

```
$ sudo systemd-run --on-active=45  ping  -c 5 server-B
```

And to create another sample transient **.timer** unit job that will start an existing service unit called **carpald.service** in 11 hours and 24 minutes, you could run this:

```
$ sudo  systemd-run --on-active="11h 24m" --unit carpald.service
```

View the status of the timer for the **carpald.service** unit that was created as follows:

```
$ sudo  systemctl list-timers carpald.timer
```

Summary

In this chapter, we discussed some important system services (**init, upstart, systemd, inetd, xinetd, rsyslog, systemd-journald** [**journald**], and **cron**) that come with most Linux systems. These services contribute to maintaining and monitoring the overall health of servers. They are relevant and useful on a server running in stand-alone mode, single-user mode, multiuser mode, networked mode, and so on. Their implementation can vary slightly from host to host or from distro to distro, but the core objectives and the problem they aim to solve remain the same.

In each section of this chapter, we discussed how to configure some core system services. Try poking around these services and familiarize yourself with what you can accomplish with them. Many powerful automation, data collection, and analysis tools have been built around these basic services—as well as many wonderfully silly and useless things. Have fun with them!

This page intentionally left blank

CHAPTER

10 The Linux Kernel

One of Linux's greatest strengths is that its source code is available to anyone who wants it. The GNU GPL (General Public License) under which Linux is distributed even allows you to tinker with the source code and distribute your changes! Real changes to the source code (at least, those to be taken seriously) go through the process of joining the official kernel tree. This requires extensive testing and proof that the changes will benefit Linux as a whole. At the end of the approval process, the code gets a final yes or no from a core group of trusted Linux kernel developers. It is this extensive review process that keeps the quality of Linux's code so noteworthy.

For system administrators who have used other proprietary operating systems, this approach to code control is a significant departure from the philosophy of waiting for "the company" to release a patch, a service pack, or some sort of hotfix.

Instead of having to wade through public relations, customer service, sales engineers, and other front-end units behind a proprietary operating system, in the Linux world you have the option of contacting the author of a kernel subsystem directly and explaining your problem. A patch can be created and sent to you before the next official release of the kernel to get you up and running.

Of course, the flip side of this working arrangement is that you need to be able to compile a kernel yourself rather than rely on someone else to supply precompiled code. Thankfully the instances when you need to do this are usually few and far between. Also, whenever it's *necessary,* you should know what to do and the reasons for doing it. Luckily, it's not difficult.

In this chapter, we'll walk through the process of acquiring a kernel source tree, configuring it, compiling it, and, finally, installing and booting the kernel.

What Exactly Is a Kernel?

Before we jump into the process of compiling, let's back up a notch and make sure you're clear on the concept of what a kernel is and the role it plays in the system. Most often, when people say "Linux," they are usually referring to a "Linux distribution"—for example, Debian is a type of Linux distribution. As discussed in Chapter 1, a distribution comprises everything

necessary to get Linux to exist as a functional operating system. Distributions make use of code from various open source projects that are independent of Linux; in fact, many of the software packages maintained by these projects are used extensively on other UNIX-like platforms as well. The GNU C Compiler, for example, which comes with most Linux distributions, also exists on many other operating systems (probably more systems than most people realize).

So, then, what is the pure definition of Linux? It's *the kernel.* The kernel of any operating system is the core of all the system's software. The kernel literarily sits *right* at the *heart* of the (operating system) matter. The only thing more fundamental than the kernel is the system hardware itself.

The kernel has many jobs. The essence of its work is to abstract the underlying hardware from the software and provide a running environment for application software through system calls. Specifically, the environment must handle issues such as networking, disk access, virtual memory, and multitasking—a complete list of these tasks would take up an entire chapter in itself!

Today's Linux kernel, version 5.* (where the asterisk is a wildcard that represents the complete version number of the kernel), contains more than 13 million lines of code (including device drivers). By comparison, the sixth edition of UNIX from Bell Labs in 1976 had roughly 9000 lines. Figure 10-1 illustrates the kernel's position in a complete system.

Although the kernel is a small part of a complete Linux distribution, it is by far the most critical element. If the kernel fails or crashes, the rest of the system goes with it. You need have no fear, however, because the stability of the Linux kernel is nothing short of *legendary*—so much so that *uptimes* (the length of time in between reboots) for Linux systems are often expressed in years!

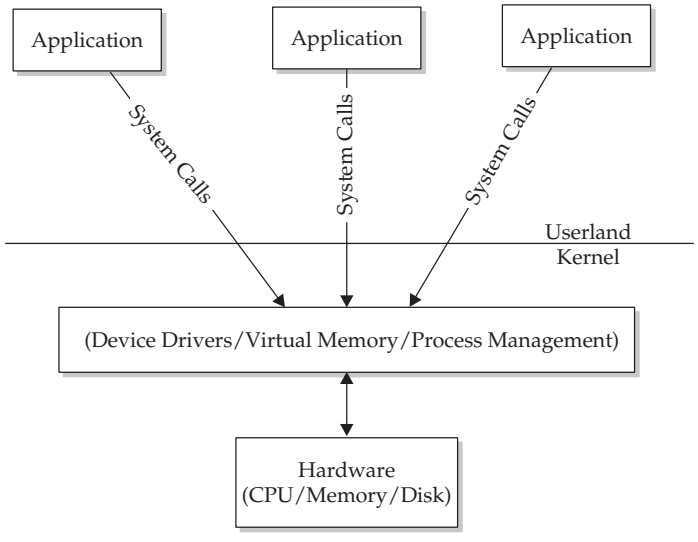


Figure 10-1 A visual representation of how the Linux kernel fits into a complete system

CAUTION The kernel is the first thing that loads when a Linux system is booted—after the boot loader, of course! If the kernel doesn't work right, it's unlikely that the rest of the system will boot.

Finding the Kernel Source Code

Your distribution of Linux probably has the source code for the specific kernel version it supports available in one form or another. These could be in the form of a compiled binary (*.src.rpm), a source RPM (*.srpm), or the like.

If you need to download a different (possibly newer) version than the one your particular Linux distribution provides, the first place to look for the source code is at the official kernel web site: www.kernel.org. This site maintains a listing of web sites mirroring the kernel source, as well as tons of other open source software, distributions, and general-purpose utilities. The list of mirrors is maintained at http://mirrors.kernel.org.

TIP The majority of the downloading, configuring, and compiling of the Linux kernel done in the following sections can/should be done as an unprivileged user. The final steps that require actual installation or altering of system files and binaries need to be done with elevated privileges.

To sidestep appending privileged commands with **sudo** as we've done for the following exercises, you can/should do most of the work as an unprivileged user by specifying an "output" directory using the "`O`" option for *all* invocations of **`make`**. For example, to work in a folder (**build/kernel**) under your home directory (for example, **/home/master**), run the following:

```
$ make   O=~/build/kernel  menuconfig
```

Getting the Correct Kernel Version

The web site listing of kernels available will contain folders for v1.0, v2.5, v2.6, v3.0, v3.x, v4.x, v5.x, v6.x, and so forth. Before you follow your natural inclination to get the latest version, make sure you understand how the Linux kernel versioning system works.

The current convention is to name and number major new kernel releases as "Linux 5.*x*" (also called the *vanilla* or *mainline* kernels). Thus, the first of this series will be Linux version 5.0 (same as 5.0.0), the next will be Linux version 5.1 (same as 5.1.0), followed by Linux version 5.2, and so on.

But *wait*, there's *more*—any minor changes or updates within each major release version will be reflected by increments to the third digit. These are commonly referred to as *stable point releases*. Thus, the next stable point release for the 5.0.0 series kernel will be Linux version 5.0.1, followed by version 5.0.2, and so forth. Another way of stating this is to say, for example, that Linux version 5.0.4 will be the fourth stable release based on the Linux 5.0.0 series.

The version of the kernel that we are going to build in the following section is version **5.4.10**, which is available at www.kernel.org/pub/linux/kernel/v5.x/linux-5.4.10.tar.xz.

TIP You can use the **`wget`** utility to download the kernel source quickly into your current working directory by typing the following:

```
# wget  https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.4.10.tar.xz
```

Unpacking the Kernel Source Code

Most of the software packages you have dealt with so far have probably been Red Hat Package Manager (RPM) or .deb packages, and you're most likely accustomed to using the tools that came with the system (such as RPM, Advanced Packaging Tool [APT], Yum, DNF, or YaST) to manage the packages. Kernel source code is a little different and requires some user interaction.

The kernel source consists of a bunch of different files, and because of the sheer number and size of these files collectively, it is useful to compress the files and put them all in a single directory structure. The kernel source that you will download from the Internet is a file that has been compressed and tarred. Therefore, to use the source, you need to decompress and untar the source file. This is what it means to "unpack the kernel." Overall, it's really a straightforward process. The traditional location for the kernel source tree on the local file system is the **/usr/src** directory. For the remainder of this chapter, we'll assume you are working out of the **/usr/src** directory.

NOTE Some Linux distributions have a symbolic link under the **/usr/src** directory. This symbolic link is usually named "linux" and is usually a link to a default or the latest kernel source tree. Some third-party software packages rely on this link to compile or build properly!

Let's unpack the kernel. First, copy the kernel tarball that you downloaded earlier into the **/usr/src** directory:

```
[master@server ~]$ sudo cp linux-5.*.xz  /usr/src/
```

Change your working directory to the **/usr/src/** directory and use the **tar** command to unpack and decompress the file:

```
[master@server ~]$  cd /usr/src/  && sudo tar xvJf linux-5.*.tar.xz
```

Building the Kernel

So now you have an unpacked kernel tree just waiting to be built. In this section, we're going to review the process of configuring and building a kernel. This is in contrast to Windows-based operating systems, which come preconfigured and therefore contain support for many features you may or may not want.

The Linux design philosophy allows the individual to decide on the important parts of the kernel. This individualized design has the important benefit of letting you thin down the feature list so that Linux can run as efficiently as possible. This is also one of the reasons why it is possible to run Linux in various hardware setups, from low-end systems, to embedded systems, to high-end systems. You may find that a box incapable of supporting a Windows-based server is more than capable of supporting a Linux-based OS.

Two main steps are required in building a kernel: *configuring* and *compiling*. We won't get into the specifics of configuration in this chapter, which would be difficult because of the fast-paced evolution of the Linux kernel. However, once you understand the basic process, you should be able to apply it from version to version. For the sake of this discussion, we'll cite examples from the v5.* kernel that we unpacked in the previous section.

The first step in building the kernel is configuring (or customizing) its features. Usually, your desired feature list will be based on whatever hardware you need to support. This, of course, means that you'll need a list of that hardware.

On a system that is already running Linux, you can run commands like **`lspci`**, **`lshw`**, and so on to help show detailed information about the exact hardware setup on your system. On RPM-based distros, these utilities are provided by the **pciutils*.rpm** and **lshw*.rpm** packages. Having a better understanding of what constitutes your underlying hardware can help you better determine what you need in your custom kernel. You're ready to start configuring the kernel.

Preparing to Configure the Kernel

With a rough idea of the types of hardware and features that our new kernel needs to support, we can begin the actual configuration. But first, some background information.

The Linux kernel source tree contains several files named **Makefile** (a makefile is simply a text file with directives, and it also describes the relationships among the files in a program). These makefiles help to glue together the thousands of other files that make up the kernel source. What is more important to us here is that the makefiles also contain *targets,* which are the commands, or directives, executed by the **`make`** program.

Avoid Needless Upgrades

Bear in mind that if you have a working system that is stable and well behaved, there is little reason to upgrade the kernel unless one of these conditions holds for you:

- A security or bug fix affects your system and must be applied.
- You need a specific new feature in a stable release.

In the case of a security fix, decide whether the risk really affects you—for example, if the security issue is found in a device driver that you don't use, then there may be no reason to upgrade. In the case of a bug fix release, read carefully through the release notes and decide if the bugs really affect you—if you have a stable system, upgrading the kernel with patches you never use may be pointless. On production systems, the kernel shouldn't simply be upgraded just to have "the latest kernel." Instead, you should have a *truly compelling reason* to upgrade.

The **Makefile** in the root of the kernel source tree contains specific targets that can be used in prepping the kernel build environment, configuring the kernel, compiling the kernel, installing the kernel, and so on. Some of the targets are discussed in more detail here:

- **`make mrproper`** This target cleans up the build environment of any stale files and dependencies that might have been left over from a previous kernel build. All previous kernel configurations will be cleaned (deleted) from the build environment.
- **`make clean`** This target does not do as thorough a job as the **`mrproper`** target. It deletes only most generated files. It does not delete the kernel configuration file (**.config**).

- **`make menuconfig`** This target invokes a text-based editor interface with menus, option lists, and text-based dialog boxes for configuring the kernel.
- **`make xconfig`** This is a GUI-based kernel configuration tool/target that relies on the Qt graphical development libraries. These libraries are used by KDE/Plasma-based applications.
- **`make gconfig`** This is also a GUI-based kernel configuration tool/target, but it relies on the GTK+ toolkit. This GTK toolkit is heavily used in the GNOME desktop world.
- **`make olddefconfig`** This target uses the existing **.config** file in the current working directory, updates the dependencies, and automatically sets new symbols to their default values.
- **`make help`** This target will show you all the other possible **make** targets and also serves as a quick online help system.

To configure the kernel in this section, we will use only one of the targets. In particular, we will use the **`make menuconfig`** command. The **`menuconfig`** kernel config editor is a simple and popular text-based configuration utility that consists of menus, radio button lists, and dialog boxes. It has a simple and clean interface that can be easily navigated with your keyboard and is *almost* intuitive to use. We need to change (**`cd`**) into the kernel source directory, after which we can begin the kernel configuration. But before beginning the actual kernel configuration, you should clean (prepare) the kernel build environment by using the **`make mrproper`** command:

```
[master@server ~]$  cd linux-5.*
[master@server ~]$ sudo make  mrproper
```

Kernel Configuration

Next, we will step through the process of configuring a Linux 5.* series kernel. To explore some of the innards of this process, we will enable the support of a specific feature that we'll *pretend* must be supported on the system. Once you understand how this works, you can apply the same procedure to add support for any other new kernel feature that you want. Specifically, we'll enable support for the NTFS file system into our custom kernel.

Most modern Linux distros that ship with the 5.* series kernels (remember that the asterisk symbol is a wildcard that represents the complete version number of the kernel) also have a kernel configuration file for the running kernel available on the local file system as a compressed or regular file. On our sample system that runs the Fedora distro, this file resides in the **/boot** directory and is usually named something like **config-5.***. The configuration file contains a list of the options and features that were enabled for the particular kernel it represents. A config file similar to this one is what we aim to create through the process of configuring the kernel. The only difference between the file we'll create and the ready-made one is that we will add further minor customizations to ours.

TIP Using a known, preexisting config file as a framework for creating our own custom file helps ensure that we don't waste too much time duplicating the efforts that other people have already put into finding what works and what doesn't work!

The following steps cover how to configure the kernel. We will be using a text-based kernel configuration utility, which will allow you to follow along in your terminal regardless of whether you are using a GUI desktop environment or not.

1. To begin, we'll copy over and rename the preexisting config file from the **/boot** directory into our kernel build environment:

   ```
   [master@server linux-5.*]$ sudo cp  /boot/config-`uname -r` .config
   ```

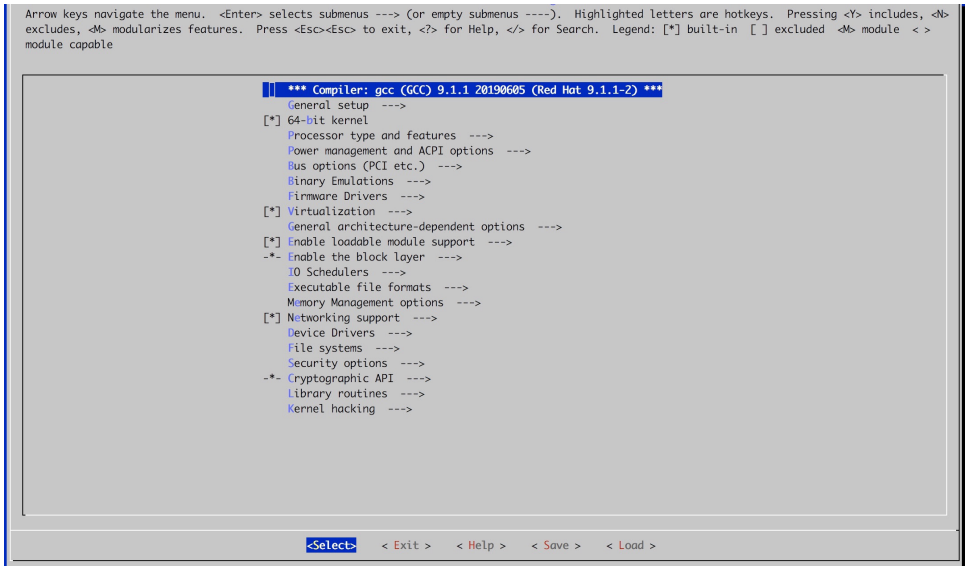
   We use **`uname -r`** here to help us obtain the configuration file for the running kernel. The **uname -r** command prints the running kernel's release. Using it helps ensure that we are getting the exact version we want, just in case other versions are present.

NOTE The Linux kernel configuration editor specifically starts up looking for, and ends up generating, a file named **.config** (pronounced "dot config") at the root of the kernel source tree. This file is hidden.

2. Launch the graphical kernel configuration utility:

   ```
   [master@server linux-5.*]$ sudo  make menuconfig
   ```

   A screen similar to this will appear:



TIP If the **make** command throws an error about being unable to find an ncurses package, you can resolve this by using the DNF package manager on a Red Hat–like distro to install a package named **ncurses-devel** to satisfy this requirement, like so:

```
$ sudo dnf -y install ncurses-devel
```

On a Debian-based distro like Ubuntu, you can install the equivalent tool by running the following command:

```
$ sudo apt-get -y  install libncurses-dev
```

Part II

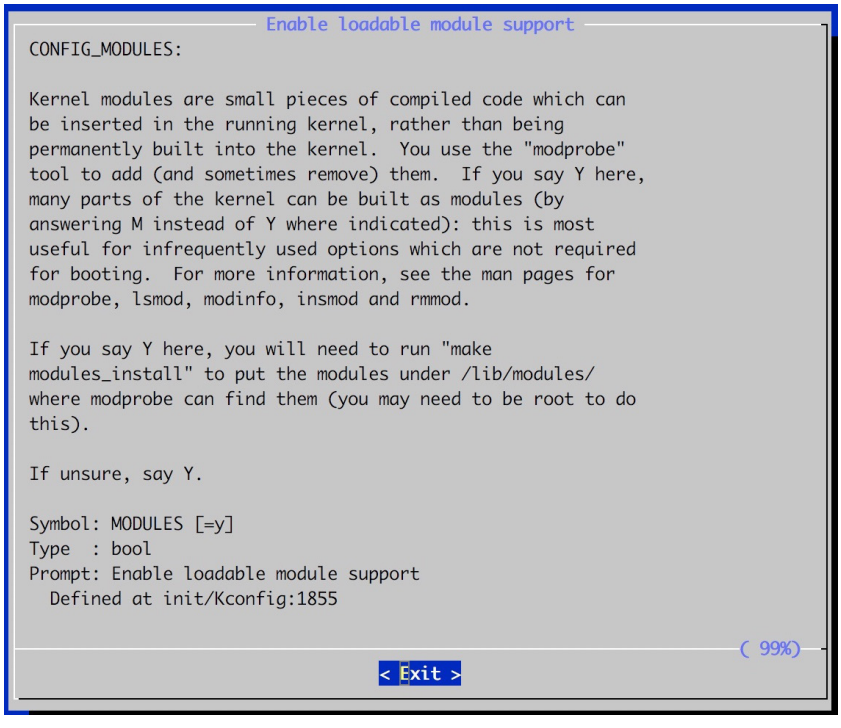
The kernel configuration screen that appears is divided into roughly three areas:

- The top part shows various helpful information, keyboard shortcuts, and legends that can help you navigate the application.
- The main body of the screen shows an expandable tree-structured list of the overall configurable kernel options. You can further drill down into items with arrows in the parent to view and/or configure submenu (or child) items.
- Finally, the bottom of the screen displays the actual actions/options you can choose.

3. Just for fun, we will examine one very important option a little more closely by selecting it in the main body of the application and consulting the Help menu for it.

   Use the ARROW keys on your keyboard to navigate to (or highlight) the Enable Loadable Module Support item. Once it is highlighted, use the left/right ARROW keys or the TAB key to select <Help> at the bottom of the screen and then press ENTER to execute the Help action on the highlighted entry.

   Now study the inline help information that appears in the ensuing screen, as shown in the following illustration:



Once you are done looking over the Help screen for the CONFIG_MODULES kernel configuration option, make sure Exit is selected at the bottom of the screen and then press ENTER to return to the main area.

4. Next, for demonstration purposes, we'll add support for NTFS into our custom kernel.

   While at the main configuration screen, use your ARROW keys to navigate to and highlight the File Systems item. With File Systems selected, press ENTER to view the submenu or child items for File Systems.

   In the File Systems section, use your ARROW keys to navigate to DOS/FAT/NT Filesystems. Press ENTER to see the child items for DOS/FAT/NT Filesystems.

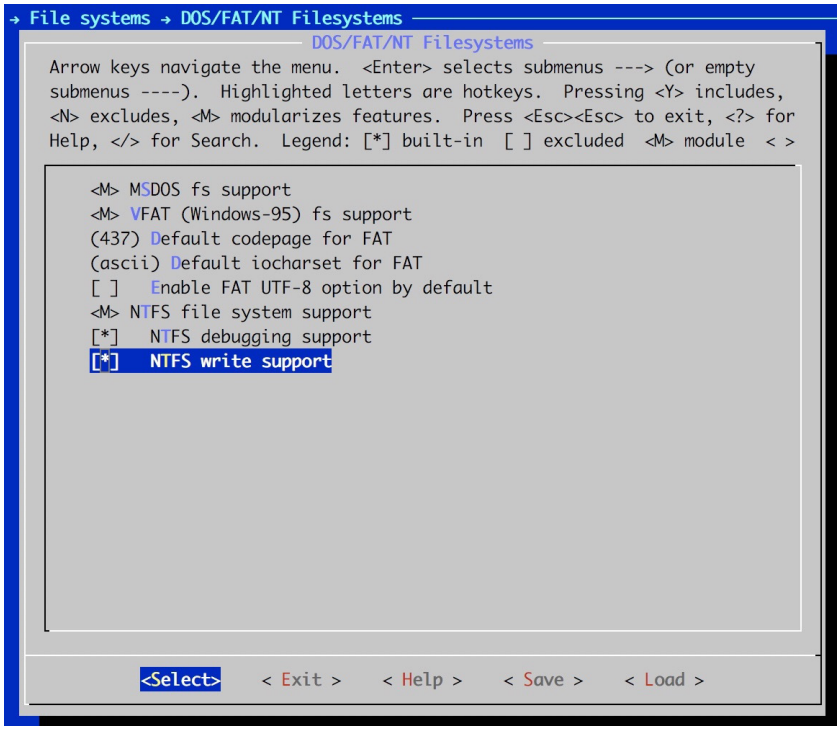
5. In the DOS/FAT/NT Filesystems section, navigate to NTFS File System Support.

   Type **M** (uppercase) to enable support for the NTFS file system via modules.

   Use the ARROW keys to navigate down to NTFS Debugging Support (NEW) and then press Y to include it.

   Use the ARROW keys to navigate down to NTFS Write Support and then press Y to include it.

   When you are done, the letter *M* or an asterisk symbol (*) should appear beside each option, like the ones shown here:



TIP For each of the configurable options, in the kernel configuration utility, empty angle brackets (<>) indicate that the feature in question is disabled. The letter *M* in angle brackets (<M>) indicates that the feature is to be compiled as a module. And the asterisk in angle brackets (<*>) indicates that support for the feature will be directly built into the kernel. You can usually toggle through all the possible options using the SPACEBAR on your keyboard.

6. Navigate back to the parent File Systems screen by pressing the ESC key twice on your keyboard in the DOS/FAT/NT Filesystems screen.

   Return to the main kernel configuration screen by pressing ESC twice again on your keyboard.
7. Save your changes to the **.config** file in the root of your kernel source tree and exit the kernel configuration application after saving the file by pressing ESC twice again on your keyboard.
8. A dialog box will appear prompting you to save your new configuration. Make sure that Yes is selected and then press ENTER.
9. After the kernel configuration utility exits, you will be thrown back to your shell—inside the kernel source tree.

   You are ready to build your kernel!

TIP To view the results of some of the changes you made using the **menuconfig** tool, use the **grep** utility to view the **.config** file you saved directly. For example, to view the effect of the NTFS file system support we enabled previously, type the following:

```
[master@server linux-5.*]$  grep -i ntfs  .config
CONFIG_NTFS_FS=m
CONFIG_NTFS_DEBUG=y
CONFIG_NTFS_RW=y
```

A Quick Note on Kernel Modules

Loadable module support is a Linux kernel feature that allows for the dynamic loading (or removal) of kernel modules. Kernel modules are pieces of compiled code that can be dynamically inserted into the running kernel, rather than being permanently built into the kernel. Features not often used can thus be enabled, but they won't occupy any room in memory when they aren't being used.

Thankfully, the Linux kernel can automatically determine what to load and when. Naturally, not every feature is eligible to be compiled as a module. The kernel must know a few things before it can load and unload modules, such as how to access the hard disk and parse through the file system where the loadable modules are stored. Some kernel modules are also commonly referred to as *drivers*.

Compiling the Kernel

In the preceding section, we walked through the process of creating a configuration file for the custom kernel that we want to build. In this section, we will perform the actual build of the kernel. But before doing this, we will add one more simple customization to the entire process.

The final customization will be to add an extra piece of information used in the final name of our kernel. This will help us be able to differentiate *this* kernel from any other kernel with the same version number. We will add the tag "custom" to the kernel version information. This can be done by editing the main **Makefile** and appending the tag that we want to the **`EXTRAVERSION`** variable.

The compilation stage of the kernel-building process is by far the easiest, but it also takes the most time. All that is needed at this point is simply to execute the **`make`** command, which will then automatically generate and take care of any dependency issues, compile the kernel itself, and compile any features (or drivers) that were enabled as loadable modules.

Because of the amount of code that needs to be compiled, be prepared to wait a few minutes, at the very least, depending on the processing power of your system. Let's dig into the specific steps required to compile your new kernel:

1. We'll add an extra piece to the identification string for the kernel we are about to build. While still in the root of the kernel source tree, open up the **Makefile** for editing with any text editor. The variable we want to change is close to the top of the file. Change the line in the file that looks like

   ```
   EXTRAVERSION =
   ```

   to this:

   ```
   EXTRAVERSION = -custom
   ```

2. Save your changes to the file and exit the text editor.

TIP In the unlikely event that you don't have a simple text editor readily available, you can use the **`sed`** utility to quickly search and make the changes to the **Makefile** in the previous step for you by running the following command:

```
# sed -i 's/^EXTRAVERSION.*/EXTRAVERSION = -custom/' Makefile
```

---

3. Pass the **`kernelversion`** target to the **`make`** command to view the full version of the kernel you just customized:

   ```
   [master@server linux-5.*]$ make  kernelversion
   ```

TIP You can take advantage of all that extra processing power (CPUs, cores, and so on) on most modern systems and greatly speed up CPU-intensive operations like compiling the kernel. To do this, you can pass a parameter to the **`make`** command that specifies the number of jobs to run simultaneously. The specified number of jobs are then distributed and executed simultaneously on each CPU core. The syntax for the command is

```
# make -j N
```

where *N* is the number of jobs to run simultaneously. For example, if you have a octa (8) core–capable CPU, you can type this:

```
# make -j 8
```

4. The only command that is needed here to compile the kernel is the **make** command:

```
[master@server linux-5.*]$ sudo make
HOSTLD  scripts/kconfig/conf
...<OUTPUT TRUNCATED>...
CC [M]  drivers/usb/wusbcore/security.o
LD      drivers/built-in.o
```

TIP A common source of failure encountered during the kernel build process may be caused by not having all the requisite software available for compiling. On a Red Hat–like distro such as Fedora, you can quickly get all the necessary development tools installed by running this command:

```
# dnf -y groupinstall 'C Development Tools and Libraries'
```

Some other pertinent libraries and header files that you might need can also be obtained by installing the following packages on an RPM-based distro, like so:

```
# dnf -y install openssl-devel elfutils-libelf-devel
```

5. The end product of this command (that is, the kernel) is sitting pretty and waiting in the path:

```
<kernel-source-tree>/arch/x86/boot/bzImage
```

6. Because we compiled portions of the kernel as modules (for example, the NTFS module), we need to install the modules. Type the following:

```
[master@server linux-5.*]$ sudo make modules_install
```

On a Fedora system, this command will install all the compiled kernel modules into the **/lib/modules/**<***new_kernel-version***> directory. In this example, this path will translate to **/lib/modules/5.4.10-custom/**. This is the path from which the kernel will load all loadable modules, as needed.

TIP The footprint (size) of the kernel modules installed via **make modules_install** is pretty large because the modules include debugging symbols. On our sample system, the size of the **/lib/modules/5.4.10-custom/** directory is over 4GB! You can reduce the total size by orders of magnitude (for example, less than 200MB!) by stripping away these debugging symbols. This can be done by including the **INSTALL_MOD_STRIP=1** option to the **make modules_install** command, like so:

```
$ sudo make INSTALL_MOD_STRIP=1 modules_install
```

Installing the Kernel

So now you have a fully compiled kernel just waiting to be installed. You probably have a couple of questions: Just where *is* the compiled kernel, and where the heck do I install it?

The first question is easy to answer. Assuming you have a PC and are working out of the **/usr/src/**<***kernel-source-tree***>**/** directory, the compiled kernel that was created in the previous exercise will be called **/usr/src/**<***kernel-source-tree***>**/arch/x86/boot/bzImage** (in our example, it's **/usr/src/linux-5.4.10/arch/x86/boot/bzImage**).

The corresponding map file for this will be located at **/usr/src/<*kernel-source-tree*>/System.map**. You'll need both files for the install phase.

The **System.map** file is useful when the kernel is misbehaving and generating "oops" messages. An "oops" is generated on some kernel errors due to kernel bugs or faulty hardware. This error is akin to the Blue Screen of Death (BSOD) in Microsoft Windows. These messages include a lot of detail about the current state of the system, including several hexadecimal numbers. **System.map** gives Linux a chance to turn those hexadecimal numbers into readable names, making debugging easier. Although this is mostly for the benefit of developers, it can be handy when you're reporting a problem.

Let's go through the steps required to install the new kernel image:

1. While in the root of your kernel build directory, copy and rename the **bzImage** file into the **/boot** directory:

   ```
   [master@server linux-5.*]$ sudo cp arch/x86/boot/bzImage  \
   /boot/vmlinuz-<kernel-version>
   ```

   Here, ***kernel-version*** is the version number of the kernel. For the sample kernel we are using in this exercise, the filename would be **vmlinuz-5.4.10 -custom**. So here's the exact command for this example:

   ```
   [master@server linux-5.*]$  sudo cp arch/x86/boot/bzImage  \
   /boot/vmlinuz-5.4.10-custom
   ```

NOTE The decision to name the kernel image **vmlinuz-5.4.10-custom** is somewhat arbitrary. It's convenient, because kernel images are commonly referred to as **vmlinuz**, and the suffix of the version number is useful when you have multiple kernels available or kernels that provide specific functionality (for example, **vmlinuz-6.50.0-wireless**).

2. Now that the kernel image is in place, copy over and rename the corresponding **System.map** file into the **/boot** directory using the same naming convention:

   ```
   [master@server linux-5.*]$ sudo cp -v  System.map \
    /boot/System.map-5.4.10-custom
   ```

3. With the kernel in place, the **System.map** file in place, and the modules in place, we are now ready for the final step. Here is the syntax for the command needed:

   ```
   # kernel-install add   <kernel-version>  <kernel-image>
   ```

   Here, <**kernel-version**> is the version number (and name) of the kernel, and **<kernel-image>** is the path to the newly compiled kernel image. For our example, type the following:

   ```
   [master@server linux-5.*]$ sudo kernel-install \
   add  5.4.10-custom /boot/vmlinuz-5.4.10-custom
   ```

The **kernel-install** command used here is a nifty little shell script. It might not be available in every Linux distribution, but it is available in newer Fedora, RHEL, and CentOS distros. A similar utility that you might find on older distros is the **new-kernel-pkg** script. Both of these tools automates a lot of the final manual things we'd ordinarily have to do to set up the system to boot the new kernel we just built. In particular, the tool does the following:

- It creates the appropriate initial RAM file system image (the **initramfs** image—that is, the **/boot/initramfs-<*kernel-version*>.img** file).

  To do this manually on systems where **kernel-install** or **new-kernel-pkg** is not available, use the **mkinitramfs** command.
- It runs the **depmod** command (which creates a list of module dependencies).
- It updates the boot loader configuration.

For systems running the newer versions of GRUB2, the file will be **/boot/grub2/grub.cfg**. For EFI-based systems, **/boot/efi/<distro>/fedora/grub.cfg** is also updated, and for systems running the legacy versions of GRUB, this will be the **/boot/grub/grub.conf** or **/boot/grub/menu.lst** file. For very new distros that have implemented the new Boot Loader Specification (BLS), a new boot loader entry will be added to the **/boot/loader/entries/** directory or any directory pointed to by the variable named "**blsdir**".

On our demo EFI-based Fedora server running GRUB 2 using BLS, a new boot entry is created in the boot loader file located here:

```
/boot/loader/entries/5dff0c3c00c14d3988d5e09b5ffacd90-5.4.10-custom.conf
```

NOTE Most distros have several **grub2-*** utilities readily available that can be used for performing various GRUB2 and boot loader housekeeping tasks. For example, you can use the **grub2-set-default** command to change or set the default kernel to be booted at system startup.

Booting the Kernel

The next stage is to test the new kernel to make sure that the system can indeed boot with it:

1. Assuming you did everything the exact way the doctor prescribed, and everything worked out exactly as the doctor said it would, you can safely reboot the system and select the new kernel from the boot loader menu during system bootup:

   ```
   [master@server ~]$  sudo reboot
   ```

2. After the system boots up, you can use the **uname** command to find out the name of the current kernel:

   ```
   [master@server ~]$  uname -r
   5.4.10-custom
   ```

3. You will recall that one of the features we added to our new kernel is the ability to support the NTFS file system. Make sure the new kernel does indeed have support for NTFS by displaying information about the NTFS module:

```
[master@server ~]$  modinfo ntfs
filename: /lib/modules/5.4.10-custom/kernel/fs/ntfs/ntfs.ko
license:  GPL
description:  NTFS 1.2/3.x driver
...<OUTPUT TRUNCATED>...
```


The Author Lied! It Didn't Work!

The kernel *didn't* fly, you say? It froze in the middle of booting? Or it booted all the way and then nothing worked, right? First and foremost, *don't panic*. This kind of problem happens to everyone, even the pros. So don't worry—the situation is most definitely reparable.

First, notice that a new block was added to the boot loader configuration file (**grub.cfg**) and no existing blocks were *removed*. Likewise, if the system is using BLS, you'll notice that a new boot loader configuration was created under the **/boot/loader/entries/** directory and no existing files were removed. These are all extra safety measures that allow you to safely fall back to the old kernel that you know works and boot into it. Reboot, and at the GRUB menu, select the name of the previous kernel that was known to work. This action should bring you back to a known system state.

Now go back to the kernel configuration and verify that all the options you selected will work for your system. For example, did you accidentally enable support for the wrong file system instead of the file system in use (for example, ext4, XFS, and so on)? Did you set any options that depend on other options being set? Remember to view the informative Help screen for each kernel option in the configuration interface, making sure that you understand what each option does and what you need to do to make it work right.

When you're sure you have your settings right, step through the compilation process again and reinstall the kernel. Creating an appropriate initial RAM file system image (**initramfs** file) is also important (see **man mkinitramfs**).

TIP You can use the **lsinitrd** command to view the contents of an **initramfs** image file. For example, to view the contents of an **initramfs** file located at **/boot/initramfs-5.4.10-custom.img**, you would run the following command:

```
# lsinitrd  /boot/initramfs-5.4.10-custom.img
```

Overall, remember that each time you compile a kernel, you'll get better at it. When you do make a mistake, it'll be easier to go back, find it, and fix it.

TIP Take a moment to check out what's inside the vanilla kernel source tree. A good portion of the kernel documentation is conveniently stored in the **Documentation** directory at the root of the kernel source tree. Here you'll find everything kernel related. For instance, you'll find documentation and tips about kernel patching in this file: ***<kernel-source>*/Documentation/process/applying-patches.rst**.

Patching the Kernel

Like any other operating system, Linux periodically requires upgrades to fix bugs, improve performance, improve security, and add new features. These upgrades come out in two forms: in the form of a complete new kernel release and in the form of a patch. The complete new kernel works well for people who don't have at least one complete kernel already downloaded. For those who do have a complete kernel already downloaded, patches are a much better solution because they contain only the changed code and, as such, are quicker to download.

Think of a patch as comparable to a Windows hotfix or service pack. By itself, it's useless, but when added to an existing version of Windows, you (hopefully) get an improved product. The key difference between hotfixes and patches is that patches contain the changes in the source code that need to be made. This allows you to review the source code changes before applying them. This is much nicer than hoping a fix won't break the system!

You can find out about new patches to the kernel at many Internet sites. Your distribution vendor's web site is a good place to start; it'll list not only kernel updates, but also patches for other packages. A primary source is the official Linux Kernel archive at www.kernel.org.

In this section, you'll learn how to apply a patch to update the Linux kernel source (major) version 5.4 to (stable) version 5.4.1. The exact patch file that we will use is **patch-5.4.1.xz**.

TIP Your distribution may or may not already have the `patch` utility installed. You can use DNF to install it on Red Hat–based distros like Fedora, CentOS, RHEL, openSUSE, and so on, by running the following command:

```
$ sudo dnf -y install patch
```

To install the `patch` utility on Debian-based distros like Ubuntu, Mint, Debian, and so on, you would instead run this command:

```
$ sudo apt-get -y install patch
```

Downloading and Applying Patches

Certain patch files are located in the same directory from which the kernel can be downloaded on the www.kernel.org servers. This applies to each major release of Linux; so, for example, the patch to update Linux version 6.6 to Linux version 6.6.3 might be located at https://www.kernel.org/pub/linux/kernel/v6.x/patch-6.6.3.xz. Other patches (see the discussion of rc patches later on) can be downloaded from the kernel's source control management system Git.

Each patch filename is prefixed with the string "patch" and suffixed with the Linux version number being installed by the patch.

Note that when dealing with patches related to *major* kernel versions, the patches need to be applied in an incremental manner. Each major patch brings Linux up by only one version. This means, for example, that to go from **linux-6.21** to **linux-6.23**, you'll need two patches, **patch-6.22** and **patch-6.23**, and these patches must be incrementally applied in order.

Note also that when dealing with patches within the *X.Y.Z* kernels (the stable release kernels), the patches are *not* incremental. Thus, the **patch-*X.Y.Z*** file can only be applied to the base

linux-*X.Y*. For example, if you have a base **linux-6.20.3** and want to bring it up to Linux version 6.20.5, you'll first need to revert your stable **linux-6.20.3** kernel source back to its base **linux-6.20** and then apply the new patch for **linux-6.20.5** (**patch-6.20.5**).

Patch files are distributed in a compressed format. In this example, we'll be using **patch-5.4.1.xz** (obtained from https://www.kernel.org/pub/linux/kernel/v5.x/patch-5.4.1.xz). You will also need the actual kernel source tarball that you want to upgrade. In this example, we'll use the kernel source that was downloaded from https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.4.tar.xz.

Once you have the files from the www.kernel.org site (or mirror), move them to the **/usr/src** directory. We'll assume that you unpacked (via **`tar xvJf linux-5.*.tar.xz`**) the kernel source that you want to upgrade into the **/usr/src/linux-5.4** directory. You will next decompress the patch using the **`xz`** utility, and then pipe the resulting output to the **`patch`** program, which will then do the actual work of patching/updating your kernel.

TIP For patching purposes, whenever possible, avoid reusing any kernel source code tree that you've previously done any customizations or builds within. Always try to use a brand-new and untainted kernel source code archive for patch-related tasks.

1. Copy the compressed patch file that you downloaded into a directory one level *above* the root of your target kernel source tree. Assuming, for example, that the kernel you want to patch has been decompressed and untarred into the **/usr/src/linux-5.4/** directory, you would copy the patch file into the **/usr/src/** directory.
2. Change your current working directory to the top level of the kernel source tree. This directory in our example is **/usr/src/linux-5.4/**. Type the following:

   ```
   [master@server ~]$  cd /usr/src/linux-5.4/
   ```

3. It is a good idea to do a test run of the patching process to make sure there are no errors and that the new patch will indeed apply cleanly:

   ```
   [master@server linux-5.*]$ sudo xz -dc ../patch-5.4.1.xz | \
    patch -p1 --dry-run
   ```

4. Assuming the preceding command ran successfully without any errors (check that the exit status is zero with **`echo $?`**), you're now ready to apply the patch. Run this command to decompress the patch and apply it to your kernel:

   ```
   [master@server linux-5.*]$ sudo xz -dc ../patch-5.4.1.xz | \
    patch -p1
   ```

   Here, **`../patch-5.4.1.xz`** is the name and path to the patch file. A stream of filenames is printed out to your screen. Each of those files has been updated by the patch file. If any problems occurred with the upgrade, you will see them reported here.

Kernel Release Candidates (rc)

You might sometimes see kernel patch files with names like **patch-5.4.10-rc1.xz** available under the kernel source tree online (for example, under https://www.kernel.org/pub/linux/kernel/v5.x/stable-review/). The "rc1" in this example, which makes up part of the patch name and version (and hence, the final kernel version), means that the patch in question is the "release candidate 1" patch that can be used to upgrade the appropriate kernel source tree to Linux kernel version 5.4.10-rc1. The same goes for a patch file named **patch-6.4-rc7.xz** (for the future 6.*x* series kernels)—which will be a "release candidate 7" for the kernel that will eventually become the next *stable* **linux-6.4**, and so on.

The **-rc*X*** patches are *not* incremental. They can be applied to "base" kernel versions. For example, an **-rc8** patch named **patch-5.5-rc8.xz** can be applied on top of the base 5.4.1, 5.4.2, 5.4.10, 5.4.99, and so on kernel sources (the base being the 5.4 kernel source in this case). This will require that any patches that might have been applied on top of the kernel be removed first to revert it back to its base version.

So assuming we are currently working with a kernel version 5.4.10 and we want to build and test an **rc** kernel from the v5.5 series, we need to first download **patch-5.4.10.xz** (from https://www.kernel.org/pub/linux/kernel/v5.x/patch-5.4.10.xz), decompress the file (using **`xz -d patch-5.4.10.xz`**), and use the **`patch`** command to downgrade/revert to a base 5.4 kernel. After which we can then attempt to upgrade to the **v5.5-rc8** kernel with **patch-5.5-rc8**. The abbreviated steps showing the process as well as the before and after effects are as follows:

Step 1: Remove the old patch to revert back to linux-5.4 After downloading all the necessary patch files and kernel source code archives (the relevant files for this step in our example are **linux-5.4.10.tar.xz and patch-5.4.10.xz**) into a directory where you have permissions, continue by running the following:

```
# xz  -d  patch-5.4.10.xz  &&  tar xvJf linux-5.4.10.tar.xz
# cd  linux-5.4.10/
# make  kernelversion
5.4.10
# patch  -p1  -R  <  ../patch-5.4.10
# make  kernelversion
5.4.0
```

Step 2: Download and apply the new patch to upgrade to linux-5.5-rc8 We'll download the necessary patch files from the *torvalds* source tree to perform the actual upgrade to linux-5.5-rc8 in the following steps:

```
# cd ../
# wget -O patch-5.5-rc8   https://git.kernel.org/torvalds/p/v5.5-rc8/v5.4
# cd  linux-5.4.10/
# patch -p1  <   ../patch-5.5-rc8
# make  kernelversion
5.5.0-rc8
# cd ../  && mv  linux-5.4.10/   linux-5.5.0-rc8/
```

(*continued*)

The kernel download, compile, and patch method *du jour* is to use the official Linux Source Code Management (SCM) system called Git. You would begin the process by cloning the mainline Linux Git repository using a command similar to the one here:

```
# git clone https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

Using the Git SCM route allows you to have access to bleeding-edge patches/updates that are made available on a daily basis.

If the Patch Worked

If the patch worked and you received no errors, you're just about done! You can rename the directory holding the patched kernel source tree to reflect the new version. Here's an example:

```
[master@server src]$ sudo mv  linux-5.4  linux-5.4.1
```

All that finally needs to be done is to recompile the kernel. Just follow the steps in the section "Compiling the Kernel," earlier in this chapter.

If the Patch Didn't Work

If you received errors during the process of patching the kernel, don't despair. This probably means one of two things:

- The patch version number cannot be applied to the kernel version number (for instance, you tried to apply **patch-5.20.50.xz** directly to **linux-5.20.60**).
- The kernel source itself has changed. This happens to developers who forget that they made changes!

The easiest way to fix either situation is to erase the kernel located in the directory where you unpacked it and then unpack an untainted full kernel archive there again. This will ensure that you have a pristine kernel. Then apply the patch. It's tedious, but if you've done it once, it's easier and faster the second time.

TIP You can usually back out of (remove) any patch that you apply by using the `-R` option with the `patch` command. For example, to back out of a patch version 5.5-rc8 that was applied to a base Linux kernel version 5.4.0, while in the root of the kernel source tree, you would type this:

```
# xz -dc ../patch-5.5-rc8.xz | patch -p1 -R
```

Remember that backing out of a patch can be risky at times, and it doesn't always work. In other words, your mileage may vary!

Summary

In this chapter, we discussed the process of configuring and compiling the Linux kernel. This isn't exactly a trivial process, but doing it gives you the power of fine-grained control over your system that simply isn't possible with most other operating systems. Compiling the kernel is a straightforward process. The Linux development community has provided excellent tools that make the process as painless as possible. In addition to compiling kernels, we walked through the process of upgrading kernels using the patches available from the Linux Kernel web site, www.kernel.org.

When you compile a kernel for the first time, do it on a non-production machine, if possible. This gives you a chance to take your time and fiddle with the many operational parameters available. It also means you won't annoy your users if something goes wrong!

For programmers curious about the kernel's innards, many references are available in the form of books and web sites, and, of course, the source code itself is the ultimate documentation!

CHAPTER

11 Knobs and Dials: API (Virtual) File Systems

Most operating systems (OSs) offer a mechanism by which the insides of the OS can be probed and operational parameters can be set when needed. In Linux, this mechanism is provided by the so-called application programming interface (API) file systems (or virtual or pseudo-file systems). Unlike the other file systems (ext4, Btrfs, XFS, and so on) that we discussed in Chapter 8, virtual file systems are not typically backed by actual physical storage (such as hard disks). The API file systems are typically not listed explicitly in the system's **/etc/fstab** configuration file, though you will still find them listed in the output of the `mount` or `findmnt` command.

The proc file system is a popular virtual file system on Linux-based OSs. The **/proc** directory is the mount point for the proc file system, and thus the two terms (proc vs. **/proc**) are often used interchangeably.

Other popular operating systems also make use of virtual file systems in different forms and to varying degrees. For example, Microsoft Windows systems make use of the Registry, which allows manipulation of system runtime parameters to some degree.

In this chapter, we discuss the proc file system and how it works under Linux. We'll step through some overviews and study some interesting entries in **/proc**, and then we'll demonstrate some common administrative tasks using **/proc**. We'll end with a brief mention of the SysFS, tmpfs, and cgroup virtual file systems.

What's Inside the /proc Directory?

Because the Linux kernel is such a key component in system operations, it's important that there be a method for exchanging information with the kernel. Traditionally, this is done through *system calls*—special functions written for programmers to use in requesting the kernel to perform functions on their behalf. In the context of system administration, however, system calls are handled magically for us in the background. When all you need is a simple tweak or to extract some statistics from the kernel, having to write a custom tool from scratch for this purpose might be overkill!

To improve and ease communication between users and the kernel, the proc file system was created. This file system is especially interesting because it doesn't really exist on disk anywhere; it's purely an abstraction of kernel information. All of the files in the directory correspond either to a function in the kernel or to a set of variables in the kernel.

NOTE The fact that proc is abstract doesn't mean it isn't a file system. It does mean that a special file system had to be developed to treat proc differently from normal disk-based file systems.

For example, to see a report on the type of processor on a system, we can consult one of the files under the **/proc** directory. The particular file that holds this information is the **/proc/cpuinfo** file and can be viewed with this command:

```
[master@server ~]$  cat /proc/cpuinfo
```

The kernel will dynamically create the report, showing processor information, and hand it back to **cat** so that we can see it. This is a simple-yet-powerful way for us to examine and query the kernel. The **/proc** directory supports an easy-to-read hierarchy using subdirectories, so finding information is easy. The directories under **/proc** are also organized such that files containing information about similar topics are grouped together. For example, the **/proc/net** directory offers reports about various (real-time) networking statistics and other parameters related to the networking subsystem.

Even more of an advantage is that the flow of information goes both ways: The kernel can generate reports for us, and we can easily pass information back into the kernel. For instance, performing an **ls -l** in the **/proc/sys/net/ipv4** directory will show us a lot of files that are not read-only, but read/write, which means some of the values stored in those files can be altered on the fly.

"Hey! Most of the **/proc** files have 0 bytes, and one is *huge*! What gives?" Don't worry if you've noticed all those 0-byte files—most of the files in **/proc** are 0 bytes because, as we've hinted, **/proc** doesn't really exist on disk. When you use **cat** to read a **/proc** file, the content of the file is dynamically generated by a special program inside the kernel. The report is ephemeral and therefore does not take up space. Think of it in the same fashion as you would the dynamic content that you see on many web sites. These web pages are often generated by web scripting languages (Python, PHP, and so on), and whatever you see isn't always written back to the web server's disk but is instead regenerated every time you visit the web page.

CAUTION That one really huge file you see in **/proc** is **/proc/kcore**, which is really a pointer to the contents of RAM. But don't worry about the size, because it isn't occupying any space on your disk-based file systems. Reading **/proc/kcore** is like reading the raw contents of memory (and, of course, requires root or other elevated permissions).

Tweaking Files Inside of /proc

As mentioned in the preceding section, some of the files under the **/proc** directory (and subdirectories) have a read/write mode. Let us examine one of these directories a little more closely. The files in **/proc/sys/net/ipv4** represent parameters in the TCP/IP stack that can be "tuned" dynamically. Use the **cat** command to look at a particular file, and you'll see that

most of the files contain nothing but numbers—and by changing these numbers, you can affect the behavior of the Linux TCP/IP stack!

For example, the file **/proc/sys/net/ipv4/ip_forward** contains a 0 (Off) by default. This tells Linux not to perform IP forwarding when there are multiple network interfaces. But if you want to set up something like a Linux router, you need to allow forwarding to occur. For this you can edit the **/proc/sys/net/ipv4/ip_forward** file and change the number to 1 (On).

A quick way to make this change is by using the **`echo`** command, like so:

```
[root@server ~]#  echo "1" > /proc/sys/net/ipv4/ip_forward
```

CAUTION Be very careful when tweaking parameters in the Linux kernel. There is no safety net to keep you from making the wrong settings for critical parameters, which means it's entirely possible that you can crash your system. If you aren't sure about a particular item, it's safer to leave it *be* until you're sure.

Some Useful /proc Entries

Table 11-1 lists some **/proc** entries that you may find useful in managing your Linux system. Note that this is a far cry from an exhaustive list. For more detail, peruse the directories yourself and see what you find. You can also read the **proc.txt** file in the **Documentation** directory of the Linux kernel source code.

Unless otherwise stated, you can simply use the **`cat`** program to view the contents of a particular file in the **/proc** directory.

Enumerated /proc Entries

A listing of the **/proc** directory will reveal a large number of directories whose names are just numbers. These numbers are the process identifiers (PIDs) for each running process in the system. Within each of the process directories are several files describing the state of the process. This information can be useful in finding out how the system perceives a process and what sort of resources the process is consuming. From a programmer's point of view, the process files are also an easy way for a program to get information about itself.

For example, here's a long listing of some of the files under **/proc**:

```
[master@server ~]$  ls -l /proc
dr-xr-xr-x  6 root      root              0 2072-12-27 08:54 1
dr-xr-xr-x  6 root      root              0 2072-12-27 08:54 1021
dr-xr-xr-x  6 root      root              0 2072-12-27 08:54 1048
....<OUTPUT TRUNCATED>....
```

If you look a little closer at the folder named **1** in this output, you will notice that this particular folder represents the information about the **init** process or the process with the process identifier number of 1 (PID = 1). Here's a listing of the files under **/proc/1/**:

```
[master@server ~]$  ls -l /proc/1
dr-xr-xr-x. 2 root root 0 Dec 28 02:33 attr
-r--------. 1 root root 0 Dec 28 02:33 auxv
-r--r--r--. 1 root root 0 Dec 26 11:06 cmdline
....<OUTPUT TRUNCATED>....
lrwxrwxrwx. 1 root root 0 Dec 26 11:06
 /proc/1/exe -> /usr/lib/systemd/systemd
```

Filename	Contents
/proc/cpuinfo	Information about the CPU(s) in the system.
/proc/interrupts	Interrupt Request (IRQ) usage in your system.
/proc/ioports	Displays a listing of the registered port regions used for I/O communication with devices.
/proc/iomem	Displays the current map of the system's memory for each physical device.
/proc/mdstat	Status of Redundant Array of Independent Disks (RAID) configuration.
/proc/meminfo	Status of memory usage.
/proc/kcore	Represents the physical memory of the system. Unlike the other files under **/proc**, this file has an unusually large size.
/proc/modules	Shows the currently loaded kernel modules. Same information produced as output from **`lsmod`**.
/proc/buddyinfo	Information stored in this file can be used for diagnosing memory fragmentation issues.
/proc/cmdline	Displays parameters passed to the kernel when the kernel started up (boot-time parameters).
/proc/swaps	Status of swap partitions, volume, and/or files.
/proc/version	Current version number of the kernel, the machine on which it was compiled, and the date and time of compilation.
/proc/scsi/*	Information about all of the SCSI devices.
/proc/net/arp	Address Resolution Protocol (ARP) table (same as output from **`arp -a`**).
/proc/net/dev	Information about each network device (packet counts, error counts, and so on).
/proc/net/sockstat	Statistics on network socket utilization.
/proc/sys/fs/*	Settings for file system utilization by the kernel. Many of these are writable values; be careful about changing them unless you are sure of the repercussions of doing so.
/proc/sys/net/core/netdev_max_backlog	When the kernel receives packets from the network faster than it can process them, it places them on a special queue. By default, a maximum of 1000 packets is allowed on the queue. Under extraordinary circumstances, you may need to edit this file and change the value for the allowed maximum.
/proc/sys/net/ipv4/icmp_echo_ignore_all	Default = 0, meaning that the kernel will respond to Internet Control Message Protocol (ICMP) echo-reply messages. Set this to 1 to tell the kernel to stop replying to those messages.
/proc/sys/net/ipv4/icmp_echo_ignore_broadcasts	Default = 1, meaning that the kernel will not allow ICMP responses to be sent to broadcast or multicast addresses.
/proc/sys/net/ipv4/ip_forward	Default = 0, meaning the kernel will not forward packets between network interfaces. To allow forwarding (such as for routing), change this to 1.
/proc/sys/net/ipv4/ip_local_port_range	Range of ports Linux will use when originating a connection. Default = 32768–60999.
/proc/sys/net/ipv4/tcp_syncookies	Default = 1 (On). Change to 0 (Off) to disable protection for the system against SYN flood attacks.

Table 11-1 Useful Entries Under /proc

Again, as you can see from the output, the **/proc/1/exe** file is a soft link that points to the actual executable for the program: **/usr/lib/systemd/systemd** or **/lib/systemd/systemd** (see Chapter 9). On older SysV-based distros, this link (PID = 1) will instead point to the traditional **/sbin/init** program. The same logic applies to the other numeric-named directories that are under **/proc**—that is, they represent processes.

Common proc Settings and Reports

As mentioned, the proc file system is a virtual file system, and, as a result, changes to default settings in **/proc** do not survive reboots. If you need a change to a value under **/proc** to be automatically set/enabled between system reboots, you can either use your system startup mechanisms (or scripts) so that the change is made at boot time or use the **`sysctl`** tool.

To use the ad-hoc system startup mechanism approach on a system that uses **systemd** as its startup manager, you can create an appropriate service unit file under the **/etc/systemd/system** directory to run a one-off script or command (see Chapter 10).

The more graceful and consistent way of interactively making persistent or temporary changes to variables stored in the proc file system is to use the **`sysctl`** tool. The **`sysctl`** utility is used for displaying and modifying kernel parameters in real time. Specifically, it can be used to tune parameters that are stored under the **/proc/sys/** directory of the proc file system. A summary of its usage and options is shown here:

```
sysctl [options]    variable[=value]
```

Some of the possible options are listed next:

Options	Explanation
`variable` **`[=`*** ***`value`*** **`]`**	Used to set or display the value of a key, where ***`variable`*** is the key and ***`value`*** is the value to which the key is set. For instance, for a key called **kernel.hostname**, a possible value might be **server.example.com**.
`-n`	Disables printing of the key name when printing values.
`-e`	This option is used to ignore errors about unknown keys.
`-w`	Use this option when you want to change a **`sysctl`** setting.
`-p <` ***`filename`*** **`>`**	Loads in **`sysctl`** settings from the file specified or **/etc/sysctl.conf** if no filename is given.
`-a`	Displays all values currently available.

We will use actual examples to demonstrate how to use the **`sysctl`** tool. Most of the examples shown here are Linux distribution independent—the only differences you might encounter are that some distros might ship with some of the variables already enabled or disabled. The examples demonstrate a few of the many things you can do with proc to complement day-to-day administrative tasks. Reports and tunable options available through proc are especially useful in network-related tasks. The examples also provide some background information about the proc setting we want to tune.

SYN Flood Protection

When TCP initiates a connection, the first thing it does is send a special packet to the destination, with the flag set to indicate the start of a connection. This flag is known as the *SYN flag*. The destination host responds by sending an acknowledgment packet back to the source, called (appropriately) a *SYN/ACK*. Then the destination waits for the source to return an acknowledgment, showing that both sides have agreed on the parameters of their transaction. Once these three packets are sent (this process is called the "three-way handshake"), the source and destination hosts can transmit data back and forth.

Because it's possible for multiple hosts to contact a single host simultaneously, it's important that the destination host keep track of all the SYN packets it gets. SYN entries are stored in a table until the three-way handshake is complete. Once this is done, the connection leaves the SYN tracking table and moves to another table that tracks established connections.

A SYN flood occurs when a source host sends a large number of SYN packets to a destination with no intention of responding to the SYN/ACK. This results in overflow of the destination host's tables, thereby potentially making the operating system unstable. Obviously, this is not a good thing.

Linux can prevent SYN floods by using a *syncookie*, a special mechanism in the kernel that tracks the rate at which SYN packets arrive. If the syncookie detects the rate going above a certain threshold, it aggressively begins to get rid of entries in the SYN table that don't move to the "established" state within a reasonable interval. A second layer of protection is in the table itself: If the table receives a SYN request that would cause the table to overflow, the request is ignored. This has the effect of temporarily preventing a client from connecting to the server, but it also prevents the server from crashing altogether—and kicking everyone off!

First use the **`sysctl`** tool to display the current value for the **tcp_syncookie** setting:

```
[master@server ~]$ sudo sysctl  net.ipv4.tcp_syncookies
net.ipv4.tcp_syncookies = 0
```

The output shows that this setting is currently disabled (value = 0) on our sample system. To turn on **tcp_syncookie** support, enter this command:

```
[master@server ~]$ sudo sysctl -w net.ipv4.tcp_syncookies=1
net.ipv4.tcp_syncookies = 1
```

Because **/proc** entries do not survive system reboots, you should add the following line to the end of your **/etc/sysctl.conf** configuration file. To do this using the **`echo`** command, type the following:

```
# echo "net.ipv4.tcp_syncookies = 1"  >>  /etc/sysctl.conf
```

NOTE You should, of course, first make sure that the **/etc/sysctl.conf** file does not already contain an entry for the key that you are trying to tune. If it does, you can manually edit the file and change the value of the key to the new value.

Issues on High-Volume Servers

Like any operating system, Linux has finite resources. If the system begins to run short of resources while servicing requests (such as web page access requests), it will begin refusing new service requests. The **/proc** entry **/proc/sys/fs/file-max** specifies the maximum number of open files that Linux can support at any one time. Let's say the default value on a system is 922337203. This may eventually quickly become exhausted on a busy system with a lot of network connections. Raising it to a larger number, such as 922337203000, can be useful. Using the **sysctl** command again, type the following:

```
[master@server ~]# sudo sysctl -w fs.file-max=922337203000
fs.file-max =   922337203000
```

Don't forget to append your change to the **/etc/sysctl.conf** file (or to a properly named file under the **/etc/sysctl.d/** directory) if you want the change to be persistent.

SysFS

SysFS (short for system file system) is similar to the proc file system. The primary similarities between the two are that they are both virtual file systems (in-memory file systems) and they both provide a means for information (data structures, actually) to be exported from within the kernel to the user space. SysFS is usually mounted at the **/sys** mount point. The SysFS file system can be used to obtain information about kernel objects, such as devices, modules, the system bus, firmware, and so on. This file system provides a view of the device tree (among other things) as the kernel sees it. This view displays most of the known attributes of detected devices, such as the device name, vendor name, PCI class, IRQ and Direct Memory Access (DMA) resources, and power status. Some of the information that used to be available in the old Linux 2.4 series kernel versions under the proc file system can now be found under SysFS. It provides a lot of useful information in an organized (hierarchical) manner.

Virtually all modern Linux distros have switched to using **udev** to manage devices. **udev** is used for managing device nodes under the **/dev** directory. This function used to be performed by the **devfs**. The new **udev** system allows for the consistent naming of devices, which, in turn, is useful for the hot-plugging of devices. Thanks to SysFS, **udev** is able to do all these wonderful things simply by monitoring the **/sys** directory. Using the information gleaned from the **/sys** directory, **udev** can dynamically create and remove device nodes as they are attached to or detached from the system.

Another feature of SysFS is that it provides a uniform view of the device space, as well as a familiar naming convention, thus providing a sharp contrast to what was previously seen under the **/dev** directory. It should be noted, however, that in Linux the device representations under SysFS do not provide a means to access the device through the device driver. For device driver–based access, administrators will need to continue using the appropriate **/dev** entry.

A listing of the top level of the **SysFS** directory shows these directories:

```
[master@server ~]$  ls /sys/
block bus class dev devices firmware fs hypervisor kernel module power
```

The contents of some of the top-level directories under **/sys** are described in the following table:

SysFS Directory	Description
block	Contains a listing of the block devices (such as sda, sr0, md0, loop2, and nvme0n1) detected on the system. Attributes that describe various items about the block devices (such as size and partitions) are also listed under each block device.
bus	Contains subdirectories for the physical buses detected and registered in the kernel.
class	Describes a type or class of device—such as an audio device, graphics printer, or network device. Each device class defines a set of behaviors to which devices in that class conform.
devices	Lists all detected devices and contains a listing of every physical device that is detected by the physical bus types registered with the kernel.
firmware	Lists an interface through which firmware can be viewed and manipulated.
module	Lists all loaded modules in subdirectories.
power	Holds files that can be used to manage the power state of certain hardware.

A deeper look into the **/sys/devices** directory reveals this listing:

```
[master@server ~]$  ls  /sys/devices/
breakpoint LNXSYSTM:00 platform software tracepoint pci0000:00 pnp0
 system  virtual  cpu …..
```

If we look at a sample representation of a device connected to the PCI bus on our system, we'll see these elements:

```
[master@server ~]$ ls -1  /sys/devices/pci0000:00/0000:00:00.0/
class
config
device
drivervendor
....<OUTPUT TRUNCATED>....
```

The topmost element under the **devices** directory in the preceding output describes the PCI domain and bus number. The particular system bus here is the **pci0000:00** PCI bus, where "0000" is the domain number and the bus number is "00." The function of some of the other files are listed here:

File	Function
class	PCI class
config	PCI config space
detach_state	Connection status
device	PCI device
irq	IRQ number
local_cpus	Nearby CPU mask
resource	PCI resource host address
vendor	PCI vendor ID (a list of vendor IDs can be found at **/usr/share/hwdata/pci.ids**)

cgroupfs

Control groups (cgroups) provide a mechanism for managing system resources on a Linux-based system. Resources such as memory allocation, process scheduling, disk I/O access to blocked devices, and network bandwidth can all be controlled and allocated via cgroups.

The resources are managed by so-called "resource controllers" (also known as subsystems or modules). Following are some common cgroup subsystems that can be used to control specific sytem-related tasks and processes:

- blkio (block input and output controller)
- cpuacct (CPU accounting controller)
- cpuset (CPUs and memory nodes controller)
- freezer (suspending, resuming, and check-pointing tasks)
- memory (memory controller)
- net_cls (network traffic controller)
- devices (tracking, granting, or denying access to the creation or use of device files)

To view a list of the subsystems supported on your system, type the following:

```
[master@server ~]$  cat /proc/cgroups
#subsys_name hierarchy   num_cgroups   enabled
cpu          3           32            1
memory       5           1             1
devices      6           1             1
blkio        9           1             1
...<OUTPUT TRUNCATED>...
```

TIP The **libcgroup** and **libcgroup-tools** packages provide various tools and libraries that can be used for manipulating, controlling, monitoring, and administrating control groups. On RPM-based distros such as Fedora, CentOS, and RHEL, you can install the **libcgroup** package using the DNF or Yum package manager. Here we use DNF by running the following:

```
# dnf -y install libcgroup-tools
```

cgroups make use of the cgroup pseudo-file system (cgroupfs or cgroup2), which in turn makes use of the Linux virtual file system (VFS) abstraction. cgroupfs provides a hierarchy of sorts for managing, grouping, and partitioning tasks and processes running on a system. Subsystems are attached to directories mounted under cgroupfs, and then different constraints can be applied to these by placing processes and tasks in control groups. In other words, the system administrator can use cgroupfs to assign resource constraints to a task or group of tasks.

If you have the **libcgroup-tools** package installed on a Fedora distro or the equivalent package named **cgroup-bin** on a Debian-based distro, you can use the **`lssubsys`** utility to list the cgroup VFS hierarchies of the subsystems as well as their corresponding mount points.

To use **`lssubsys`**, type the following:

```
[master@server ~]$  lssubsys -am
```

On RPM-based Linux distros, the location of the mount points for the cgroupfs hierarchy can be manually managed by entries in the **/etc/cgconfig.conf** configuration file or can be automatically managed by **systemd**.

In practical terms, cgroups can be used to isolate and force memory-hungry applications to use only a fixed amount of memory, thereby making other user/system applications appear more responsive. Chapter 9 discussed the **systemd** service manager, which makes extensive use of cgroups to speed up the system boot process as well as to manage the starting up and stopping of system services and daemons.

TIP cgroups are a very important and crucial kernel construct that other major applications and subsystems heavily rely on to provide their functionality. Enterprise-grade applications, stacks, and subsystems such as Docker, Kubernetes, **systemd**, LXC, libvirt, and so on make use of control groups either directly or indirectly.

tmpfs

tmpfs is a file system that keeps data in the system virtual memory. It is similar to the other virtual file systems discussed earlier in this chapter in that any files created are not stored permanently on disk, but are instead temporarily stored in the Kernel's internal caches. The default size of the tmpfs file system is half of the physical RAM without swap—however, the file system can grow and shrink as needed.

The **/tmp** (pronounced slash tmp) file system is traditionally used as the storage location for temporary and/or volatile, user-space, or application data. Traditionally, **/tmp** is usually backed by an actual block- or disk-based storage mechanism and on its own file system. However, most modern Linux distros and installers default to treating **/tmp** as an API file system—*if* you don't explicitly place it on its own disk-backed storage volume or partition.

One advantage to treating **/tmp** as tmpfs is that doing so will incur less overhead on the disk device in the way of disk I/O, which in turn helps to save a bit of power. Another advantage is that it makes accessing files stored therein slightly faster and the contents of /tmp are now guaranteed not to remain there after system reboots!

TIP If you want to set up **/tmp** on a disk-backed file system after the initial OS disk partitioning and setup, you can do this later on by carving up the disk or volume, formatting it with a file system (such as ext4, Btrfs, or XFS), creating the appropriate entry in **/etc/fstab**, and then mounting the file system at the **/tmp** mount point.

Whenever **/tmp** is implemented as an API file system, you will often find that there is *no* corresponding entry in the **/etc/fstab** file for it.

TIP For processes or applications that need a disk- or device-backed storage area for storing (semi)temporary data or extremely large files that may need to be preserved across reboots, it is recommended that such processes/applications store their data under the **/var/tmp** directory or file system. This is the recommendation of the Filesystem Hierarchy Standard (FHS).

The **systemd** service manager is responsible for automatically creating and mounting the **/tmp** API file system. Specifically, the **tmp.mount** unit is responsible for this task. You can view the status of the **tmp.mount** unit by running the following:

```
[master@server ~]$   systemctl  status tmp.mount
```

tmpfs Example

You can quickly set up an ad-hoc tmpfs-style file system of a specific size whenever you need to (perhaps to provide speedier access for an application or files).

For example, to set up a 5GB tmpfs file system under the **/home/mytempfs** directory (mount point), first create the mount point (if it doesn't already exist) like so:

```
$ sudo mkdir /home/mytempfs
```

Then, set up your tmpfs instance:

```
$ sudo  mount -t tmpfs -o size=5G tmpfs /home/mytempfs
```

That's it!

Summary

In this chapter, you learned about the proc file system and how you can use it to get a peek inside the Linux kernel, as well as to influence the kernel's operation. The tools used to accomplish these tasks are relatively trivial (**`echo`** and **`cat`**), but the concept of a pseudo-file system that doesn't exist on disk can be a little difficult to grasp.

Looking at proc from a system administrator's point of view, you learned the basics about the proc file system and how to get reports from various subsystems (especially the networking subsystem). You learned how to set kernel parameters to accommodate possible future enhancements. Finally, brief mention was made of the SysFS virtual file system, the cgroup file system, as well as the tmpfs file system.

This page intentionally left blank

PART

III

Networking and Security

CHAPTER 12
TCP/IP for System Administrators

CHAPTER 13
Network Configuration

CHAPTER 14
Linux Firewall (Netfilter)

CHAPTER 15
Local Security

CHAPTER 16
Network Security

This page intentionally left blank

CHAPTER

12 TCP/IP for System Administrators

Along with understanding the ins and outs of the workings of a server, system administrators today need to have a reasonably thorough understanding of the network and the protocols used for communicating with other servers, clients, and applications. The usefulness of a networked server that is not receiving or sending information over the network or that has no awareness of the network is slightly suspect!

The most prevalent protocol over which the vast majority of network communications happen is the Transmission Control Protocol/Internet Protocol—better known as TCP/IP. This chapter provides an introduction to the guts of TCP/IP. We'll tackle the contents in two parts: First, we will walk through the details of packets, Ethernet, TCP/IP, and some related protocol details. This part may seem a little tedious at first, but perseverance will pay off in the second part. In the second part, we will walk through some common problems and how you can quickly identify them. Along the way, we will use a wonderful tool called **`tcpdump`**, which you'll find indispensable in your sysadmin toolkit. Needless to say, this chapter is not meant to be a complete replacement for the many books on TCP/IP, but rather an introduction from the standpoint of someone who needs to learn about system/network administration.

The Layers

TCP/IP is composed of layers, often heard referenced as TCP/IP *stacks*. In this section, we take a look at what the TCP/IP layers are, their relationship to one another, and, finally, why they really don't match the International Organization for Standardization (ISO) seven-layer Open Systems Interconnection (OSI) model. We'll also translate the OSI layers into concepts that are relevant to your network.

Packets

At the bottom of the layering system is the smallest unit of data that networks like dealing with: *packets*. Packets contain the data that we want to transmit between our systems as well as some control information that helps networking gear determine where the packet should go.

NOTE The terms "packet" and "frame" are often used interchangeably in network discussions. In these situations, people referring to a frame often mean a packet. The difference is subtle. A *frame* is the space in which packets go on a network. At the hardware level, frames on a network are separated by preambles and post-ambles that tell the hardware where one frame begins and ends. A *packet* is the data that is contained within the frame.

A typical TCP/IP packet flowing in an Ethernet network looks like that shown in Figure 12-1. As you can see in Figure 12-1, packets are layered by protocol, with the lowest layers coming first. Each protocol uses a *header* to describe the information needed to move data from one host to the next. Packet headers tend to be small—the headers for TCP, IP, and Ethernet in their simplest and most common combined form take only 54 bytes of space from the packet. This leaves the rest of the 1446 bytes of the packet for data.

Figure 12-2 illustrates how a packet is passed up the protocol stack. Let's look into this process a little more closely.

When a host's network card receives a packet, it first checks to see if it is supposed to accept the packet. This is done by looking at the destination addresses located in the packet's headers (more about this in the "Headers" section, later in the chapter). If the network card thinks it should accept the packet, it keeps a copy of it in its memory and generates an interrupt to the operating system.

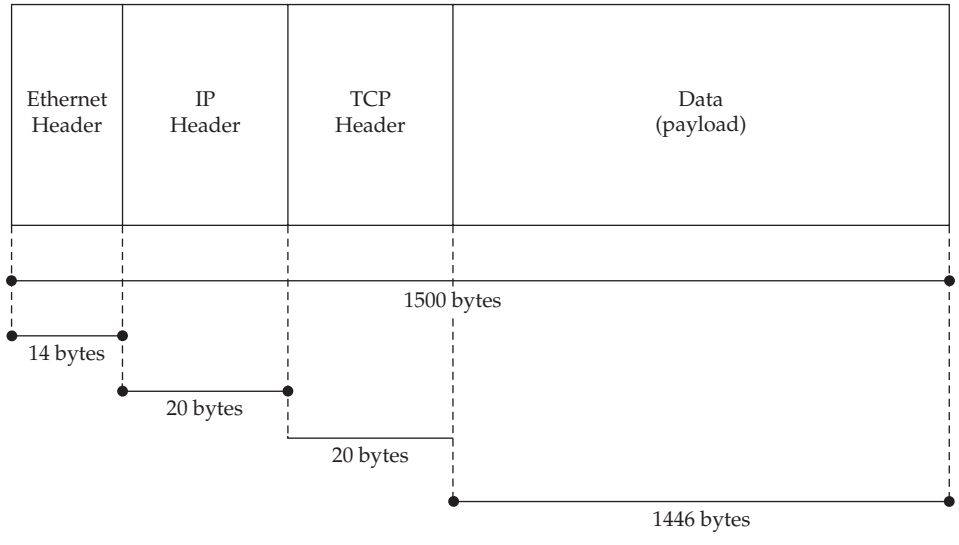


Figure 12-1 A TCP/IP packet on an Ethernet network

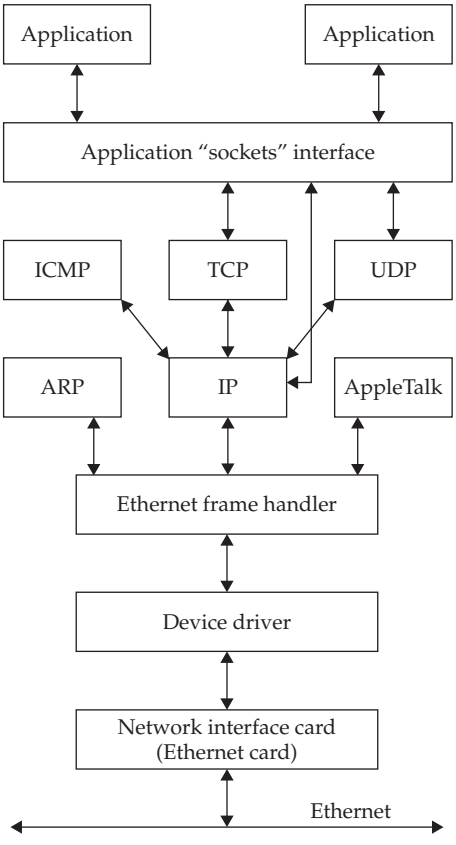


Figure 12-2 The path of a packet through the Linux networking stack

Upon receiving this interrupt, the operating system calls on the device driver of the network interface card (NIC) to process the new packet. The device driver copies the packet from the NIC's memory to the system's memory. Once it has a complete copy, it can examine the packet and determine what type of protocol is being used. Based on the protocol type, the device driver makes a note to the appropriate handler for that protocol that it has a new packet to process. The device driver then puts the packet in a place where the protocol's software ("the stack") can find it and returns to the interrupt processing.

Note that the stack does not begin processing the packet immediately. This is because the operating system may be doing something else important that it needs to finish. Because it is possible for the device driver to receive many packets from the NIC quickly, a *queue* exists between the driver and the stack software. The queue simply keeps track of the order in which packets arrive and notes where they are in memory. When the stack is ready to process those packets, it grabs them from the queue in the appropriate order.

As each layer processes the packet, appropriate headers are removed. In the case of a TCP/IP packet over Ethernet, the driver will strip the Ethernet headers, IP will strip the

IP headers, and TCP will strip the TCP headers. This will leave just the data that needs to be delivered to the appropriate application.

TCP/IP Model and the OSI Model

The TCP/IP architectural model helps describe the components of the TCP/IP protocol suite. It is also known by other names, including Internet reference model and Department of Defense (DoD) ARPANET reference model. The original TCP/IP model (RFC 1122) loosely identifies four layers: *Link layer, Internet layer, Transport layer,* and *Application layer*.

The ISO's OSI (Open Systems Interconnection) model is a well-known reference model for describing the various abstraction layers in networking. The OSI model has seven layers: *Physical layer, Data Link layer, Network layer, Transport layer, Session layer, Presentation layer,* and *Application layer*.

The TCP/IP model was created before the OSI model. Unfortunately, the newer OSI model does not have a convenient one-to-one mapping to the original TCP/IP model. Fortunately, there doesn't have to be one to make the concepts useful. Software and hardware network vendors managed to make a general mapping, and a basic understanding of what each layer of the OSI model represents in each layer of the TCP/IP model has emerged. Figure 12-3 shows the relative mapping between the OSI model and the TCP/IP model.

The following sections discuss the layers of the OSI model in more detail.

Layer 1 (The Wire)

This is the Physical layer. It describes the actual medium on which the data flows. In a network infrastructure, a pile of CAT 5 or 6 Ethernet cable and the signaling protocol are considered part of the Physical layer.

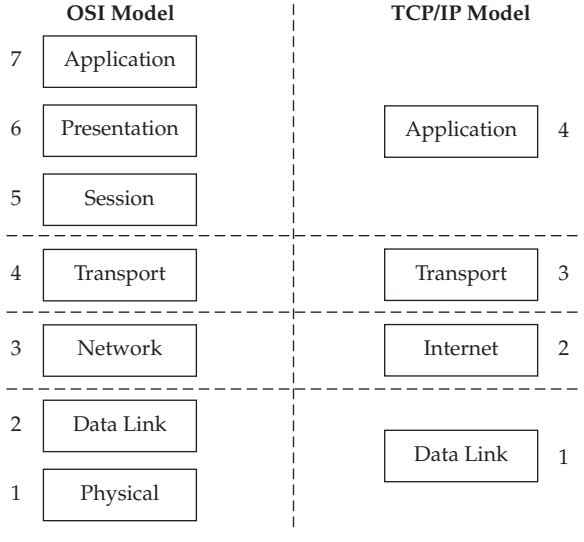


Figure 12-3 The OSI reference model and the TCP/IP model

Layer 2 (Ethernet)

This is the Data Link layer. It is used to describe the Ethernet protocol. The difference between the OSI's view of Layer 2 and Ethernet is that Ethernet concerns itself only with sending frames and providing a valid *checksum* for them. The purpose of the checksum is to allow the receiver to validate whether the data arrived as it was sent. This is done by computing the Cyclic Redundancy Check (CRC) of the packet contents and comparing them against the checksum that was provided by the sender. If the receiver gets a corrupted frame (that is, the checksums do not match), the packet is dropped here. From the Linux point of view, it should not receive a packet that the NIC knows is corrupted.

Although the OSI model formally specifies that Layer 2 should handle the automatic retransmission of a corrupted packet, Ethernet does not do this. Instead, Ethernet relies on higher level protocols (TCP in this case) to handle retransmission. Ethernet's primary responsibility is simple: get the packet from one host on a network to another host.

NOTE Ethernet is used in some metro area networks (MANs) and wide area networks (WANs) as a framing protocol for connectivity. Although the distance may be great between two endpoints, these networks are not the standard broadcast-style Ethernet you see in a typical switch. Rather, networking vendors have opted to maintain the Layer 2 framing information as Ethernet so that routers don't need to fragment packets between networks. From a system administrator's point of view, don't be concerned if your network provider says they use Ethernet in their WAN/MAN—they haven't strung together hundreds of switches to make the distance!

Layer 3 (IP)

This is the Network layer. And this is the layer at which the Internet Protocol (IP) exists. IP is wiser to the world around it than Ethernet. IP understands how to communicate with hosts inside the immediate LAN as well as with hosts that are not directly connected to you (for example, hosts on other subnets, the Internet, via routers, and so on). This means that an IP packet can make its way to any other host, so long as a path (route) exists to the destination host.

IP understands how to get a packet from one host to another. Once a packet arrives at the host, there is no information in the IP header to tell it to which application to deliver the data. The reason why IP does not provide any more features than those of a simple transport protocol is that it was meant to be a foundation upon which other protocols can rest. Of the protocols that use IP, not all of them need reliable connections or guaranteed packet order. Thus, it is the responsibility of higher level protocols to provide additional features if needed.

Layer 4 (TCP and UDP)

This is the Transport layer. TCP and User Datagram Protocol (UDP) are mapped to the Transport layer. TCP actually maps to this OSI layer quite well by providing a reliable transport for one *session*—that is, a single connection from a client program to a server program. For example, using Secure Shell (SSH) to connect to a server creates a session. You can have multiple windows running SSH from the same client to the same server, and each instance of SSH will have its own session.

In addition to sessions, TCP handles the ordering and retransmission of packets. If a series of packets arrives out of order, the stack will put the packets back into order before passing them up to the application. If a packet arrives with any kind of problem or goes missing altogether, TCP will automatically request that the sender retransmit. Finally, TCP connections are also *bidirectional*. This means that the client and server can send and receive data on the same connection. UDP, by comparison, doesn't map quite as nicely to OSI. Although UDP understands the concept of sessions and is bidirectional, it does not provide reliability. In other words, UDP won't detect lost or duplicate packets the way TCP does.

UDP: Why Even Bother?

UDP's seeming limitations are also its strengths! UDP is a good choice for two types of traffic: short request/response transactions that fit in one packet (such as Domain Name System [DNS]) and streams of data that are better off skipping lost data and moving on (such as streaming audio and video). In the first case, UDP is better, because a short request/response usually doesn't merit the overhead that TCP requires to guarantee reliability. The application is usually better off adding additional logic to retransmit on its own in the event of lost packets.

In the case of streaming data, developers actually don't want TCP's reliability. They would prefer that lost packets are simply skipped on the (reasonable) assumption that most packets will arrive in the desired order. This is because human listeners/viewers are much better at handling (and much less annoyed by!) short drops in audio than they are in delays.

Layers 5–7 (HTTP, SSL, and XML)

In the OSI model, Layers 5, 6, and 7 each have a specific purpose, but in the TCP/IP model lingo, they're all clumped together into the Application layer. Technically, all applications that use TCP or UDP sit here. For example, the popular Hypertext Transport Protocol (HTTP) traffic is considered to be a Layer 7 entity.

Secure Sockets Layer (SSL) and its successor, Transport Layer Security (TLS), are odd birds and not commonly associated with any one single layer. They sit squarely between Layer 4 (TCP) and Layer 7 (Application) and can both be used to encrypt TCP streams. Although most commonly used to encrypt web traffic (via HTTPS), TLS/SSL can encrypt arbitrary TCP connections, not just HTTP. Many protocols, such as Post Office Protocol (POP) and Internet Message Access Protocol (IMAP), offer TLS/SSL as an encryption option, and the emergence of TLS/SSL virtual private network (VPN) technology shows how TLS/SSL can be used as an arbitrary tunnel.

Extensible Markup Language (XML) data can also be confusing. To date, there is no framing protocol for XML that runs on top of TCP directly. Instead, XML data uses existing protocols, such as HTTP, Dual Independent Map Encoding (DIME), and Simple Mail Transfer Protocol (SMTP). DIME was created specifically for transmitting XML. For most applications, XML uses HTTP, which, from a layering point of view, looks like this:

Ethernet → IP → TCP → HTTP → XML

XML can wrap other XML documents within it. For example, Simple Object Access Protocol (SOAP) can wrap digital signatures within it. For additional information on XML itself, take a look at www.oasis-open.org and www.w3c.org.

ICMP

The Internet Control Message Protocol (ICMP) was designed as a simple means for hosts to communicate the state of the network among themselves. Because the data is used only by the operating system and not by end users, ICMP does not support the concept of port numbers, reliable delivery, or guaranteed order of packets.

Every ICMP packet contains a *type* that tells the recipient the nature of the message. The most popular type is *Echo-Request,* which is used by the infamous **`ping`** program. When a host receives the ICMP Echo-Request message, it responds with an ICMP *Echo-Reply* message. This allows the sender to confirm that the other host is up, and because we can see how long it takes the message to be sent and replied to, we can get an idea of the latency of the network between the two hosts.

Headers

Earlier in the chapter, you learned that a TCP/IP packet over Ethernet consists of a series of *headers* for each protocol, followed by the actual data being sent. *Packet headers,* as they are typically called, are simply those pieces of information that tell the protocol how to handle the packet. In this section we look at each of these headers (Ethernet, IP, TCP, and UDP) using the **`tcpdump`** tool. Most Linux distributions have it preinstalled, but if your distro doesn't, you can quickly install it using the package management suite in your Linux distro.

TIP You must have superuser privileges to capture packets with **`tcpdump`**.

Ethernet

Ethernet has an interesting history. As a result, there are two common types of Ethernet headers: 802.3 and Ethernet II. Thankfully, although they both look similar, you can use a simple test to tell them apart. Let's examine the contents of the Ethernet header (see Figure 12-4).

The Ethernet header contains three entries: destination address, source address, and the packet's protocol type. Ethernet addresses—also called Media Access Control (MAC) addresses—are 48-bit (6-byte) numbers that uniquely identify every Ethernet card in the world. Although it is possible to change the MAC address of an interface, this is not recommended, as the default is guaranteed to be unique, and all MAC addresses on a LAN segment should be unique.

NOTE A packet that is sent as a broadcast (meaning all network cards should accept this packet) has the destination address set to ff:ff:ff:ff:ff:ff.

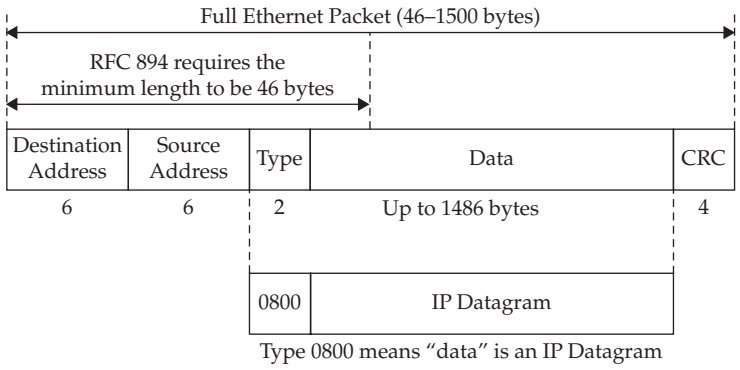


Figure 12-4 The Ethernet header

The packet's protocol type is a 2-byte value that tells us what protocol this packet should be delivered to on the receiver's side. For IP packets, this value is hex 0800 (decimal 2048).

The packet we have just described here is an Ethernet II packet. (Typically, it is just called *Ethernet*.) In 802.3 packets, the destination and source MAC addresses remain in place; however, the next 2 bytes represent the length of the packet.

The way you can tell the difference between the two types of Ethernet is that for Ethernet II, the protocol type field value is greater than or equal to 1536, while the protocol type field value in 802.3 is less than or equal to 1500. In other words, any Ethernet header where the protocol type field value is less than 1500 is really an 802.3 packet. Realistically, you probably won't see many (if any) 802.3 packets anymore.

Viewing Ethernet Headers

To see the Ethernet headers on your network, run the following command:

```
[root@server ~]#  tcpdump -e
```

This tells **`tcpdump`** to display the Ethernet headers along with the TCP and IP headers in its output.

Now generate some traffic by visiting a web site, or use SSH to communicate with another host. Doing so will generate output like this:

```
15:46:08.026966 0:d0:b7:6b:20:17 0:10:4b:cb:15:9f ethertype IPv4 (0x0800),
length 191: server.ssh > 10.2.2.2.4769: Flags [P.], seq 1724400:1724976,
ack 529, win 5740, options [nop,nop,TS val 593457613 ecr 1354193], length 576
```

The start of each line is a timestamp of when the packet was seen. The next two entries in the lines are the source and destination MAC addresses, respectively, for the packet. In the first line, the source MAC address is 0:d0:b7:6b:20:17 and the destination MAC address

is 0:10:4b:cb:15:9f. After the MAC address is the packet's type. In this case, **tcpdump** saw **0800** and automatically converted it to **ip** for us so that it would be easier to read.

If you don't want **tcpdump** to convert numbers to names for you (especially handy when your DNS resolution isn't working), you can include the **-nn** option, like this:

```
[root@server ~]#  tcpdump -e -nn
```

The **-n** option tells **tcpdump** to not do name resolution. The same preceding output without name resolution would look like this:

```
15:46:08.026966 0:d0:b7:6b:20:17 0:10:4b:cb:15:9f ethertype IPv4 (0x0800),
length 191: 10.2.2.1.22 > 10.2.2.2.4769: Flags [P.], seq 1724400:1724976,
ack 529, win 5740, options [nop,nop,TS val 593457613 ecr 1354193], length 576

15:46:08.044151 0:10:4b:cb:15:9f 0:d0:b7:6b:20:17 0800 60: 10.2.2.2.4769 >
    10.2.2.1.22: . ack 5396 win 32120 (DF)
```

Notice that in each line of the new output, the host name **server** became **10.2.2.1** and the port number **ssh** became **22**. We will discuss the meaning of the rest of the lines in the section "TCP," later in this chapter.

IP (IPv4)

The Internet Protocol has a more complex header than Ethernet. Figure 12-5 shows the fields of an IPv4 packet header. Let's go through what each of the header values signifies.

The first field in the IP header is the *version,* and for IPv4 this is equal to 4.

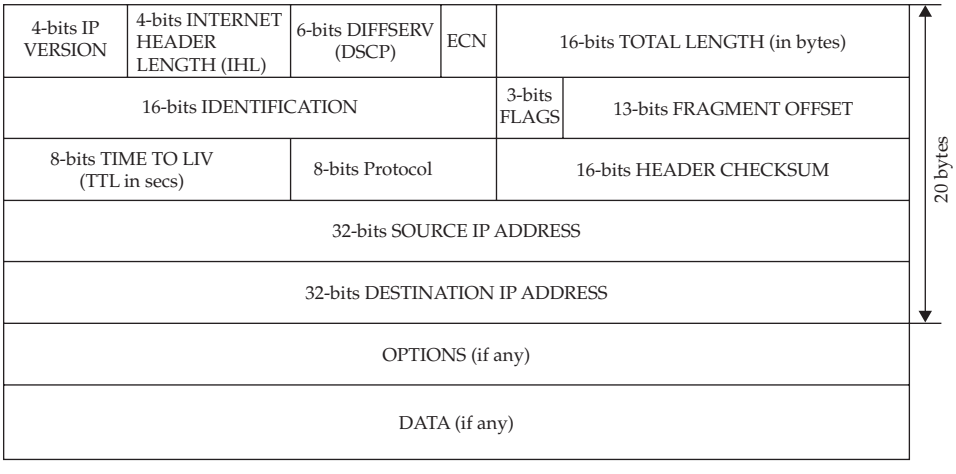


Figure 12-5 The IPv4 header

NOTE The version of IP that is in most common use today is version 4 (IPv4); however, IP version 6 (IPv6) has also become more commonplace and will increasingly become even more common.

Next is the *Internet Header Length (IHL)* field, which is the length of the IP header. The value of the length of the header is useful to know because optional parameters may be appended to the base header. The header length tells you how many, if any, options are there. To get the byte count of the total IP header length, multiply this number by 4. Typical IP headers will have a header length value of 5, indicating that there are 20 bytes in the complete header.

The *Differentiated Services Code Point (DSCP)* field tells IP stacks what kind of treatment should be given to the packet. It can, for example, be used by networking devices or applications for specifying that packets carrying interactive or real-time data be treated with special priority to minimize delays.

The *Explicit Congestion Notification (ECN)* field serves as way for supported networking devices to gracefully indicate to one another that they are experiencing network congestion, instead of simply dropping packets.

The *total length* field indicates how long the complete packet is, including the IP and TCP headers, but not including the Ethernet headers. This value is represented in bytes. An IP packet cannot be longer than 65,535 bytes.

The *identification* field is a unique number used by a host to identify the group of fragments that constitutes a datagram. The value in the *flags* field in the IP packet header indicates whether the packet is fragmented. Fragmentation occurs when an IP packet is larger than the smallest *maximum transmission unit (MTU)* between two hosts. MTU defines the largest packet that can be sent over a particular network. For example, Ethernet's MTU is 1500 bytes. Thus, if we have a 4000-byte (3980 bytes of data + 20-byte IP header) IP packet that needs to be sent over Ethernet, the packet will be fragmented into three smaller packets. The first packet can be 1500 bytes (1480 bytes of data + 20-byte IP header), the second packet can also be 1500 bytes (1480 bytes of data + 20-byte IP header), and the last packet will be 1040 bytes (1020 bytes of data + 20-byte IP header).

The *fragment offset* field tells you which part of the complete packet you are receiving. Continuing with the 4000-byte IP packet example, the first fragment will include bytes 0–1479 of data and will have an offset value of 0. The second fragment will include bytes 1480–2959 of data and will have an offset value of 185 (or 1480/8). And the third and final fragment will include fragments 2960–3999 of data and will have an offset value of 370 (or 2960/8). The receiving IP stack will take these three packets and reassemble them into one large packet before passing it up the stack.

The *time-to-live (TTL)* field is a number between 0 and 255 that signifies how much time [in seconds] a packet is allowed to have on the network before being dropped. The idea behind this is that in the event of a routing error, where the packet is going around in a circle (also known as a "routing loop"), the TTL would cause the packet to time out eventually and be dropped, thus keeping the network from becoming completely congested with circling packets. As each router processes the packet, the TTL value is decreased by one. When the TTL reaches zero, the router where this happens drops the packet and sends an *ICMP Time Exceeded* message (refer to the "ICMP" section, earlier in the chapter), informing the sender of this.

NOTE Layer 2 switches do not decrement the TTL; only routers decrement the TTL. Layer 2 switch loop detection does not rely on tagging packets but instead uses the switch's own protocol for communicating with other Layer 2 switches to form a "spanning tree." In essence, a Layer 2 switch maps all adjacent switches and sends test packets (bridge protocol data units, or BPDUs) and looks for test packets generated by itself. When a switch sees a packet return to it, a loop is found and the offending port is automatically shut down to normal traffic. Tests are constantly run so that if the topology changes or the primary path for a packet fails, ports that were shut down to normal traffic may be reopened.

The **protocol** field in the IP header indicates to which higher level protocol this packet should be delivered. Typically, this has a value for TCP, UDP, or ICMP.

The next field in the IP header is the *header checksum*. It serves as a way to perform rudimentary error-checking. When a host builds an IP packet to send, it computes the IP checksum and places the value into this field. The receiver can then do the same math and compare values. If the values do *not* match, the receiver discards the packet because it knows that the packet was corrupted during transmission. For example, a lightning strike creating an electrical disturbance might lead to packet corruption; same thing with a bad connection in the wire between the NIC and the transmission media.

Finally come the numbers that matter the most in an IP header: the *source* and *destination IP addresses.* These values are stored as 32-bit integers instead of the more human-readable dotted-decimal notation. For example, instead of 192.168.1.1, the value would be hexadecimal c0a80101 or decimal 3232235777.

tcpdump and IP

By default, **tcpdump** doesn't dump all the details of the IP header. To see everything, you need to specify the **-v** option. The **tcpdump** program will continue displaying all matching packets until you press CTRL-C to stop the output. You can ask **tcpdump** to stop automatically after a fixed number of packets by using the **-c** parameter followed by the desired number of packets. Finally, you can remove the timestamp for brevity by using the **-t** parameter.

Assuming we want to see the next two IP packets without any DNS decoding, we would run the following:

```
[root@server ~]#  tcpdump -v -t -n -c 2 ip
IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto ICMP (1),
length 84)   68.121.105.169 > 68.121.105.170:  ICMP echo request,
id 7043, seq 222, length 64
IP (tos 0x0, ttl 64, id 26925, offset 0, flags [none],
proto ICMP (1), length 84)  68.121.105.170 > 68.121.105.169:  ICMP echo reply,
id 7043, seq 222, length 64
2 packets captured
2 packets received by filter
0 packets dropped by kernel
```

The output shows a ICMP packet sent and replied to (triggered via the **ping** utility). Here's the format for a part of the output in our example:

```
src > dest: [deeper protocols] (id, seq, length)
```

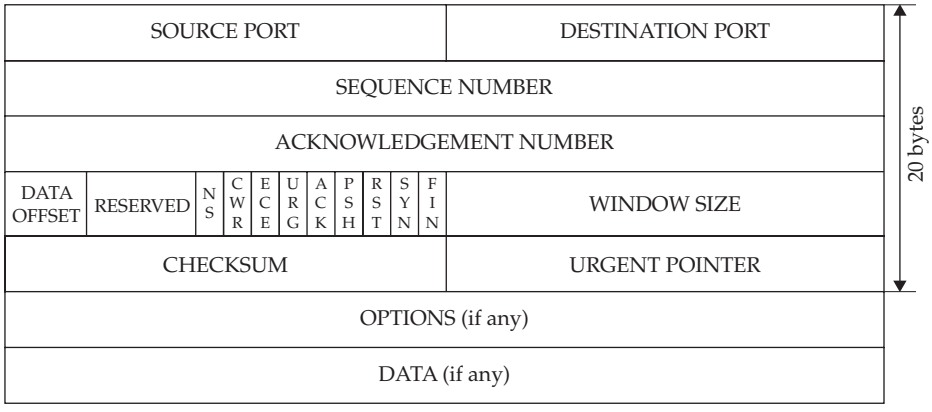


Figure 12-6 The TCP header

Here, **`src`** and **`dest`** refer to the source and destination of the packet, respectively. For TCP and UDP packets, the source and destination will include the port number after the IP address. The beginning of each individual packet captured shows the TOS, TTL, IP ID, offsets, length, and so on. Without the **`-v`** option, the TTL is shown only when it is equal to 1.

TCP

The TCP header is similar to the IP header in that it packs quite a bit of information into a little bit of space. Let's start by reviewing Figure 12-6.

The first two fields in a TCP header are the source and destination port numbers. Because each is only a 16-bit value, the possible range is 0 to 65535 (decimal). Typically, the source port is a value greater than 1024 or in the 1025–65535 range, while the destination port number is typically low or in the 1–1023 range. These low numbered ports are usually reserved for (privileged) system use on most operating systems. Not coincidentally, you'll find that most of the popular services reside within this range, although this is not a requirement.

Next we will examine the different fields of the TCP header in Figure 12-6 as well as the fields as they are seen in an actual **`tcpdump`** capture. The output of the command **`tcpdump -n -t -v`** is shown here:

```
192.168.1.1.2046 > 192.168.1.12.79: Flags [P.], ck sum 0xf4b1 (correct),
seq 1:6, ack 1, win 5740, options [nop,nop,TS val 593457613 ecr 1354193],
length 5
```

192.168.1.1.2046 > 192.168.1.12.79 We've already explained the starting fields of the output. It is simply the source and destination IP address and port number combination. The port numbers are appended immediately after the IP address. In this example, the source port number is 2046 and the destination port is 79.

Flags [P.] This next part is a bit tricky. TCP uses a series of flags to indicate whether the packet is supposed to initiate a connection, contain data, or terminate a connection. The flags are Explicit Congestion Notification–Nonce (NS), Congestion Window Reduced (CWR), Explicit Congestion-Notification Echo (ECE), Urgent (URG), Acknowledge (ACK), Push (PSH), Reset (RST), Synchronize (SYN), and Finish (FIN). Short descriptions of these flags are provided in the following table:

Flag	Meaning
NS	The ECN-Nonce flag is used for protection against accidental or malicious concealment of marked packets.
CWR	Used for indicating that a TCP segment with the ECE flag has been received.
ECE	The ECE flag serves various purposes, depending on the value of the SYN flag.
URG	Implies that urgent data in the packet should receive priority processing.
ACK	Acknowledges successfully received data.
PSH	Requests that any received data be processed immediately.
RST	Immediately terminates or resets the connection.
SYN	Requests that a new connection start. The very first packet sent from each node usually have this flag set.
FIN	Signifies the last packet from the sender.

These flags are typically used in combination with one another. For example, it is common to see PSH and ACK together. Using this combination, the sender essentially tells the receiver two things:

- Data in this packet needs to be processed.
- I am acknowledging that I have received data from you successfully.

You can see which flags are in a packet in **`tcpdump`**'s output immediately after the destination IP address and port number. Here's an example:

```
192.168.1.1.2046 > 192.168.1.12.79: Flags [P.]...<OUTPUT TRUNCATED>...
```

In this line, the flag is **`P`** for PSH. **`tcpdump`** uses the first character of the flag's name to indicate the flag's presence (such as **`S`** for SYN or **`F`** for FIN). The only exception to this is ACK, which is actually spelled out as **`ack`** later in the line. If the packet has only the ACK bit set, a period is used as a placeholder where the flags are usually printed. ACK is an exception, because it makes it easier to find what the acknowledgment number is for that packet. See the discussion on acknowledgment numbers earlier in this section; we will discuss flags in greater detail when we discuss connection establishment and teardown.

cksum 0xf4b1 The next element in the TCP header is the checksum. This is similar to the IP checksum in that its purpose is to provide the receiver a way of verifying that the data received isn't corrupted. Unlike the IP checksum, the TCP checksum actually takes into account both the TCP header and the data being sent.

seq 1:6 In **tcpdump**'s output, we see sequence numbers in packets containing data. Here's the format:

```
starting number:ending number
```

The sequence numbers in our sample **tcpdump** output are 1:6, meaning that the data started at sequence number 1 and ended at sequence number 6. These values are used by TCP to ensure that the order of packets is correct. In day-to-day administrative tasks, you shouldn't have to deal with them.

NOTE To make the output more readable, **tcpdump** uses relative values. Thus, a sequence number of 1 really means that the data contained within the packet is the first byte being sent. If you want to see the actual sequence number, use the **-S** option.

ack 1 In this sample output, we also see the acknowledgment number. When the packet has the acknowledgment flag set, it can be used by the receiver to confirm how much data has been received from the sender (refer to in the discussion of the ACK flag later in this section) and also to let the sender know which packets have been properly received.

tcpdump prints **ack**, followed by the acknowledgment number, when it sees a packet with the acknowledgment bit set.

In this case, the acknowledgment number is 1, meaning that 192.168.1.1 is acknowledging the first byte sent to it by 192.168.1.12 in the current connection.

win 5740 The next entry in the header is the window size. TCP uses a technique called *sliding window,* which allows each side of a connection to tell the other how much buffer space it has available for dealing with connections. When a new packet arrives on a connection, the available window size decreases by the size of the packet until the operating system has a chance to move the data from TCP's input buffer to the receiving application's buffer space. Window sizes are computed on a connection-by-connection basis.

Let's look at some truncated output from **tcpdump -n -t** as an example:

```
192.168.1.1.2046 > 192.168.1.12.79: . seq 6:8, ack 1 win 32120, length 2
192.168.1.12.79 > 192.168.1.1.2046: . seq 1:494, ack 8 win 17520, length 493
192.168.1.1.2046 > 192.168.1.12.79: . seq 8:8, ack 495 win 31626, length 0
192.168.1.1.2046 > 192.168.1.12.79: . seq 8:8, ack 495 win 32120, length 0
```

In the first line, 192.168.1.1 tells 192.168.1.12 that it currently has 32,120 bytes available in its buffer for this particular connection.

In the second packet, 192.168.1.12 sends 493 bytes to 192.168.1.1. And at the same time, 192.168.1.12 tells 192.168.1.1 that its available window is 17,520 bytes.

192.168.1.1 responds to 192.168.1.12 with an acknowledgment saying it has properly accepted everything up to the 495th byte in the stream, which in this case includes all of the data that has been sent by 192.168.1.12. It's also acknowledging that its available window is now 31,626, which is exactly the original window size (32,120) minus the amount of data that has been received (493 bytes). A few moments later, in the fourth line, 192.168.1.1 sends a note to 192.168.1.12 implicitly hinting that it has successfully transferred the data to the application's buffer and that its window is back to 32,120.

A little confusing? Don't worry too much about it. As a system administrator, you shouldn't have to deal with this level of detail, but it is helpful to know what the numbers mean.

NOTE You may have noticed an off-by-one error in the math here. 32,120 subtracted from 493 is 31,627, not 31,626. This has to do with the nuances of sequence numbers, calculations of available space, and other factors. For the full ugliness of how the math works, read RFC 793 (ftp://ftp.isi.edu/in-notes/rfc793.txt).

length 5 At the end of the output, you can see the length of the data being sent (5 in this example). Similar to IP's header length, TCP's header length shows how long the header is, including any TCP options. Whatever value appears in the header length field is multiplied by 4 to get the byte value.

Finally, the last notable piece of the TCP header is the *urgent pointer* (see Figure 12-6). The urgent pointer points to the offset of the octet following important data. This value is observed when the URG flag is set and tells the receiving TCP stack that some important data is present. The TCP stack is supposed to relay this information to the application so that it knows it should treat that data with extra special importance.

In reality, you'll be hard pressed to see a packet that uses the URG bit. Most applications have no way of knowing whether data sent to them is urgent or not, and most applications don't really care. As a result, a small chord of paranoia should strike you if you do see urgent flags in your network. Make sure it isn't part of a probe from the outside trying to exploit bugs in your TCP stack and cause your servers to crash. Don't worry about Linux—it knows how to handle the urgent bit correctly.

UDP

In comparison to TCP headers, UDP headers are much simpler. Let's start by looking at Figure 12-7.

The first fields in the UDP header are the source and destination port numbers. These are conceptually the same thing as the TCP port numbers. In **`tcpdump`** output, they appear in a similar manner. Let's look at a DNS query to resolve www.example.com into an IP address as an example with the command **`tcpdump -nn -t port 53`**:

```
192.168.1.1.1096 > 192.168.1.8.53: 25851+ A? www.example.com. (31)
```

In this output, you can see that the source port of this UDP packet is 1096 and the destination port is 53. The rest of the line is the DNS request in a human-readable form. The next field in the UDP header is the length of the packet. **`tcpdump`** does not display this information.

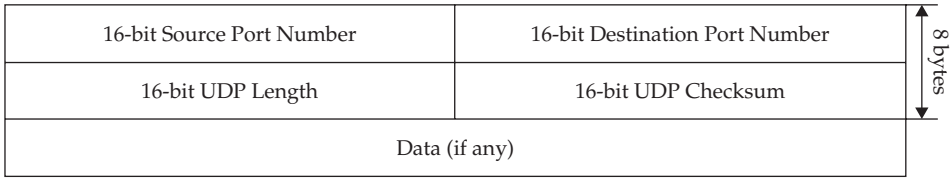


Figure 12-7 The UDP packet header

The last field is the UDP checksum. This is used by UDP to validate that the data arrived at its destination without corruption. If the checksum is corrupted, **tcpdump** will tell you.

A Complete TCP Connection

As we discussed earlier, TCP operates on the notion of a *connection.* Each connection must go through a sequence to get established; after both sides are done sending data, they must go through another sequence to close the connection. In this section, we review the complete process of a simple HTTP request and view the process as seen by **tcpdump**. Note that all of the **tcpdump** logs in this section were generated with the **tcpdump -nn -t port 80** command. Unfortunately, because of the complex nature of TCP, we cannot cover every possible scenario that a TCP connection can take. However, the coverage provided here should be enough to help you determine when things are going wrong at the network level rather than at the server level.

Opening a Connection

TCP undergoes a *three-way handshake* for every connection that it opens. This allows both sides to send each other their state information and give each other a chance to acknowledge the receipt of that data.

The first packet is sent by the (source) host that wants to open the connection with a server. For this discussion, we will call this host the *client.* The client sends a TCP packet over IP and sets the TCP flag to SYN. The sequence number is the initial sequence number that the client will use for all of the data it will send to the other (destination) host, which we'll call the *server*.

The second packet is sent from the server to the client. This packet has two TCP flags set: SYN and ACK. The purpose of the ACK flag is to tell the client that it has received the first SYN packet. This is double-checked by placing the client's sequence number in the acknowledgment field. The purpose of the SYN flag is to tell the client with which sequence number the server will be sending its responses.

Finally, the third packet goes from the client to the server. It has only the ACK bit set in the TCP flags for the purpose of acknowledging to the server that it received its SYN. This ACK packet has the client's sequence number in the sequence number field and the server's sequence number in the acknowledgment field.

Sound a little confusing? Don't worry—it is! Let's try to clarify it with a real example from **tcpdump**. The first packet is sent from 192.168.1.1 to 207.126.116.254, and it looks like this really long line:

```
192.168.1.1.1367 > 207.126.116.254.80: Flags[S],seq 2524389053:2524389053
win 32120 <mss 1460,sackOK,timestamp 26292983 0,nop,wscale 0>, length 0
```

You can see the client's port number is 1367 and the server's port number is 80 (HTTP). The **S** means that the SYN bit is set and that the sequence number is 2524389053. The **length 0** at the end of the output means that there is no data in this packet. After the window is specified as being 32,120 bytes large, you can see that **tcpdump** has shown which

TCP options were part of the packet. The only option worth noting as a system administrator is the MSS (Maximum Segment Size) value. This value tells you the maximum size that TCP is tracking for a non-segmented packet for that given connection. Connections that require small MSS values because of the networks that are being traversed typically require more packets to transmit the same amount of data. More packets mean more overhead, and that means more CPU cycles required to process a given connection.

Notice that no acknowledgment bit is set and there is no acknowledgment field to print. This is because the client has no sequence number to acknowledge yet! Time for the second packet from the server to the client:

```
207.126.116.254.80 > 192.168.1.1.1367: Flags[S.], seq 1998624975:1998624975
ack 2524389054 win 32736 <mss 1460>, length 0
```

Like the first packet, the second packet has the SYN bit set, meaning that it is telling the client what it will start its sequence number with (in this case, 1998624975). It's okay that the client and server use different sequence numbers. What's important, though, is that the server acknowledges receiving the client's first packet by turning on the ACK bit and setting the acknowledgment field to 2524389054 (the sequence number that the client used to send the first packet plus one).

Now that the server has acknowledged receiving the client's SYN, the client needs to acknowledge receiving the server's SYN. This is done with a third packet that has only the ACK bit set in its TCP flags. This packet looks like this:

```
192.168.1.1.1367 > 207.126.116.254.80: Flags [.], ack 1, win 32120
```

You can clearly see that there is only one TCP bit set: ACK (indicated by the dot). The value of the acknowledgment field is shown as a 1. But wait! Shouldn't it be acknowledging 1998624975? Well, don't worry—it is. **`tcpdump`** has been kind enough to switch automatically into a mode that prints out the relative sequence and acknowledgment numbers instead of the absolute numbers. This makes the output much easier to read. So in this packet, the acknowledgment value of 1 means that it is acknowledging the server's sequence number plus one. We now have a fully established connection.

So why all the hassle to start a connection? Why can't the client just send a single packet over to the server stating, "I want to start talking, okay?" and have the server send back an "okay" response? The reason is that without all three packets going back and forth, neither side is sure that the other side received the first SYN packet—and that packet is crucial to TCP's ability to provide a reliable and ordered transport.

Transferring Data

With a fully established connection in place, both sides are able to send data. Since we are using an HTTP request as an example, we will first see the client generate a simple request for a web page. The **`tcpdump`** output looks like this:

```
192.168.1.1.1367 > 207.126.116.254.80:  Flags [P.],
seq 1:8, ack 1 win 32120 , length 7
```

Here we see the client sending 7 bytes to the server with the PSH bit set. The intent of the PSH bit is to tell the receiver to process the data immediately, but because of the nature of the Linux network interface to applications (sockets), setting the PSH bit is unnecessary. Linux (like all socket-based operating systems) automatically processes the data and makes it available for the application to read as soon as it can.

Along with the PSH bit is the ACK bit, because TCP always sets the ACK bit on outgoing packets. The acknowledgment value is set to 1, which, based on the connection setup we observed in the previous section, means that there has been no new data that needs acknowledging.

Given that this is an HTTP transfer, it is safe to assume that since it is the first packet going from the client to the server, it is probably the request itself.

Now the server sends a response to the client with this packet:

```
207.126.116.254.80 > 192.168.1.1.1367: Flags [P],
seq 1:767, ack 8 win 32736 , length 766
```

Here the server is sending 766 bytes to the client and acknowledging the first 8 bytes that the client sent to the server. This is probably the HTTP response. Since we know that the web page we requested is small, this is probably all of the data that is going to be sent in this request.

The client acknowledges this data with the following packet:

```
192.168.1.1.1367 > 207.126.116.254.80:   Flags [.], seq  8:8,
ack 767 win 31354, length 0
```

This is a *pure acknowledgment,* meaning that the client did not send any data, but it did acknowledge up to the 767th byte that the server sent.

The process of the server sending some data and then getting an acknowledgment from the client can continue as long as there is data that needs to be sent.

Closing the Connection

TCP connections have the option of ending ungracefully. That is to say, one side can tell the other "stop *now!*" Ungraceful shutdowns are accomplished with the RST (reset) flag, which the receiver does not acknowledge upon receipt. This is to keep both hosts from getting into an "RST war," where one side resets and the other side responds with a reset, thus causing a never-ending ping-pong effect.

Let's start with examining a clean shutdown of the HTTP connection we've been observing so far. In the first step of shutting down a connection, the side that is ready to close the connection sends a packet with the FIN bit set, indicating that it is finished. Once a host has sent a FIN packet for a particular connection, it is not allowed to send anything other than acknowledgments. This also means that even though it might be finished, the other side may still send it data. It is not until both sides send a FIN that both sides are finished. And like the SYN packet, the FIN packet must receive an acknowledgment.

In the next two packets, we see the server tell the client that it is finished sending data, and the client acknowledges this:

```
207.126.116.254.80 > 192.168.1.1.1367:   Flags [F],
seq 767:767, ack 8, win  32736, length 0
192.168.1.1.1367 > 207.126.116.254.80:   Flags [.], seq 8:8,
ack 768, win 31353, length 0
```

We then see the reverse happen. The client sends a FIN to the server, and the server acknowledges it:

```
192.168.1.1.1367 > 207.126.116.254.80: Flags [F],
seq 8:8, ack 768,win 32120, length 0
207.126.116.254.80 > 192.168.1.1.1367: Flags [.],
seq 768:768, ack 9 win 32735, length 0
```

And that's all there is to a graceful connection shutdown.

As was indicated earlier, an ungraceful shutdown is simply one side sending another the RST packet, which looks like this:

```
192.168.1.1.1368 > 207.126.116.254.80: Flags [R],
seq 93949335:93949349, win 0, length 14
```

In this example, 192.168.1.1 is ending a connection with 207.126.116.254 by sending a reset. After receiving this packet, a run of **`netstat`** on 207.126.116.254 (which happens to be another Linux server) affirmed that the connection was completely closed.

How ARP Works

The Address Resolution Protocol (ARP) is a mechanism that allows IP to map Ethernet addresses to IP addresses. This is important, because when you send a packet on an Ethernet network, it is necessary to put in the Ethernet address of the destination host.

The reason we separate ARP from Ethernet, IP, TCP, and UDP is that ARP packets do not go up the normal packet path. Instead, because ARP has its own Ethernet header type (0806), the Ethernet driver sends the packet to the ARP handler subsystem, which has nothing to do with TCP/IP. The basics of how ARP works is as follows:

1. The client looks in its ARP cache to see if it has a mapping between the IP address and the Ethernet address. (You can see your local ARP cache by running **`arp -a`** on your system.)
2. If an Ethernet address for the requested IP address is not found, a broadcast packet is sent out requesting a response from the (destination) node with the IP we want.
3. If the host with that IP address is on the LAN, it will respond to the ARP request, thereby informing the sender of its Ethernet address/IP address combination.
4. The client saves this information in its cache and is now ready to build a packet for transmission.

Here's an example of this from **tcpdump** with the command **tcpdump -e -t -n arp**:

```
0:a0:cc:56:fc:e4 0:0:0:0:0:0, ethertype ARP (0x0806),
length 42: Request who-has 192.168.1.1 tell 192.168.1.8
0:10:4b:cb:15:9f 0:a0:cc:56:fc:e4, ethertype ARP (0x0806),
 length 60: Reply  192.168.1.1 (0:10:4b:cb:15:9f) is-at 0:10:4b:cb:15:9f
```

The first packet is a broadcast packet (from 192.168.1.8) asking all of the hosts on the LAN for 192.168.1.1's Ethernet address. The second packet is a response from 192.168.1.1 stating its IP/MAC address mapping.

This, of course, begs the question, "If we can find the MAC address of the destination host using a broadcast, why can't we just send all packets to the broadcast?" The answer has two parts. The first is that the broadcast packet requires that hosts on the LAN receiving the packet take a moment and process it. This means that if two hosts are having an intense conversation (such as a large file transfer), all the other hosts on the same LAN would incur a lot of overhead checking on packets that don't belong to them. The second part is that networking hardware (such as switches) relies on Ethernet addresses to forward packets quickly to the right place and to minimize network congestion. By default, any time a switch sees a broadcast packet, it forwards that packet to *all* of its ports. This will make a switch no better than a hub.

"Now, if I need the MAC address of the destination host to send a packet to it, does that mean I have to send an ARP request to hosts that are sitting across the Internet?" The answer is a reassuring *no.*

When IP figures out where a packet should head off to, it first checks the routing table. If it can't find the appropriate route entry, IP looks for a *default route.* This is the path that, when all else fails, should be taken. Typically, the default route points to a router or firewall that understands how to forward packets to the rest of the world.

This means that when a host needs to send something to another server across the Internet, it only needs to know how to get the packet to a router (also know as its default gateway), and therefore it only needs to know the MAC address of the router.

To see this happen on your network, do a **tcpdump** on your host and then visit a web site that is elsewhere on the Internet, such as www.kernel.org. You will see an ARP request from your machine to your default router, a reply from your default router, and then the first packet from your host with the destination IP of the remote web server.

The ARP Header: ARP Works with Other Protocols, Too!

The ARP protocol is not specific to Ethernet and IP. To see why, let's take a quick peek at the ARP header (see Figure 12-8).

The first field that we see in the ARP header (which follows the Ethernet header) is the hardware type. The hardware type field specifies the type of hardware address. Ethernet has a value of 1. The next field is the protocol type. This specifies the protocol address being mapped. In the case of IP, this is set to 0800 (hexadecimal). The hardware size and protocol size fields that immediately follow tell ARP the size of the addresses it is mapping. Ethernet has a size of 6, and IP has a size of 4.

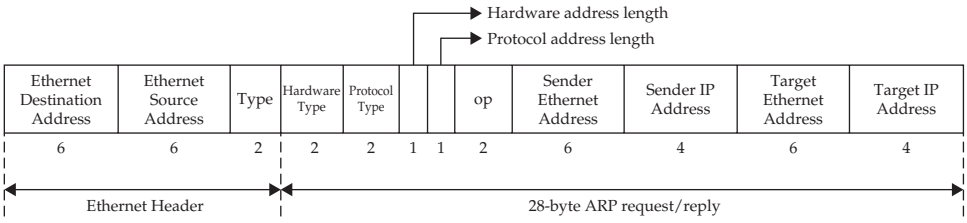


Figure 12-8 The ARP packet header

The op field tells ARP what needs to be done. ARP requests are 1, and ARP replies are 2.

Finally, there are the fields that we are trying to map. A request has the sender's Ethernet and IP addresses as well as the destination IP address filled in. The reply fills in the destination Ethernet address and responds to the sender. A variant of ARP called RARP (which stands for Reverse ARP) exists, but it has different values for the op field.

Bringing IP Networks Together

Now that we have some of the fundamentals of TCP/IP covered, let's take a look at how they work to let you glue networks together. This section will cover hosts and networks, netmasks, static routing, and some basics in dynamic routing.

The purpose of this section isn't to show you how to configure a Linux router but to introduce the concepts. Although you might find it less exciting than actually getting down and dirty, you'll find that understanding the basics makes the other stuff a little more interesting.

Hosts and Networks

The Internet consists of a bunch of interconnected networks. All of these networks have agreed to connect with some other network, thus allowing them to connect to one another. Each of these component networks is assigned a network address.

Traditionally, in a 32-bit IP address, the network component typically takes up 8, 16, or 24 bits to encode a class A, B, or C network, respectively. Since the remainder of the bits in the IP address is used to enumerate the host within the network, the fewer bits that are used to describe the network, the more bits that are available to enumerate the hosts. For example, class A networks have 24 bits left for the host component, which means there can be upward of 16,777,214 hosts within that network. (Classes B and C have 65,534 and 254 nodes, respectively.)

NOTE There are also class D and class E ranges. Class D is used for multicast, and class E is reserved for experimental use.

In order to better organize the various classes of networks, it was decided early in IP's life that the first few bits would decide to which class the address belonged. For the sake of readability, the first *octet* of the IP address specifies the class.

NOTE An octet is 8 bits, which in the typical dotted-decimal notation of IP means the number before a dot. For example, in the IP address 192.168.1.42, the first octet is 192, the second octet is 168, and so on.

The ranges are as follows:

Class	Start Address	End Address
A	0.0.0.0	127.255.255.255
B	128.0.0.0	191.255.255.255
C	192.0.0.0	223.255.255.255

Within the previous ranges, there are some special addresses reserved for special uses. The first special address is one you'll likely find familiar: 127.0.0.1. This is also known as the *loopback address.* It is set up on every host using IP so that it can refer to itself. It seems a bit odd to do it this way, but just because a system is capable of speaking IP doesn't mean it has an IP address allocated to it! On the other hand, the 127.0.0.1 address is virtually guaranteed. (If it isn't there, more likely than not, something has gone wrong.)

Three other ranges are notable, and they are considered *private IP address* blocks*.* These ranges are not allowed to be allocated to anyone on the Internet; therefore, you may use them on your internal networks. They include the following:

Start Address	End Address
10.0.0.0	10.255.255.255
172.16.0.0	172.31.255.255
192.168.0.0	192.168.255.255

NOTE We define internal networks as networks that are often behind a firewall—not directly connected to the Internet. These usually have a router performing network address translation (NAT) at the edge of the network facilitating access to the Internet.

Subnetting

Imagine a network with a few thousand hosts on it, which is normal in most medium- to large-sized companies and very routine in cloud-based environments. Trying to tie them all together into a single large network would probably lead you to pull out all your hair, bang your head on the wall, or possibly both!

The reasons for not keeping a network as a single large entity range from technical issues to political ones. On the technical front, there are limitations to every technology on how large a network can get before it becomes too large. Ethernet, for instance, cannot have more than 1024 hosts on a single collision domain. Realistically, having more than a dozen on an even mildly busy network will cause serious performance issues. Even migrating hosts to switches doesn't solve the entire problem, since switches, too, have limitations on how many hosts they can deal with. Of course, you're likely to run into management issues before you hit limitations of switches; managing a single large network is difficult. Furthermore, as an

organization grows, individual departments will begin compartmentalizing. Human resources is usually the first candidate to need a secure network of its own so that nosy engineers don't peek into things they shouldn't. To support a need like that, you need to create subnetworks, a task more commonly referred to as *subnetting*.

Assuming our corporate network is 10.0.0.0, we could subnet it by setting up smaller class C networks within it, such as 10.1.1.0, 10.1.2.0, 10.1.3.0, and so on. These smaller networks would have 24-bit network components and 8-bit host components. Since the first 8 bits would be used to identify our corporate network, we could use the remaining 16 bits of the network component to specify the subnet, giving us 65,534 possible subnetworks. Of course, you don't have to use all of them!

NOTE As you've seen earlier in this chapter, network addresses have the host component of an IP address typically set to all zeros. This convention makes it easy for other humans to recognize which addresses correspond to entire networks and which addresses correspond specifically to hosts.

Netmasks

The purpose of a *netmask* is to tell the IP stack which part of the IP address is the network and which part is the host. This allows the stack to determine whether a destination IP address is on the LAN or if it needs to be sent to a router for forwarding elsewhere.

The best way to start looking at netmasks is to look at IP addresses and netmasks in their binary representations. Let's look at the 192.168.1.42 address with the netmask 255.255.255.0:

Dotted Decimal	Binary			
192.168.1.42	11000000	10101000	00000001	00101010
255.255.255.0	11111111	11111111	11111111	00000000

In this example, we want to find out what part of the IP address 192.168.1.42 is network and what part is host. Now, according to the definition of netmask, those bits that are zero are part of the host. Given this definition, we see that the first three octets make up the network address and the last octet makes up the host.

In discussing network addresses with other people, you'll often find it handy to be able to state the network address without having to give the original IP address and netmask. Thankfully, this network address is computable, given the IP address and netmask, using a bitwise AND operation.

The way the bitwise AND operation works can be best explained by observing the behavior of two bits being ANDed together. If both bits are 1, then the result of the AND is also 1. If either bit (or both bits) is zero, the result is zero. You can see this more clearly in this table:

Bit 1	Bit 2	Result of Bitwise AND
0	0	0
0	1	0
1	0	0
1	1	1

So computing the bitwise AND operation on 192.168.1.42 and 255.255.255.0 yields the bit pattern 11000000 10101000 00000001 00000000. Notice that the first three octets remained identical and the last octet became all zeros. In dotted-decimal notation, this reads 192.168.1.0.

NOTE Remember that we usually need to give up one IP to the network address and one IP to the broadcast address. In this example, the network address is 192.168.1.0 and the broadcast address is 192.168.1.255.

Let's walk through another example. This time, we want to find the address range available to us for the network address 192.168.1.176 with a netmask of 255.255.255.240. This type of netmask is commonly given by ISPs to businesses that require more than a single IP address (aka address blocks).

A quick breakdown of the last octet in the netmask shows us that the bit pattern for 240 is 11110000. This means that the first three octets of the network address, plus four bits into the fourth octet, are held constant (255.255.255.240 in binary is 11111111 11111111 11111111 11110000). Since the last four bits are variable, we know we have 16 possibilities ($2^4 = 16$). Thus, our address range goes from 192.168.1.176, 192.168.1.177, 192.168.1.178 …. 192.168.1.191.

Because it is so tedious to type out complete netmasks, most people use the abbreviated format, where the network address is followed by a slash and the number of bits in the netmask. So the network address 192.168.1.0 with a netmask of 255.255.255.0 would be abbreviated to 192.168.1.0/24. And the network address 192.168.1.176 with a netmask of 255.255.255.240 will be abbreviated to 192.168.1.176/28.

NOTE Using netmasks that do not neatly fall on the class A, B, or C boundaries is also known as *classless interdomain routing (CIDR)*. You can read more about CIDR in RFC 4632 (www.rfc-editor.org/rfc/rfc4632.txt).

Static Routing

When two hosts on the same LAN want to communicate, it is quite easy for them to find each other: Simply send out an ARP message, get the other host's MAC address, and be done with it. But when the second host is not local, things become trickier.

To get two or more LANs to communicate with one another, a router needs to be put into place. A router often knows about the topology of multiple networks. When you want to communicate with another network, your machine will set the destination IP as the host on the other network, but the destination MAC address will be for the router. This allows the router to receive the packet and examine the destination IP, and since it knows that IP is on the other network, it will forward the packet. The reverse is also true for packets that are coming from the other network to your network (see Figure 12-9).

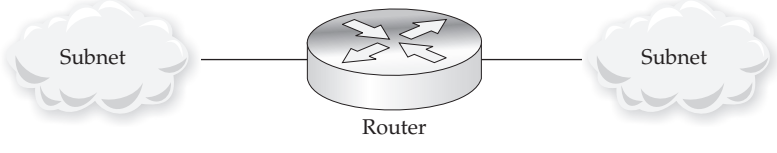


Figure 12-9 Two networks connected by a router

The router, in turn, must know what networks are plugged into it. This information is called a *routing table.* When a router (or host) is *manually* informed (by a human operator, for example) about what paths it can take, the table is called *static,* thus the term *static routing.*

Unfortunately, commercial-grade routers can be rather expensive devices. They are typically dedicated pieces of hardware that are highly optimized for forwarding packets from one interface to another. Fortunately, you can easily create a highly performant Linux-based router (see Chapter 13) using a stock PC that has two or more network interfaces. As with any advice, take it within the context of your requirements, budget, and skills. Open source and Linux are great tools, but like anything else, make sure you're using the right tool for the job.

Routing Tables

As mentioned earlier, routing tables consist of lists of host addresses, network addresses, netmasks, and destination interfaces. A simplified version of a table might look like this:

Network Address	Netmask	Destination Interface
192.168.1.0	255.255.255.0	Interface 1
192.168.2.0	255.255.255.0	Interface 2
192.168.3.0	255.255.255.0	Interface 3
Default	0.0.0.0	Interface 4

When a packet arrives at a router that has a routing table like this, it will go through the list of routes and apply each netmask to the destination IP address. If the resulting network address is equal to the network address in the table, the router knows to forward the packet on to that interface.

So let's say that the router receives a packet with the destination IP address set to 192.168.2.233. The first table entry has the netmask 255.255.255.0. When this netmask is applied to 192.168.2.233, the result is not 192.168.1.0, so the router moves on to the second entry. Like the first table entry, this route has a netmask of 255.255.255.0. The router will apply this to 192.168.2.233 and find that the resulting network address is equal to 192.168.2.0. So now the appropriate route is found. The packet is forwarded out of interface 2.

If a packet arrives that doesn't match the first three routes, it will match the default case. In our sample routing table, this will cause the packet to be forwarded to interface 4. More than likely, this is a gateway to the Internet.

Limitations of Static Routing

The example of static routing we've used is typical of smaller networks. Static routing is useful when only a handful of networks need to communicate with one another and they aren't going to change often.

There are, however, limitations to this technique. The biggest limitation is human—you are responsible for updating all of your routers with new information whenever you make any changes. Although this is usually easy to do in a small network, it does leave room for (human) error. Furthermore, as your network grows and more routes get added, it is more likely that the routing table will become trickier to manage this way.

The second—but almost as significant—limitation is that the time it takes the router to process a packet is almost proportional to the number of routes that exist. With only three or four routes, this isn't a big deal. But as you start getting into dozens of routes, the overhead becomes noticeable. Given these two limitations, it is best to use static routes only in small networks.

Dynamic Routing with RIP

As networks grow, the need to subnet them grows, too. Eventually, you'll find that you have a lot of subnets that can't all be tracked easily, especially if they are being managed by different administrators. One subnet, for instance, might need to break its network in half for security reasons. In a situation this complex, going around and telling everyone to update their routing tables would be a real nightmare and would lead to all sorts of network headaches.

The solution to this problem is to use *dynamic routing.* The idea behind dynamic routing is that each router knows only immediately adjacent networks when it starts up. It then announces to other routers connected to it what it knows, and the other routers reply back with what they know. Think of it as "word of mouth" advertising for your network. You tell the people around you about your network, they then tell their friends, and their friends tell their friends, and so on. Eventually, everyone connected to the network knows about your new network.

On campus-wide networks (such as a large company with many departments), you'll typically see this method of announcing route information. Popular examples of commonly used routing protocols are Routing Information Protocol (RIP) and Open Shortest Path First (OSPF).

RIP is a simple protocol that is easy to configure. Simply tell the router information about one network (making sure each subnet in the company has a connection to a router that knows about RIP), and then have the routers connect to one another. RIP broadcasts happen at regular time intervals (usually less than a minute), and in only a few minutes, the entire campus network knows about you. Let's see how a smaller campus network with four subnets would work with RIP. Figure 12-10 shows how the network is connected.

NOTE For the sake of simplicity, we're serializing the events. In reality, many of these events would happen in parallel.

As illustrated in this figure, Router 1 would be told about 192.168.1.0/24 and about the default route to the Internet. Router 2 would be told about 192.168.2.0/24, Router 3 would know about 192.168.3.0/24, and so on. At startup, each router's table looks like this:

Router	Route Table
Router 1	192.168.1
	Internet gateway
Router 2	192.168.2
Router 3	192.168.3
Router 4	192.168.4

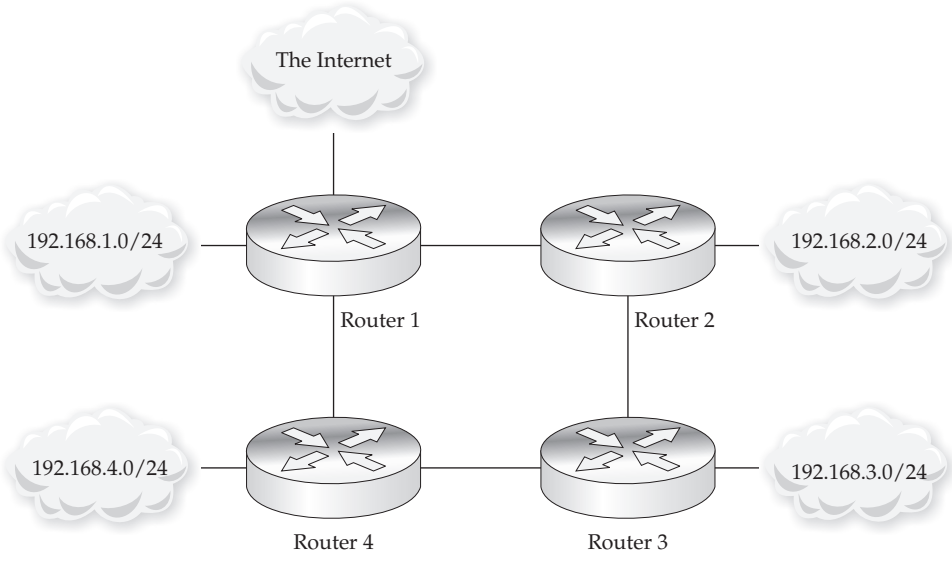


Figure 12-10 A small campus network using RIP

Router 1 then makes a broadcast stating what routes it knows about. Since Routers 2 and 4 are connected to it, they update their routes. This makes the routing table look like this (new routes in italics):

Router	Route Table
Router 1	192.168.1.0/24
	Internet gateway
Router 2	192.168.2.0/24
	192.168.1.0/24 via router 1
	Internet gateway via router 1
Router 3	192.168.3.0/24
Router 4	192.168.4.0/24
	192.168.1.0/24 via router 1
	Internet gateway via router 1

Router 2 then makes its broadcast. Routers 1 and 3 see these packets and update their tables as follows (new routes in italics):

Router	Route Table
Router 1	192.168.1.0/24
	Internet gateway
	192.168.2.0/24 via router 2
Router 2	192.168.2.0/24
	192.168.1.0/24 via router 1
	Internet gateway via router 1
Router 3	192.168.3.0/24
	192.168.2.0/24 via router 2
	192.168.1.0/24 via router 2
	Internet gateway via router 2
Router 4	192.168.4.0/24
	192.168.1.0/24 via router 1
	Internet gateway via router 1

Router 3 then makes its broadcast, which Routers 2 and 4 hear. This is where things get interesting, because this introduces enough information to open up multiple routes to the same destination. The routing tables now look like this (new routes in italics):

Router	Route Table
Router 1	192.168.1.0/24
	Internet gateway
	192.168.2.0/24 via router 2
Router 2	192.168.2.0/24
	192.168.1.0/24 via router 1
	Internet gateway via router 1
	192.168.3.0/24 via router 3
Router 3	192.168.3.0/24
	192.168.2.0/24 via router 2
	192.168.1.0/24 via router 2
	Internet gateway via router 2
Router 4	192.168.4.0/24
	192.168.1.0/24 via router 1 *or 3*
	Internet gateway via router 1 *or 3*
	192.168.3.0/24 via router 3
	192.168.2.0/24 via router 3

Next, Router 4 makes its broadcast. Routers 1 and 3 hear this and update their tables to the following (new routes in italics):

Router	Route Table
Router 1	192.168.1.0/24
	Internet gateway
	192.168.2.0/24 via router 2 *or 4*
	192.168.3.0/24 via router 4
	192.168.4.0/24 via router 4
Router 2	192.168.2.0/24
	192.168.1.0/24 via router 1
	Internet gateway via router 1
	192.168.3.0/24 via router 3
Router 3	192.168.3.0/24
	192.168.2.0/24 via router 2
	192.168.1.0/24 via router 2 *or 4*
	Internet gateway via router 2 *or 4*
	192.168.4.0/24 via router 4
Router 4	192.168.4.0/24
	192.168.1.0/24 via router 1
	Internet gateway via router 1
	192.168.3.0/24 via router 3
	192.168.2.0/24 via router 3

Once all the routers go through another round of broadcasts, the complete table resembles this:

Router	Route Table
Router 1	192.168.1.0/24
	Internet gateway
	192.168.2.0/24 via router 2 or 4
	192.168.3.0/24 via router 4 or 2
	192.168.4.0/24 via router 4 or 2
Router 2	192.168.2.0/24
	192.168.1.0/24 via router 1 or 3
	Internet gateway via router 1 or 3
	192.168.3.0/24 via router 3 or 1

Router	Route Table
Router 3	192.168.3.0/24
	192.168.2.0/24 via router 2 or 4
	192.168.1.0/24 via router 2 or 4
	Internet gateway via router 2 or 4
	192.168.4.0/24 via router 4 or 2
Router 4	192.168.4.0/24
	192.168.1.0/24 via router 1 or 3
	Internet gateway via router 1 or 3
	192.168.3.0/24 via router 3 or 1
	192.168.2.0/24 via router 3 or 1

Why is this multipath and mesh-like structure important? We'll explain the importance with an example: Let's say Router 2 fails and is offline. If Router 3 was previously relying on Router 2 to send packets to the Internet, it can immediately update its tables, reflecting that Router 2 is no longer available, and then forward Internet-bound packets through Router 4.

RIP's Algorithm (and Why You Should Use OSPF Instead)

Unfortunately, when it comes to figuring out the most optimal path from one subnet to another, RIP is not the smartest protocol. Its method of determining which route to take is based on the fewest number of routers (hops) between it and the destination. Although that sounds optimal, what this algorithm doesn't take into account is how much traffic is on the link or how fast the link is.

Looking back at Figure 12-10, you can see where this situation might play itself out. Let's assume that the link between Routers 3 and 4 becomes congested. Now if Router 3 wants to send a packet out to the Internet, RIP will still evaluate the two possible paths (3 to 4 to 1, and 3 to 2 to 1) as being equidistant. As a result, the packet may end up going via Router 4 when, clearly, the path through Router 2 (whose links are not congested) would be much faster!

OSPF (Open Shortest Path First) is similar to RIP in how it broadcasts information to other routers. What makes it different is that instead of keeping track of how many hops it takes to get from one router to another, it keeps track of how quickly each router is talking to the others. Thus, in our example, where the link between Routers 3 and 4 becomes congested, OSPF will realize that and be sure to route a packet destined to Router 1 via Router 2.

Another feature of OSPF is its ability to realize when a destination address has two possible paths that would take an equal amount of time. When it sees this, OSPF will share the traffic across both links—a process called *equal-cost multipath*—thereby making optimal use of available resources.

There are two "gotchas" with OSPF: Older networking hardware and some lower end networking hardware might not have OSPF available. The second gotcha is complexity: RIP is much simpler to set up than OSPF. For a small network, RIP may be a better choice at first.

tcpdump Bits and Bobs

The **tcpdump** tool is truly one of the more powerful network troubleshooting tools at your disposal as a system administrator. The GUI equivalent of it, Wireshark, is an even better choice when a graphical front-end is available. Wireshark offers all of the power of **tcpdump**, with the added bonus of richer filters, additional protocol support, the ability to follow TCP connections quickly, and some handy statistics. This section covers some useful bits and bobs (odds and ends) about **tcpdump** while walking through a few examples.

Reading and Writing Dumpfiles

If you need to capture and save a lot of data, you'll want to use the **-w** option to write all the packets to disk for later processing. Here is a simple example:

```
[root@server:~]#   tcpdump -w /tmp/trace.pcap -i eth0
```

The **tcpdump** tool will continue capturing packets seen on the eth0 interface until the terminal is closed, the process is killed, or CTRL-C is pressed. The resulting file can be loaded by Wireshark or read by any number of other programs that can process **tcpdump**-formatted captures. The **tcpdump** packet format itself is referred to as "pcap."

TIP When the **-w** option is used with **tcpdump**, it is not necessary to issue the **-n** option to avoid DNS lookups for each IP address seen.

To read back the packet trace using **tcpdump**, use the **-r** option. When you're reading back a packet trace, additional filters and options can be applied to affect how the packets will be displayed. For example, to show only ICMP packets from a trace file and avoid DNS lookups (using the **-n** option) as the information is displayed, do the following:

```
[root@server:~]#   tcpdump -r /tmp/trace.pcap -n icmp
```

Capturing More or Less per Packet

By default, **tcpdump** limits itself to capturing the first 65,535 bytes of data from each packet. If you're just looking to track some flows and see what's happening on the wire, this is usually good enough. However, if you need to capture the entire packet for further decoding, you'll need to increase this value. Conversely, you might need to capture less of the packet, possibly to speed up the capture process.

To change the length of the packet that **tcpdump** captures, use the **-s *snaplen*** option. For example, to capture the first 1500 bytes of each packet and write it to disk, try this:

```
[root@server:~]#  tcpdump -w /tmp/dump.pcap -i eth0 -s 1500
```

Performance Impact

Taking a packet trace can have a performance impact, especially on a heavily loaded server. There are two parts to the performance piece: the actual capture of packets and the decoding/printing of packets.

The actual capture of packets, while somewhat costly, can be minimized with a good filter. In general, unless your server load is already extremely high or you're moving a lot of traffic (a lot being hundreds of megabits/second), this penalty is not too significant. The main cost comes from the overhead/penalty of moving packets from the kernel up to the **tcpdump** application, which requires both a buffer copy and a context switch.

The decoding/printing of packets, by comparison, is substantially more expensive. The decoding itself is a small fraction of the cost, but the printing is high. If your server is heavily loaded, you want to avoid printing for two reasons: it generates load to format the strings that are output, and it generates load to update your screen. The latter factor can be especially costly if you're using a serial console, since each byte sent over the serial port generates a high-priority interrupt (higher than the network cards) that takes a long time to process, because serial ports are comparatively so much slower than everything else. Printing decoded packets over a serial port can generate enough interrupt traffic to cause network cards to drop packets as they are starved for attention (cycles) from the main CPU!

To alleviate the stress of the decode/print process, use the **-w** option to write raw packets to disk. The process of writing raw packets is much faster and lower in cost than printing them in real time. Furthermore, writing raw packets means you skip the entire decode/print step, since that is done only when you need to read the packets.

In short, if you're not sure, use the **-w** option to write the packets to disk, copy them off to another machine, and then read them there.

Don't Capture Your Own Network Traffic

A common mistake made when using **tcpdump** is to log in via the network and then start a capture. Without the appropriate filter, you'll end up capturing your session packets, which, in turn, if you're printing them to the screen, can generate new packets, which get captured again, and so on. For example, a quick way to skip your own SSH traffic is simply to skip port 22 (the SSH port) in the capture, like so:

```
[root@server:~]# tcpdump not tcp port 22
```

If you want to see what *other* people are doing on port 22, add a filter that excludes only your host. For instance, if you want to exclude 192.168.1.8 (your host), you can write this:

```
[root@server:~]# tcpdump "not (host 192.168.1.8 and tcp port 22)"
```

(Note the addition of the quotation marks. This was done so as not to confuse the shell with the added parentheses, whose contents are meant for **tcpdump**.)

Troubleshooting Slow Name Resolution (DNS) Issues

tcpdump is a great resource for troubleshooting odd or intermittent network problems. Using a trace of the packets themselves, you can look at activity over a period of time and identify issues that might be masked by other activity on the system.

Let's assume for a moment that you are using the DNS server managed by your Internet service provider. Everything is working until one day, things seem to be acting up.

Specifically, when you visit a web site, the first connection seems to take a long time, but once connected, the system seems to run pretty quickly. Every couple of sites, the connection doesn't even work, but clicking browser "reload" button seems to do the trick. That means that DNS is working and connectivity is there. What gives?

Time to take a packet trace. Because this is web traffic, we know that two protocols are at work: DNS for the hostname resolution and TCP for connection setup. That means we want to filter out all the other noise and focus on those two protocols. Because there seems to be some kind of speed issue, getting the packet timestamps is necessary, so we don't want to use the **-t** option. Here's the result:

```
[root@server:~]#  tcpdump -nn port 80 or port 53
```

Now visit the desired web site. For this example, we'll go to http://linuxserverexperts.com using a web browser. Let's look at the first few UDP packets:

```
21:27:40 IP 172.16.0.100.4102 > 192.168.0.1.53: A? linuxserverexperts.com (31)
21:27:50 IP 172.16.0.100.4103 > 192.168.0.1.53: A? linuxserverexperts.com (31)
21:27:58 IP 192.168.0.1.53 > 172.16.0.100.4102: 1/4/4 A 174.120.8.130 (206)
```

That's interesting; we needed to retransmit the DNS request to get the IP address for the hostname. Looks like some kind of connectivity problem is happening here, because we do eventually get a response. What about the rest of the connection? Does the connectivity problem affect other activity?

```
21:27:58 172.16.0.100.3013 > 174.120.8.130.80: Flags [S],\
seq 1031:1031, win 57344, Length 0
21:27:58 174.120.8.130.80 > 172.16.0.100.3013: Flags [S],\
seq 192:192 ack 1031, win 5840, Length 0
21:27:58 172.16.0.100.3013 > 174.120.8.130.80: Flags [.], ack 1 win 58400\
21:27:58 172.16.0.100.3013 > 174.120.8.130.80: Flags [P],\
seq 1:17, ack 1 win 58400, Length 16
......<OUTPUT TRUNCATED>.......
21:27:58 172.16.0.100.3013 > 174.120.8.130.80: Flags [.], ack 2156, win 56511
21:27:58 172.16.0.100.3013 > 174.120.8.130.80: Flags [F],\
seq 94:94 ack 2156, win  58400, Length 0
21:27:58 174.120.8.130.80 > 172.16.0.100.3013: Flags [.], ack 95 win 5840
```

Clearly, the rest of the connection went quickly. Time to poke at the DNS server:

```
[root@server:~]$  ping 192.168.0.1
64 bytes from 192.168.0.1: icmp_seq=1 ttl=247 time=213.0 ms
....<OUTPUT TRUNCATED>....
10 packets transmitted, 5 received, 50% packet loss, time 9023ms
```

Wow! We're losing packets (50 percent packet loss), and the jitter on the wire is bad. This explains the odd DNS behavior. Time to look for another DNS server!

IPv6

IPv6, Internet Protocol version 6, is also referred to as *IPng*—Internet Protocol: the Next Generation. IPv6 offers many new features and improvements over its predecessor, IPv4, including the following:

- A larger address space
- Built-in security capabilities, such as Network layer encryption and authentication
- A simplified header structure
- Built-in auto-configuration capabilities and improved routing capabilities

IPv6 Address Format

IPv6 offers an increased address space, because it is 128 bits long (compared to the 32 bits for IPv4). Because an IPv6 address is 128 bits long, there are about 2^{128} or 3.4×10^{38} possible addresses available (compared to the 2^{32} or roughly 4 billion available for IPv4).

A human being representing or memorizing (without error) a string of digits that is 128 bits long on paper is not easy. Therefore, several abbreviation techniques exist that make it easier to represent or shorten an IPv6 address to make it more human-friendly. The 128 bits of an IPv6 address can be shortened by representing the digits in hexadecimal format. This effectively reduces the total length to 32 digits in hexadecimal. IPv6 addresses are written in groups of four hexadecimal numbers. The eight groups are separated by colons (:). Here's a sample IPv6 address:

```
0012:0001:0000:0000:2345:0000:0000:6789
```

The leading zeros of a section of an IPv6 address can be omitted; so, for example, the sample address can be shortened to this:

```
12:1:0000:0000:2345:0000:0000:6789
```

The rule also permits the previous address to be rewritten like this:

```
12:1:0:0:2345:0:0:6789
```

One or more consecutive four-digit groups of zeros in an IPv6 address can be shortened and represented by double colon symbols (::), as long as this is done only once in the entire address. Therefore, using this rule, our sample address can be abbreviated to this:

```
12:1::2345:0:0:6789
```

Using the proviso in the rule would make the following address *invalid* because there is now more than one set of double colons in use:

```
12:1::2345::6789
```

IPv6 Address Types

There are several types of IPv6 addresses. Each address type has additional special address types, or scopes, that are used for different things. Three particularly special IPv6 address classifications are *unicast, anycast,* and *multicast* addresses.

Unicast Addresses

A unicast address in IPv6 refers to a single network interface. Any packet sent to a unicast address is meant for a specific interface on a host. Examples of unicast addresses are link-local (such as ::/128, an unspecified address; ::1/128, a loopback address; and fe80::/10, an auto-configuration address), global unicast, site-local, and other special addresses.

Anycast Addresses

An anycast address is a type of IPv6 address that is assigned to multiple interfaces (possibly belonging to different hosts). Any packet sent to an anycast address will be delivered to the closest interface that shares the anycast type address—"closest" is interpreted according to the routing protocol's idea of distance or simply the most easily accessible host. Hosts in a group sharing an anycast address have the same address prefix.

Multicast Addresses

An IPv6 multicast-type address is similar in functionality to an IPv4-type multicast address. A packet sent to a multicast address will be delivered to all the hosts (interfaces) that have the multicast address. The hosts (or interfaces) that make up a multicast group do not need to share the same prefix and do not need to be connected to the same physical network.

IPv6 Backward Compatibility

The designers of IPv6 built in backward-compatibility functionality into IPv6 to accommodate the various hosts or sites that are not fully IPv6 compliant or ready. The support for legacy IPv4 hosts and sites is handled in several ways: mapped addresses (IPv4-mapped IPv6 address), compatible addresses (IPv4-compatible IPv6 address), and tunneling.

Mapped Addresses

Mapped addresses are special unicast-type addresses used by IPv6 hosts. They are used when an IPv6 host needs to send packets to an IPv4 host via a mostly IPv6 infrastructure. The format for a mapped IPv6 address is as follows: the first 80 bits are all 0's, followed by 16 bits of 1's, and then it ends with 32 bits of the IPv4 address.

Compatible Addresses

The compatible type of IPv6 address is used to support IPv4-only hosts or infrastructures—that is, those that do not support IPv6 in any way. It can be used when an IPv6 host wants to communicate with another IPv6 host via an IPv4 infrastructure. The first 96 bits of a compatible IPv6 address are made up of all 0's, and it ends with 32 bits of the IPv4 address.

Tunneling

This method is used by IPv6 hosts that need to transmit information over a legacy IPv4 infrastructure using configured tunnels. This is achieved by encapsulating an IPv6 packet in a traditional IPv4 packet and sending it via the IPv4 network.

Summary

This chapter covered the fundamentals of TCP/IP and other protocols, ARP, subnetting and netmasks, and routing. It's a lot to digest, but hopefully this simplified version should make things easier to understand. Specifically, we covered TCP/IP's relation to the ISO OSI seven-layer model, the composition of a packet, the specifications of packet headers and how to explore them using **`tcpdump`**, the complete process of a TCP connection setup, data transfer, and connection teardown, and how static routing and dynamic routing work. We also looked at several packet analysis examples covering the use of **`tcpdump`**, and finally we had a brief overview of IPv6.

Because the information here is (substantially) simplified, you might want to take a look at some other books for more information regarding these topics. This is especially important if you have complex networks in which your machines need to participate or if you need to understand the operation of your firewall better.

One classic book we recommend is *TCP/IP Illustrated, Volume 1, Second Edition,* by W. Richard Stevens and Kevin R. Fall (Addison-Wesley Professional, 2012). This book covers TCP/IP in depth as well as several popular protocols that send their data over IP. Another recommended book is *TCP/IP Network Administration, Third Edition,* by Craig Hunt (O'Reilly Media, 2002).

As always, the manual pages for the various tools and utilities discussed will always be a good source of information. For example, the latest version of **`tcpdump`**'s documentation (man page) can be found at www.tcpdump.org/tcpdump_man.html.

CHAPTER

13 Network Configuration

Linux distributions come with a robust set of command-line and GUI tools for administrating the networking-related functions within the system. Invariably, the GUI tools are merely pretty front-ends for manipulating plain-text files or very simple data structures in the back-end.

Understanding how network configuration works under the hood in Linux distributions is invaluable and can come in handy in several scenarios. First and foremost is that when things are breaking and you can't start your favorite GUI, you'll find that being able to handle network configuration from the command-line interface (CLI) is crucial. Another benefit is remote administration: You might not be able to remotely run a graphical configuration tool easily. Issues such as firewalls and network latency will probably restrict your remote administration to the command line only. Finally, it's always nice to be able to manage network configuration through scripts, and command-line tools are well suited for scripting and automation.

In this chapter, we'll breeze through an overview of network interface drivers and then explore popular tools used for administering the network stack from the CLI—tools such as NetworkManager (NM), **`ip`**, and so on.

Modules and Network Interfaces

Network devices under Linux break the tradition of accessing all devices through the file abstraction layer. Not until the network driver initializes the interface and registers itself with the kernel does there exist a mechanism for anything to access the interface. Typically, Ethernet devices register themselves with names such as eth*X*, em*X*, ens*X*, pXp*X*, or enp*X*s*Y*, where *X* and *X* are the device or index numbers, respectively.

Depending on how your kernel was compiled, the device drivers for your network interface cards (NICs) may have been compiled as a module. For most distributions, this is the default mechanism, since it makes it much easier to probe for new hardware.

For legacy Linux distros, it might be necessary to tell the kernel the mapping between device names and the module to load. This was done by using (or creating) an appropriate

configuration file under the **/etc/modprobe.d/** directory. For example, if your eth0 device is an Intel PRO/1000 card that uses the e1000 device driver, you could add the following line to your /etc/modprobe.d/example.conf file:

```
alias eth0 e1000
```

Most modern Linux distros now use a mechanism called predictable device names. This is implemented using **udev** and other core subsystems. This is now the official mechanism for manipulating device names assigned to network devices such as Ethernet cards. It is useful in overcoming the occasional unpredictability with which the Linux kernel names and detects network devices.

You can find a listing of all the network device drivers that are installed for your running kernel under the **/lib/modules/`uname -r`/kernel/drivers/net** directory, like so:

```
[master@server ~]$  ls /lib/modules/`uname -r`/kernel/drivers/net
```

Note that backticks (versus single quotes) surround the embedded **uname-r** command, which returns your current (running) kernel version. This is handy when you have multiple kernel versions installed.

To glean more info about a particular driver without having to load the driver itself, use the **modinfo** command. For example, to see just the description of the **yellowfin.ko** driver, type the following:

```
[master@server ~]$ modinfo  yellowfin | grep  -i description
```

Keep in mind that not all drivers have descriptions associated with them, but most do.

Network Device Configuration Utilities (ip, ifconfig, and nmcli)

A number of graphical and menu-driven tools have been written to wrap around the various command-line tools that we discuss in the following sections. Many of these wrapper GUI programs ship with Linux distros that run and boot into full desktop graphical mode.

As an administrator, you should get familiar with and know how to configure network devices using any of the underlying command-line utilities directly. Knowing how to do this is invaluable, because, for example, many additional options not shown in GUIs are exposed in the command-line interface (CLI). To help you gain familiarity with these utilities, we will focus on using these command-line utilities to manipulate network settings in this section.

ip

ip is a powerful program that can be used to manage network devices on Linux-based systems. The **ip** utility comes with the **iproute** and **iproute2** software packages. The package contains networking utilities (such as **ip**) that are designed to be used to take advantage of and manipulate the advanced networking capabilities of the Linux kernel. The syntax for the **ip** utility is a little terser and less forgiving than that of other utilities discussed. But then again, the **ip** command is much more powerful!

ifconfig

The **ifconfig** program is one of several legacy utilities for setting up NICs. Although still popular, the use of **ifconfig** has become deprecated and people are encouraged to instead use the tools that ship with the **iproute** package (for example, **ip**).

Administrators who have used the Windows **ipconfig** program may see some similarities with **ifconfig**, as Microsoft implemented some CLI networking tools that mimic functional subsets of their UNIX counterparts.

TIP Administrators tasked with managing Windows servers may find the **%SYSTEMROOT%\system32\netsh.exe** program a handy tool for exposing and manipulating the details of Windows networking via the CLI.

nmcli

One of the more popular methods for managing network devices and connections in the Linux world is via the suite of tools that comes with the NetworkManager (NM) package. One such tool is the command-line **nmcli** utility, which can be used to control the NetworkManager daemon.

One distinguishing concept in the NM world is the notion of connections (or profiles). A *connection* or *profile* is a collection of related information that describes how to create or connect to a network. NM stores all network configurations (MAC addresses, IP addresses, DNS, gateways, and so on) as connections. A physical (or virtual) network device can use the information in a *connection* assigned to it for configuration purposes. You can define as many connections as you want to handle different environments, but remember that a connection is active only when a device uses that connection's configuration to create or connect to a network. In other words, multiple connections may apply to a device, but only one of them can be active on that device at any given time! NM connections have some flexibility in their names. Connections can be named after the interfaces associated with them or simply given more descriptive/functional names. These are all examples of valid NM connection names: eth0, Wired Connection, ens3, and foo-connection.

Sample Usage: ifconfig, ip, and nmcli

Let's walk through examples of using **ifconfig**, **ip**, and **nmcli** commands to view networking information as well as configure the network devices on our sample server. Wherever possible, we will show how to do the same thing using different methods. On a production system, you would of course want to standardize on one or two of the utilities.

Viewing IPv4 Connection and Device Information

To use **ifconfig** to display all the network interfaces available on the system, type:

```
[master@server ~]$ ifconfig  -a
```

To use **ip** to display all the network interfaces available on the system, type:

```
[master@server ~]$ ip link show
```

To use **ip** to display the IP configuration for the eth1 interface (if it exists), type:

```
[master@server ~]$  ip addr  show eth1
```

To use **nmcli** to display all the network devices available on the system, type:

```
[master@server ~]$ nmcli device status
```

Because **nmcli** understands the notion of a connection, query **nmcli** to view the current connections that it knows about:

```
[master@server ~]$ nmcli connection show
```

To use **nmcli** to display only the *active* connections (that is, connections with a device using it), type:

```
[master@server ~]$ nmcli connection show --active
```

To use **nmcli** to display the IP address associated with the sample ens3 device, type:

```
[master@server ~]$ nmcli  -f  ip4.address  device show ens3
```

To use **nmcli** to display the IP address associated with the sample connection named eth0, type:

```
[master@server ~]$ nmcli  -f ip4.address  connection show eth0
```

Configuring IPv4 Address and Connection Information

To use **ifconfig** in its simplest form to configure an IP address for an interface, all you need to do is provide the name of the interface being configured and the IP address. The **ifconfig** program will deduce the rest of the information from the IP address. Thus, you could enter the following:

```
[master@server ~]$ sudo ifconfig eth1 192.168.1.42
```

This will assign the eth1 device the IP address 192.168.1.42. Because 192.168.1.42 is a class C address, the calculated default netmask will be 255.255.255.0 and the broadcast address will be 192.168.1.255.

If the IP address you are setting is a class A, B, or C address that is subnetted differently, you may set the broadcast and netmask addresses explicitly on the command line, like so:

```
# ifconfig dev ip netmask nmask broadcast bcast
```

Here, ***dev*** is the network device you are configuring, ***ip*** is the IP address to which you are setting the device, ***nmask*** is the netmask, and ***bcast*** is the broadcast address.

The following example will assign the eth1 device the IP address 10.1.1.1 with a netmask of 255.255.255.0 and a broadcast address of 10.1.1.255:

```
[master@server ~]$ sudo ifconfig eth1 10.1.1.1 netmask 255.255.255.0\
 broadcast 10.1.1.255
```

To do the same thing using the **ip** command, you would type this:

```
[master@server ~]$ sudo ip address add 10.1.1.1/24 broadcast \
 10.1.1.255 dev eth1
```

If you want to accept the automatically determined broadcast address for the interface/network, you can leave out the **broadcast** option, like so:

```
[master@server ~]$ sudo ip address add 10.1.1.1/24 dev eth1
```

To use **ip** to delete the IP address created previously, type this:

```
[master@server ~]$ sudo ip address del 10.1.1.1/24 broadcast \
 10.1.1.255  dev eth1
```

To use **nmcli** to create a new *Ethernet* type *connection* arbitrarily named foobar-ip4 associated with the eth1 device and give it the IP address 10.1.1.1, type the following:

```
[master@server ~]$ sudo nmcli connection add con-name foobar-ipv4 \
ifname eth1 type ethernet ip4 10.1.1.1/24
```

NOTE On Red Hat–like systems, using **nmcli** to create a new connection will automatically create a new interface configuration file under the **/etc/sysconfig/network-scripts** directory. For example, a new Ethernet connection named foobar-connection will have a corresponding configuration file named **/etc/sysconfig/network-scripts/ifcfg-foobar-connection**.

If the previously created connection foobar-ipv4 is *not* yet active, make it active by running the following:

```
[master@server ~]$ sudo nmcli connection up foobar-ipv4
```

Use **nmcli** to delete the **foobar-ipv4 connection** as follows:

```
[master@server ~]$ sudo nmcli connection delete foobar-ipv4
```

ip and nmcli (Command Abbreviations and Auto-Completion)

The **ip** and **nmcli** commands allow unique abbreviations to be made in their syntax. Therefore, for example, you can shorten the command to assign an IP address to an interface from

```
# ip addr add 10.1.1.1/24 dev eno1
```

to something like this:

```
# ip a ad 10.1.1.1/24 dev eno1
```

And you can shorten the command to show all the NM connections using **nmcli** from

```
# nmcli connection show
```

to something like this:

```
# nmcli c  s
```

Additionally, both **ip** and **nmcli** offer the convenience of option and argument auto-completion. This means that once you enter the first few letters of an option to either command, you can press the TAB key to view a list of helpful suggestions that match the letters that you entered. This saves the trouble of having to remember or memorize the arcane options of the commands!

For example, entering the command

```
# ip addr[TAB]
```

and pressing the TAB key will output the following:

```
addr        addrlabel
```

This shows that there are two possible arguments that match the letters "addr."

Viewing IPv6 Address and Connection Information

To display the IPv6 addresses on all interfaces, you can use the **ip** command like so:

```
[master@server ~]$ ip -6 a show
```

To display the IPv6 address associated with the eno1 connection, you can use the **nmcli** command like so:

```
[master@server ~]$ nmcli -f ip6.address connection show eno1
```

Configuring IPv6 Address and Connection Information

To use the **ip** command to assign an IPv6 address (for example, 2001:DB8::1) to the interface eno2, you would use this command:

```
[master@server ~]$ sudo ip  -6 addr add  2001:DB8::1/64  dev  eno2
```

To use **ip** to delete the IPv6 address created previously, type this:

```
[master@server ~]$ sudo ip -6 addr del 2001:DB8::1/64  dev  eno2
```

The **ifconfig** command can also be used to assign an IPv6 address to an interface. For example, we can assign the IPv6 address 2001:DB8::3 to eth2 by running the following:

```
[master@server ~]$ sudo ifconfig eth2 inet6  add 2001:DB8::3/64
```

To use **nmcli** to add a new connection profile named foobar-ipv6 that specifies an IPv6 address for eth1 when active, type this:

```
[master@server ~]$ sudo nmcli connection add  con-name foobar-ipv6 \
 ifname eth1 type ethernet ip6   2001:DB8::1/64
```

Recall that a device can have only one connection active on it at a time. So, to activate the new **foobar-ipv6** connection on the eth1 device, you have to run this:

```
[master@server ~]$ sudo nmcli connection up foobar-ipv6
```

Activating a connection profile on a device effectively deactivates the current active connection on the interface device.

IP Aliasing

IP aliasing is used for assigning multiple IP addresses to a single network interface. In older Linux distros, aliasing is implemented by enumerating each instance of the same interface with a colon followed by a number—for example, eth0 is the main interface, eth0:0 is an aliased interface, eth0:1 is also an aliased interface, eth0:2 is another aliased interface, and so on.

Configuring an aliased interface is just like configuring any other interface: simply use the **ifconfig** or **ip** utility. For example, to use the **ip** utility to set eth0:0 with the address 10.0.0.2 and netmask 255.255.255.0, we can do the following:

```
[master@server ~]$ sudo ip a add  10.0.0.2/24 dev eth0:0
```

You can view your changes by typing the following:

```
[master@server ~]$ ip address show  eth0:0
```

(*continued*)

Note that network connections made *to* the aliased interface will communicate on the aliased IP address; however, in most circumstances, any connection originating from the host to another host will use the first assigned IP of the interface. For example, if eth0 is 192.168.1.15 and eth0:0 is 10.0.0.2, a connection from the machine that is routed through eth0 will use the IP address 192.168.1.15. The exception to this behavior is for applications that bind themselves to a specific IP address. In those cases, it is possible for the application to originate connections from the aliased IP address.

On newer Linux distributions that have the latest **`iproute`** package installed, you no longer have to use the previous interface naming conventions to bind multiple IP addresses to an interface. For a given interface, you now only have to repeat the correct **`ip`** command multiple times with the different IP addresses to achieve the same effect. For example, to bind the IP addresses 10.0.0.2/24 and 192.168.100.100/24 to the eth0 interface, you would run

```
[master@server ~]$ sudo ip a add  10.0.0.2/24 dev eth0
[master@server ~]$ sudo ip a add  192.168.100.100/24  dev eth0
```

Setting Up NICs at Boot Time

The landscape of solutions for managing networking in the Linux FOSS world is littered with the carcasses of various network managers—in other words, *many have come and gone*. The current approach includes NetworkManager, **systemd-networkd**, good ol' fashioned scripts, as well as a hybrid solutions that serve as a form of wrapper for any of the other tools!

The mainstream Linux distributions adopt slightly different approaches when it comes to automating and maintaining network configuration. We will cover specifics on the Red Hat–like distros (Fedora, CentOS, and so on) as well as Debian-like distros (Ubuntu, Mint, and so on) in the next section. For other distributions, you need to handle this procedure in one of two ways:

- Use the network administration tool that comes with that distribution to manage the network settings. This is probably the easiest and most reliable method.
- Find the startup script that is responsible for configuring network cards. At the end of the script, add the necessary **`ifconfig`**, **`ip`**, or **`nmcli`** statements. Another place to add commands is in the **rc.local** script or its equivalent—not as pretty, but it works!

Setting Up NICs Under Fedora, CentOS, and RHEL

Recent Fedora and other Red Hat–type releases currently favor using NetworkManager (NM) for configuring network cards at boot time. Fortunately, the use of NM also provides backward compatibility with the old way of network configuration management on those systems. The old way essentially involved the creation of files in the **/etc/sysconfig/network-scripts** directory that are read at boot time. NM can now transparently manage and create the files.

For each network interface, there is an **ifcfg** file in **/etc/sysconfig/network-scripts**. This filename is suffixed by the name of the device or the NetworkManager connection or profile; thus, **ifcfg-eth0** is for the eth0 device, **ifcfg-eth1** is for the eth1 device, and so on.

Here is a sample snippet of the **ifcfg-eth0** configuration file for the eth0 network interface that was automatically configured using Dynamic Host Configuration Protocol (DHCP) during the operating system installation:

```
TYPE="Ethernet"
BOOTPROTO="dhcp"
DEFROUTE="yes"
IPV6INIT="yes"
IPV6_AUTOCONF="yes"
IPV6_DEFROUTE="yes"
NAME="eth0"
UUID="28792a9c-57e2-4b76-9be9-21a5869976b9"
ONBOOT="yes"
```

If instead you choose to statically configure IP address information for the eth1 interface, during the OS installation, the corresponding configuration file, **ifcfg-eth1**, might look like this:

```
TYPE=Ethernet
ONBOOT=yes
BOOTPROTO=none
NAME=eth1
IPADDR=10.72.72.72
PREFIX=24
IPV6ADDR=2002:db8::1/64
```

These fields determine the IP configuration information for our sample **eth1** device. Note how some of the values correspond to parameters used in **`nmcli`** invocations.

To change the configuration information for a device, you can either directly modify the appropriate text configuration file **ifcfg-*** or use the **`nmcli`** command (or use a higher level GUI tool) with the proper options to automatically update or create the file.

After making any network configuration changes, you can quickly run the following sequence of commands to take down and bring back up the interface:

```
[master@fedora-server /]$ sudo ifdown eth0
[master@fedora-server /]$ sudo ifup eth0
```

To use **`nmcli`** to take the interface down and bring it back up, run the following:

```
[master@fedora-server /]$ sudo nmcli device disconnect eth0
[master@fedora-server /]$ sudo nmcli device connect eth0
```

TIP In Red Hat–like distros, the file **/usr/share/doc/initscripts/sysconfig.txt** provides information about the various configuration files under **/etc/sysconfig/**. For example, it shows the options and variables that can be used in the different **/etc/sysconfig/network-scripts/ifcfg-*** config files.

If you need to configure a second network interface card (for example, ens3), you can copy the syntax used in the existing **ifcfg-eth0** file by copying and renaming the **ifcfg-eth0** file to **ifcfg-ens3** and changing the information in the new **ifcfg-ens3** file to reflect the second network card's information. When doing this, you have to make sure that the **HWADDR** variable (media access control, or MAC, address) in the new file reflects the MAC address of the actual physical network device you are trying to configure. Once the new **ifcfg-ens3** file exists, Fedora will automatically consult the file to configure the device during the next boot or the next time the network service is restarted.

You can also restart the NetworkManager service to make your changes take effect, like this:

```
[master@fedora-server /]$ sudo systemctl restart NetworkManager
```

Setting Up NICs Under Debian-Like Systems (Ubuntu, Kubuntu, Mint, and So On)

Debian-based systems such as Ubuntu use a different mechanism for managing network configuration. Traditionally, network configuration is done via the **/etc/network/interfaces** file. We say "traditionally" because newer Ubuntu distros (versions 18.04 and later) are standardizing on **netplan** (discussed later on) for configuring networking.

The format of the **/etc/network/interfaces** file is simple and well documented. The entries in a sample **interfaces** file are discussed next (note that line numbers have been added to aid readability):

```
1. # The loopback network interface
2. auto lo
3. iface lo inet loopback
4.
5. # The first network interface eth0
6. auto eth0
7. iface eth0 inet static
8.           address 192.168.1.45
9.           netmask 255.255.255.0
10.          gateway 192.168.1.1
11.
12. # The second network interface ens5
13. auto ens5
14. iface ens5 inet dhcp
15. iface ens5 inet6 static
16.       address 2001:DB8::3
17.       netmask 64
```

- **Line 1** Any line that begins with the pound sign (**#**) is a comment and is ignored. Same thing goes for blank lines.
- **Line 2** Lines beginning with the word **auto** are used to identify the physical interfaces to be brought up when the **ifup** command executes, such as during

system boot or when the network run control script is run. The entry **`auto lo`** in this case refers to the loopback device. Additional options can be given on subsequent lines in the same stanza. The available options depend on the family and method.

- **Line 7** The **`iface`** directive defines the physical name of the interface being processed. In this case, it is the eth0 interface. The **`iface`** directive in this example supports the **`inet`** option, where **`inet`** refers to the address family. The **`inet`** option, in turn, supports various methods. Methods such as **`loopback`** (line 3), **`static`** (line 7), and **`dhcp`** (line 14) are supported. The **`static`** method here is simply used to define Ethernet interfaces with statically assigned IP addresses.
- **Lines 8–10** The static method specified in line 7 allows various options, such as address, netmask, gateway, and so on. The **`address`** option here defines the interface IP address (192.168.1.45), the **`netmask`** option defines the subnet mask (255.255.255.0), and the **`gateway`** option defines the default gateway (192.168.1.1).
- **Line 14** The **`iface`** directive defines the physical name of the interface being processed. In this case, it is the ens5 interface. The **`iface`** directive in this example supports the **`inet`** option, which is using the **`dhcp`** option. This means that the interface will be dynamically configured using DHCP.
- **Lines 15–17** These lines assign a static IPv6 address to the ens5 interface. The address assigned in this example is **`2001:DB8::3`** with the netmask **`64`**.

After making and saving any changes to the **interfaces** file, the network interface can be brought up or down using the **`ifup`** command. For example, after creating a new entry for the ens5 device, you would type this:

```
master@ubuntu-server:~$ sudo ifup  ens5
```

To bring down the eth1 interface, you would run this:

```
master@ubuntu-server:~$ sudo ifdown  ens5
```

The sample **interfaces** file discussed here is a simple configuration. The **/etc/network/interfaces** file supports a vast array of configuration options that were barely covered here. Fortunately, the man page (**`man 5 interfaces`**) for the file is well documented.

Setting Up NICs Using systemd-networkd

`systemd-networkd` is a **systemd**-esque service that can be used for automatically managing physical and virtual network devices (bridge, vlan, tun) on Linux hosts. It is currently the default underlying mechanism for managing networking configuration in some popular Linux distros.

Network configuration information is stored in INI-style files that end in **.network**, and virtual network devices are configured in **.netdev**-suffixed files. And these configurations can in turn be stored in the system network directory **/lib/systemd/network**, the volatile runtime network directory **/run/systemd/network**, and the local administration network directory **/etc/systemd/network**. The latter location has the highest precedence.

To demonstrate how it works, we'll use **systemd-networkd** to configure and manage a network interface on our demo server. Keep in mind that you only need one network manager running at a time, so we'll start off by disabling NM.

Ensure that NM is not currently running and then disable it from automatically starting up at the next system reboot:

```
[master@server ~]$ sudo systemctl stop NetworkManager
[master@server ~]$ sudo systemctl disable NetworkManager
```

Start the **sytemd-networkd** service and then ensure that it automatically starts up at the next system reboot:

```
[master@server ~]$ sudo systemctl start  systemd-networkd
[master@server ~]$ sudo systemctl enable systemd-networkd
```

Use the **networkctl** command to view what **systemd-networkd** thinks or knows about your network devices:

```
[master@server ~]$ networkctl
```

We are going to configure a sample **ens3** interface on our demo server. (You can verify the available interface names using the **ip link show** command.) Create a **networkd** configuration file named **ens3.network** under the **/etc/systemd/network** directory. We'll set up the interface with a static IP address, a default gateway, and DNS address:

```
[master@server ~]$ sudo tee  /etc/systemd/network/ens3.network << EOF
[Match]
Name=ens3

[Address]
Address=192.168.122.102/24

[Network]
DNS=8.8.8.8

[Route]
Gateway=192.168.122.1
EOF
```

Once the configuration file is created, restart the **networkd** service by running the following:

```
[master@server ~]$ sudo systemctl restart systemd-networkd
```

Use the **ip** and **networkctl** commands to view the effects of your changes.

Setting Up NICs Using netplan

With the plethora of network managers available to Linux distros, network configuration management can get a little hairy (and annoying). Same goes for keeping track of what's

doing what and learning the syntax and nuances of the various subsystems. **netplan** is the latest attempt to address these issues. The idea behind **netplan** is deceptively simple: it uses basic YAML (YAML Ain't Markup Language) configuration files to "rule them all." In this case, "all" refers to NM, **systemd-networkd**, and other network managers that might be available in the future. In **netplan**, the network manager is called the *renderer*. The **netplan** configuration file (or files) serves like a type of *manifest* for "other" systems (renderer) to consult and create their own native network configuration with. **netplan** is currently widely adopted only on Debian-like distros. Let's quickly review how to use **netplan** to set up a static IP address for an Ubuntu-based server.

Check that you have the **netplan** package installed on the system:

```
[master@ubuntu-server ~]$ hash netplan 2> /dev/null || \
echo "Netplan is NOT installed"
```

If the previous commands show "Netplan is NOT installed," you can install the **netplan** package on a Debian-based distro (like Ubuntu) by typing the following:

```
[master@ubuntu-server ~]$ sudo apt -y install netplan.io
```

To avoid any unexpected behaviors from existing or default **netplan** configuration files, move any existing files under the netplan configuration directory out of the way by renaming them:

```
[master@ubuntu-server ~]$ for i in  /etc/netplan/*.yaml; \
 do sudo mv "$i" "${i/yaml/bak}"; done
```

CAUTION The YAML file format is easy to read and understand. However, it is very unforgiving when you don't follow its formatting rules. YAML is all about key-value pairs, lists, maps, and Booleans. Here are some important YAML rules to keep in mind:

- Indentations or lack thereof can make or break a YAML file.
- Python-style indentation are used to indicate nesting.
- YAML uses spaces, never tabs!
- By convention only, most tools use two spaces for indentation.
- Each list member is denoted by a leading hyphen (-) or enclosed in square brackets ([ ]) and separated by comma and space (,).
- Associative arrays or maps are denoted by a colon and space (:) in the form of one "**key: value**" per line or enclosed in curly braces ({ }) separated by comma and space.

Let's create a simple **netplan** file named **10-mynetwork.yaml**.

This file contains directives for configuring two physical network interfaces (enp0s3 and enp1s0). We will configure the first interface, enp0s3, for DHCP. For the second interface, enp1s0, we will create a static IP address of 10.0.0.102/24, a gateway address, DNS server

addresses, and DNS search suffixes. We will also configure and use **systemd-networkd** as the renderer.

```
[master@ubuntu-server ~]$ sudo tee /etc/netplan/10-mynetwork.yaml << END
network:
  version: 2
  renderer: networkd
  ethernets:
    enp0s3:
      addresses: []
      dhcp4: true
    enp1s0:
      addresses:
        - 10.0.0.102/24
      gateway4: 10.0.0.1
      nameservers:
          search: [example.com, sub.example.com]
          addresses: [10.0.0.1, 8.8.8.8]
END
```

When done creating the file, use the **netplan** command to apply the changes to the system:

```
[master@ubuntu-server ~]$  sudo netplan apply
```

Use **ip** to confirm that the desired changes have been applied to the interface(s):

```
[master@ubuntu-server ~]$ ip address show
```

Because we specified **networkd** as the renderer, you can use the **networkctl** command to view the status of network links:

```
[master@ubuntu-server ~]$ networkctl
```

And that's **netplan** in a nutshell!

TIP If you don't explicitly specify a renderer in **netplan** YAML configuration files, **systemd-networkd** (**networkd**) will be used by default. To use NetworkManager as your renderer instead, ensure that the renderer line in the YAML file looks like this:

```
renderer: NetworkManager
```

Managing Routes

If your host is connected to a network with multiple subnets, you might need a *router* or *gateway* device to communicate with other hosts. This device sits between networks and redirects packets toward their actual destination. (Typically, most hosts don't know the correct path to a destination; they know only the destination itself.)

Whenever a host doesn't have any idea about where to send a packet, it uses its *default route*. This path points to a router, which ideally does have an idea of where the packet should go, or at least knows of another router that can make smarter decisions.

NOTE Traditionally, on Fedora, RHEL, and CentOS systems, it is also possible to set certain system-wide network-related values such as the default route, the hostname, NIS domain name, and so on, in the **/etc/sysconfig/network** file. The interface-specific settings are made in the appropriate **/etc/sysconfig/network-scripts/ifcfg-*** interface configuration files.

A typical single-homed Linux host knows of several standard routes. One of the standard routes is the *loopback route*, which simply points toward the loopback device. Another is the route to the local area network (LAN) so that packets destined to hosts within the same LAN are sent directly to them. Another standard route is the *default route*. This route is used for packets that are destined for other networks outside of the LAN. Yet another route that you might see in a typical Linux routing table is the *link-local route* (169.254.0.0). This is relevant in auto-configuration scenarios.

If you accepted the default network settings during the OS installation and allowed your network device to be automatically set up via DHCP, then your default route information would have been automatically provided to you. This can, of course, be changed after the fact.

Sample Usage: Route Configuration

In some instances, you will need to change your routes by hand. Typically, this is necessary when multiple network interfaces are available on the same host, where each NIC is connected to a different network (multi-homed). You should know how to add a route so that packets can be sent to the appropriate network for a given destination address.

The typical **route** command is structured as follows:

```
# route cmd type address netmask mask gw gway dev dn
```

The parameters are described as follows:

Parameter	Description
cmd	Either **add** or **del**, depending on whether you are adding or deleting a route. If you are deleting a route, the only other parameter you need is **address**.
type	Either **-net** or **-host**, depending on whether **address** represents a network address or a router address.
address	The destination network for which you want to specify a route.
netmask ***mask***	Sets the netmask of the **address** address to ***mask***.
gw ***gway***	Sets the gateway address for **address** to ***gway***. Typically used for the default route.
dev ***dn***	Sends all packets destined to **address** through the network device ***dn***.

Here's how to set the default route on a sample host that has a single Ethernet device and a default gateway at 192.168.1.1:

```
[master@server ~]$ sudo route add -net default gw 192.168.1.1 dev eth0
```

To add a default route to a system without an existing default route using the **ip** utility, you would type this:

```
[master@server ~]$ sudo ip route add default via 192.168.1.1
```

To set the default IPv6 route to point to the IPv6 gateway at the address 2001:db8::1 using the **ip** command, type this:

```
[master@server ~]$ sudo ip  -6 route add default via 2001:db8::1
```

To use the **ip** command to replace or change an existing default route on a host, you would use this:

```
[master@server ~]$ sudo ip route replace  default via 192.168.1.1
```

To use **ip** to set a host route to a host 192.168.2.50 via the ens2 interface, type this:

```
[master@server ~]$ sudo ip route add 192.168.2.50 dev ens2
```

To use the **ip** command to set up an IPv6 route to a network (for example, 2001::/24) using a specific gateway (such as 2001:db8::3), run this command:

```
[master@server ~]$ sudo ip -6 route add 2001::/24  via 2001:db8::3
```

Here's how to delete the route destined for 192.168.2.50:

```
[master@server ~]$ sudo route del 192.168.2.50
```

To delete using **ip**, you would type this:

```
[master@server ~]$ sudo ip route del  192.168.2.50  dev eth2
```

To delete an IPv6 route (for example, to 2001::/24 via 2001:db8::3) using the **ip** command, run this:

```
[master@server ~]$ sudo ip -6 route del 2001::/24  via 2001:db8::3
```

NOTE Don't just set routes arbitrarily on production systems without having a proper understanding of the network topology. Doing so can easily break the network connectivity. If you are using a gateway, you need to make sure a route exists to the gateway before you reference it for another route. For example, if your default route uses the gateway at 192.168.1.1, you need to be sure you have a route to get to the 192.168.1.0 network first.

Displaying Routes

Several utilities are available for viewing the system routing table from the command line. We discuss **route**, **netstat**, and **ip route** in the following sections.

route

Using **route** is one of the easiest ways to display your route table—simply run **route** without any parameters. Here is a complete run, along with the output:

```
[master@server ~]$ route
Kernel IP routing table
Destination  Gateway  Genmask         Flags  Metric  Ref   Use   Iface
10.10.2.0    0.0.0.0  255.255.255.0  U       0       0      0    eth0
192.168.1.0  0.0.0.0  255.255.255.0  U       0       0      0    eth1
169.254.0.0  0.0.0.0  255.255.0.0    U       0       0      0    eth0
0.0.0.0     my-gateway 0.0.0.0      UG      0       0      0    eth0
```

Here, you see two networks. The first is the 10.10.2.0 network, which is accessible via the first Ethernet device, eth0. The second is the 192.168.1.0 network, which is connected via the second Ethernet device, eth1. The third entry is the link-local destination network, which is used for auto-configuration hosts. The final entry is the default route. Its actual value in our example is 10.10.2.1; however, because the IP address resolves to the hostname "my-gateway" in Domain Name System (DNS), **route** prints its hostname instead of the IP address.

Note that **route** displayed the hostnames to any IP addresses it could look up and resolve. Although this is nice to read, it presents a problem when there are network outages and name resolution services (like DNS) are unavailable. The **route** command will appear to hang (until the request times out) while trying to resolve hostnames and waiting to see if the services come back online. To get around this, use the **-n** option with **route** so that the same information is shown, but **route** will make no attempt to perform hostname resolution on the IP addresses.

To view the IPv6 routes using the **route** command, type the following:

```
[master@server ~]$ route -A inet6
```

netstat

Normally, the **netstat** program is used to display the status of all of the network connections on a host. However, with the **-r** option, it can also display the kernel routing table.

Here is a sample invocation of **netstat -r** and its corresponding output:

```
[master@server ~]$ netstat -r
Kernel IP routing table
Destination   Gateway      Genmask        Flags  MSS  Window  irtt  Iface
192.168.1.0   0.0.0.0      255.255.255.0  U      0      0       0    eth0
127.0.0.0     0.0.0.0      255.0.0.0      U      0      0       0     lo
Default      192.168.1.1   0.0.0.0        UG     0      0       0     eth0
```

In this example, we see that the host has a single network interface card, is connected to the 192.168.1.0 network, and has a default gateway set to 192.168.1.1.

Like the **route** command, **netstat** can also take the **-n** parameter so that it does not perform hostname resolution.

To use the **netstat** utility to display the IPv6 routing table, you can run the following command:

```
[master@server ~]$ netstat -rn -A inet6
```

ip route

As mentioned, the **iproute** package provides advanced IP routing and network device configuration tools. The **ip** command can also be used to manipulate the routing table on a Linux host. This is done by using the **route** object with the **ip** command.

As with most commercial carrier-grade routing devices, a Linux-based system can actually maintain and use numerous routing tables at the same time. The **route** command that you saw earlier was actually displaying and managing only one of the default routing tables on the system—the **main** table.

For example, to view the contents of table **main** (as displayed by the **route** command), you would type this:

```
[master@server ~]$ ip  route show table main
10.10.2.0/24 dev eth0  proto kernel  scope link  src 10.99.99.45
192.168.1.0/24 dev eth2  proto kernel  scope link  src 192.168.1.42
169.254.0.0/16 dev eth0  scope link
default via 10.10.2.1 dev eth0 proto static metric 1024
```

To view the contents of all the routing tables on the system, type this:

```
[master@server ~]$ ip route show table all
```

To display only the IPv6 routes, type this:

```
[master@server ~]$  ip -6 route show
```

Network Configuration in the Cloud

The theoretical aspects of the networking stack and protocols discussed in Chapter 13 hold very true in any cloud implementation (Openstack, AWS, GCE, Linode, and so on). Therefore, you should try as much as possible to have a decent understanding of the basic concepts, such as subnets, routing, gateways, IP addressing, and so on.

Be that as it may, network configuration and management in cloud environments is a completely different beast. It's not a complicated or impossible beast to deal with; it's just different. The Linux servers you will be spinning up in the cloud are completely derived

(*continued*)

from the popular Linux distros, such as Fedora, Debian, Ubuntu, RHEL, CentOS, Amazon Linux, and OpenSUSE. This means that these virtual systems will most likely come pre-baked with the usual command-line utilities like—**ifconfig**, **ip**, **iptables**, **route**, **ip6tables**, and so on. Even though the tools are available and usable, keep in mind that using them willy-nilly might have unintended consequences! The worst-case scenario is that you will temporarily lose network connectivity to the virtual servers or network—but have no fear because you will almost always be able to regain access to your virtual world through the magic of application programming interfaces (APIs).

You will recall that, from your perspective, almost every object in the cloud is virtual—the routers, the switches, the firewalls, the servers, and so on. The only thing that is not virtual in any commercial cloud is the bill that comes due!

You'll have to rely on APIs, software development kits (SDKs), command-line tools, web consoles, and so on to interact with and manipulate the network objects in the cloud environment. An overly simplistic summary of the major components of networking in cloud environments consists of internal and external IP addresses, VPCs, and firewalls:

- **Internal IP addresses** are usually of the nonroutable RFC 1918 private address type, such as 172.16.0.0/12, 192.18.0.0/16 and 10.0.0.0/8. Cloud resources use these internal addresses to communicate privately with each and also to address each other.
- **External IP addresses** comprise the routable IP address class and are reachable via the public Internet. Externally facing cloud resources generally use external addresses to communicate with the rest of the public Internet.
- **Virtual private clouds (VPCs)** are used for creating (the illusion of) logically isolated sections on the cloud provider's network where you can launch and use various cloud resources. The VPC defines and represents the overall topology of your network in the cloud. By design, VPCs are global, meaning they can span multiple geographical locations. Every VPC usually has a primary classless inter-domain routing (CIDR) address block associated with it. Individual smaller subnets are then carved out of the larger VPC blocks to further segment resources. You have control over your virtual networking environment, including the selection of your own internal IP address ranges, the creation of subnets, and the configuration of route tables. You can use both IPv4 and IPv6 in your VPC for secure and easy access to resources and applications.
- **Firewalls** are implemented as rules or access controls lists (ACLs) that you define to help protect or secure your cloud assets at the network level. Some cloud implementations refer to firewall rules as security groups. Using ingress and egress rules, the firewalls can help to protect your internal IP addresses from your external IP addresses, or to protect other public external IP addresses from your external cloud IP addresses.

A Simple Linux Router

Linux has an impressive number of networking features, including its ability to act as a full-featured router. Wherever a powerful, low-cost router is needed, a standard PC running Linux with a few network cards can work quite nicely. A Linux router is able to move tons of data packets per second—depending, of course, on the CPU, the CPU cache, the type of NIC, the system bus speeds, and so on. In fact, several commercial routers exist that are running a stripped and optimized Linux kernel under their hood with a nice GUI administration front-end!

Routing with Static Routes

Let's assume that we want to configure a dual-homed Linux system as a router, as shown in Figure 13-1.

In this network, we want to route packets between the 192.168.1.0/24 network and the 192.168.2.0/24 network. The default route is through the 192.168.1.8 router, which is performing network address translation (NAT) to the Internet. (We discuss NAT in further detail in Chapter 14.) For all the machines on the 192.168.2.0/24 network, we want to set their default route to 192.168.2.1 and let the Linux router figure out how to forward on to the Internet and the 192.168.1.0/24 network. For the systems on the 192.168.1.0/24 network, we want to configure 192.168.1.15 as the default route so that all the machines can see the Internet and the 192.168.2.0/24 network.

This requires that our Linux system have two (physical or virtual) network interfaces: eth0 and eth1. We configure them as follows:

```
[master@server ~]$ sudo  ip address add 192.168.1.15/24 eth0
[master@server ~]$ sudo  ip address add 192.168.2.1/24 eth1
```

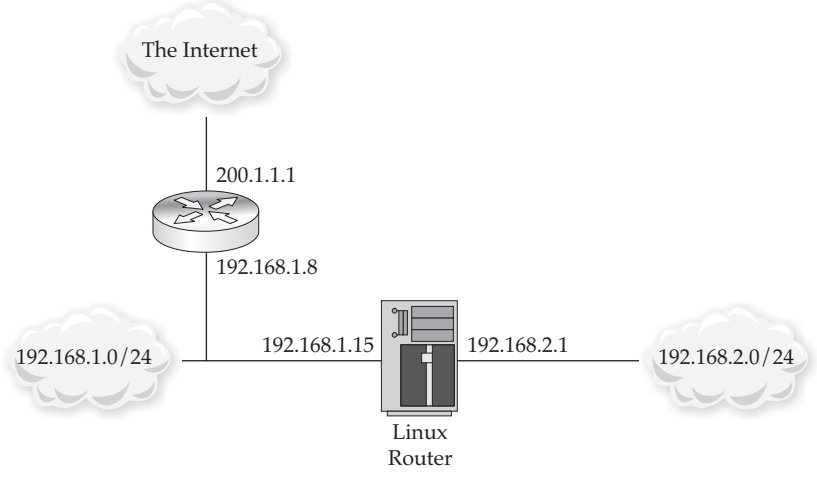


Figure 13-1 Our sample network

The result looks like this:

```
[master@server ~]$  ifconfig -a
eth0    Link encap:Ethernet HWaddr 00:30:48:21:2A:36
        inet addr:192.168.1.15 Bcast:192.168.1.255 Mask:255.255.255.0
....<OUTPUT TRUNCATED>....
eth1    Link encap:Ethernet HWaddr 00:02:B3:AC:5E:AC
        inet addr:192.168.2.1 Bcast:192.168.2.255 Mask:255.255.255.0
....<OUTPUT TRUNCATED>....
```

NOTE It is possible to configure a one-armed router where the eth0 interface is configured with 192.168.1.15 and eth0:0 is configured with 192.168.2.1. However, doing this will eliminate any benefits of network segmentation. In other words, any broadcast packets on the wire will be seen by both networks. Thus, it is usually preferable to put each network on its own physical interface.

When **ip** adds an interface, it also creates a route entry for that interface based on the netmask value. Thus, in the case of 192.168.1.0/24, a route is added on eth0 that sends all 192.168.1.0/24 traffic to it. Let's take a look at the routing table:

```
[master@server ~]$  route -n
Kernel IP routing table
Destination   Gateway  Genmask         Flags Metric Ref Use  Iface
192.168.2.0   0.0.0.0  255.255.255.0   U      0      0   0   eth1
192.168.1.0   0.0.0.0  255.255.255.0   U      0      0   0   eth0
127.0.0.0     0.0.0.0  255.0.0.0       U      0      0   0    lo
```

All that is missing here is the default route to 192.168.1.8. Let's add that using the **route** command and view the resulting updated table:

```
[master@server ~]$ sudo  route add default gw 192.168.1.8
[master@server ~]$ route -n
Kernel IP routing table
Destination  Gateway    Genmask         Flags Metric Ref Use  Iface
192.168.2.0  0.0.0.0    255.255.255.0   U      0      0   0    eth1
192.168.1.0  0.0.0.0    255.255.255.0   U      0      0   0    eth0
127.0.0.0    0.0.0.0    255.0.0.0       U      0      0   0    lo
0.0.0.0      192.168.1.8 0.0.0.0        UG     0      0   0    eth0
```

A quick check with **ping** verifies that we have connectivity through each route:

```
[master@server ~]$ ping -c 1 4.2.2.1
PING 4.2.2.1 (4.2.2.1) from 192.168.1.15 : 56(84) bytes of data.
64 bytes from 4.2.2.1: icmp_seq=1 ttl=245 time=15.2 ms
1 packets transmitted, 1 received, 0% loss, time 0ms
....<OUTPUT TRUNCATED>.....

[master@server ~]$ ping -c 1 192.168.1.30
PING 192.168.1.30 (192.168.1.30) from 192.168.1.15 : 56(84) bytes of data.
64 bytes from 192.168.1.30: icmp_seq=1 ttl=64 time=0.233 ms
....<OUTPUT TRUNCATED>.....
```

```
[master@server ~]$ ping -c 1 192.168.2.2
PING 192.168.2.2 (192.168.2.2) from 192.168.2.1 : 56(84) bytes of data.
64 bytes from 192.168.2.2: icmp_seq=1 ttl=64 time=0.192 ms
....<OUTPUT TRUNCATED>.....
```

Looks good. Now it's time to enable IP forwarding. This tells the Linux kernel that it is allowed to forward packets that are not destined to it, if it has a route to the destination. This can be done temporarily by setting **/proc/sys/net/ipv4/ip_forward** to 1, as follows:

```
[master@server ~]$ sudo echo "1" > /proc/sys/net/ipv4/ip_forward
```

Hosts on the 192.168.1.0/24 network should set their default route to 192.168.1.15, and hosts on 192.168.2.0/24 should set their default route to 192.168.2.1. Most importantly, don't forget to make the route additions and the enabling of **ip_forward** part of the startup scripts.

TIP Need a DNS server off the top of your head? For a quick query against an external reliable DNS server, try 4.2.2.1, 4.2.2.2, 4.2.2.3, 4.2.2.4, 4.2.2.5, 4.2.2.6, 8.8.8.8, or 8.8.4.4. These addresses have been around for a long time and are composed of a sequence of numbers that is easy to remember.

VPCs, Subnets, IPs, and Route Configuration (AWS Cloud Example)

By the way of an example, we'll walk through how to set up the network topology to launch and host a Fedora Linux server in a third-party cloud environment. Please note that besides the choice of tools and operating environment we've chosen in this section, there is nothing inherently Linux specific here. You'll feel right at home because the commands used here are very *Linux-y* or FOSS-like. You can follow almost the same exact steps to launch other supported non-FOSS (non-Linux) systems.

NOTE It might interest you to know that most cloud provider systems you will be indirectly interacting with are powered by some kind of Linux-based server! These are the servers doing all the heavy lifting by servicing your requests, hosting the APIs, hosting your virtual machines, doing your routing, hosting your applications, and so on.

We'll assume you have an authorized AWS account and that you've properly set up all the prerequisites (AWS account setup, billing, AWS access keys, AWS secret keys, tokens, AWS CLI tooling, and so on).

Using AWS as our sample cloud implementation, in the following sections we are going to create a VPC, create a subnet, make the subnet public, create firewall rules, and then finally launch an instance into the subnet.

NOTE Cloud providers make use of special identifiers consisting of a long, random string of letters and numbers to uniquely identify (or tag) the numerous resources available in their cloud. Amazon refers to these as resource IDs. Because these resource IDs are unique, we've deliberately masked the actual values throughout the following exercises in the sample outputs to avoid any confusion. For example, we've changed the sample value for our AWS-generated VPC ID from the actual vpc-0321z238 to vpc-*.

VPCs and Subnets (AWS)

The VPC is the overarching container that houses and gives access to all the cloud resources under AWS. To create a VPC that spans the 10.10.0.0/16 CIDR block, type this:

```
# aws ec2 create-vpc --cidr-block 10.10.0.0/16
{
    "Vpc": {
        "VpcId": "vpc-*",
 ...<OUTPUT TRUNCATED>...
```

Make note of the VPC ID (VpcId) in the output.

Create a subnet with a 10.10.0.0/24 CIDR block under the newly created VPC ID, as follows:

```
# aws ec2 create-subnet --vpc-id vpc-* --cidr-block  10.10.0.0/24
```

Internet Gateways and Routing (AWS)

The Internet gateway (IG) is used for facilitating communications between different networks. We are going to make the sole subnet in our VPC a public VPC. This can be achieved by attaching an IG to the VPC, creating a custom route table, and configure routing between the subnet and the IG, thereby giving resources in the VPC a way to reach the public Internet.

Let's create an Internet gateway:

```
# aws ec2 create-internet-gateway
{
    "InternetGateway": {
        "InternetGatewayId": "igw-*",
     ...<OUTPUT  TRUNCATED>...
```

Make note of the value of the Internet gateway ID (InternetGatewayId) in the output.

Attach the Internet gateway (igw-*) to the VPC:

```
# aws ec2 attach-internet-gateway --vpc-id vpc-* \

 --internet-gateway-id igw-*
```

To define a custom route table for the VPC, type the following:

```
# aws ec2 create-route-table --vpc-id vpc-*
{
    "RouteTable": {
        "RouteTableId": "rtb-*",
      ...<OUTPUT TRUNCATED>...
```

Make note of the value of the route table ID (RouteTableId) in the output.

Next, create a route in the newly created route table. This will be the default route (0.0.0.0/0), and we'll point it to the Internet gateway, like so:

```
# aws ec2 create-route --route-table-id rtb-* \
--destination-cidr-block 0.0.0.0/0 --gateway-id igw-*
```

Let's view the route that was just created by querying the custom route table:

```
# aws ec2 describe-route-tables --route-table-id rtb-*
{
    "RouteTables": [
        {
            "Associations": [],
            "RouteTableId": "rtb-*",
            "VpcId": "vpc-*",
            "Routes": [
               ........
                {
                    "GatewayId": "igw-*",
                    "DestinationCidrBlock": "0.0.0.0/0",
                    "State": "active",
                    "Origin": "CreateRoute"
...<OUTPUT TRUNCATED>...
```

The custom route table looks correct. However, in order to direct traffic from our subnet to the Internet gateway, we need to associate the route table with that subnet.

Let's query the VPC for the list of subnets we have defined:

```
# aws ec2 describe-subnets --filters "Name=vpc-id,Values=vpc-*"\
 --query 'Subnets[*].{ID:SubnetId,CIDR:CidrBlock}'
[
    {
        "CIDR": "10.10.0.0/24",
        "ID": "subnet-*"
    }
]
```

Make note of the value of the subnet ID (ID) in the output.

To make it a public subnet, let's tie the subnet with our custom route table:

```
# aws ec2 associate-route-table  --subnet-id subnet-* \
--route-table-id rtb-*
```

Security Groups (AWS)

Security groups act as virtual firewalls that control the traffic to and from cloud instances. The security group that will be assigned to an instance is defined at the time of launching the instance. You'll therefore need to punch holes in your virtual firewall so that resources or services can be reached from the public Internet.

Because we want to be able to connect to and manage our cloud Linux server, we'll first define a new security group in our VPC to permit SSH access from anywhere:

```
# aws ec2 create-security-group --group-name SSH \
 --description "Security group for SSH" --vpc-id vpc-*
{
    "GroupId": "sg-*"
}
```

Make note of the value of the security group ID (GroupId) in the previous output.

Create the firewall TCP/IP rules that will be enforced by the newly created security group:

```
# aws ec2 authorize-security-group-ingress --group-id sg-* \
--protocol tcp --port 22 --cidr 0.0.0.0/0
```

Launch a Linux Server in Its Own Subnet (AWS)

We are finally done with setting up the underlying network cruft for our demo environment. Assuming you've created and have access to an existing key pair named **THE_KEY_PAIR_NAME**, you should be able to provision a new virtual machine in the public subnet under your AWS account.

Let's test launching a new virtual machine image in the public subnet we just created. The Amazon Machine Image (AMI) identifier for the sample Red Hat Enterprise Linux 8 server that we want to launch is **ami-0520e698dd500b1d1**. Type the following:

```
# aws ec2 run-instances --image-id  ami-0520e698dd500b1d1 --count 1 \
--instance-type t2.micro --key-name <THE_KEY_PAIR_NAME> \
 --security-group-ids sg-* --subnet-id  subnet-*
```

Make note of the instance ID in the newly launched instance in the output.

To view various info about the new instance, type this:

```
# aws ec2 describe-instances --instance-id  i-*
```

Hostname Configuration

A system's hostname is the friendly name by which other systems or applications can address the system on a network. Configuring the hostname for a system is therefore considered an important network configuration task. You would have been prompted to create or choose a hostname for your system during the OS installation. If you didn't explicitly specify a hostname during the OS installation, it is possible that a hostname was automatically assigned.

You should take time to pick and assign hostnames to your servers that best describe their function or role. You should also pick names that can scale easily as your collection of servers grows. Examples of good (or simply playful) and descriptive hostnames are webserver01.example.org, dbserver09.example.com, logger-datacenterB, jupiter.example.org, saturn.example.org, pluto, sergent.example.com, hr.example.com, and major.example.org.

Once you've settled on a hostname and naming scheme, you next need to configure the system with the name. There is no standardized method for configuring the hostname among the various Linux distros, and so we offer the following recipes and pointers to different configuration files/tools that can be used on the popular Linux distros:

- **Fedora, CentOS, and RHEL** The hostname is set on these distributions by assigning the desired value to the **HOSTNAME** variable in the **/etc/sysconfig/network** file.
- **openSUSE and SLE** The hostname is set on these systems via the **/etc/HOSTNAME** file.
- **Debian, Ubuntu, and Kubuntu** Debian-based systems use the **/etc/hostname** file to configure the hostname of the system.
- **All Linux distributions** The **`sysctl`** tool can be used to temporarily change the system hostname on the fly on virtually all the Linux distros. The hostname value set using this utility will not survive system reboots. The syntax is

  ```
  # sysctl kernel.hostname=NEW-HOSTNAME
  ```

Modern Linux distros that have fully incorporated **systemd** as the service manager now support the notion of a single host having multiple hostnames for different purposes. Here are some different supported hostname classes:

- **Static hostname** This is the same as the traditional hostname used to initialize the kernel at boot. It is stored in the **/etc/hostname** file.
- **Transient hostname** This is a dynamic type of hostname that is subject to change during the system runtime. It can be changed, for example, via DHCP. It initially starts off being set to the same value as the static hostname.
- **Pretty hostname** This is a pretty, free-form version of the hostname. It supports the inclusion of special characters that the other hostname forms do not support. An example of a pretty hostname uses a space character is "KickAss Linux Server."

The tool for managing the different types of hostnames on **systemd** distros is **`hostnamectl`**. This tool can also be used for querying and configuring other metadata-type information that can be associated with a system, such as location, chassis, system icon, and even emoji.

To view the traditional/regular/static hostname on any Linux distribution, type the following:

```
[master@server ~]$ hostname
```

To use **`hostnamectl`** to view the *static* hostname on a system running **systemd** (such as Fedora or Ubuntu), type the following:

```
[master@server ~]$ hostnamectl status  --static
```

Use **`hostnamectl`** to view the *transient* hostname:

```
[master@server ~]$ hostnamectl status   --transient
```

To use **hostnamectl** to view the *pretty* hostname, type this:

```
[master@server ~]$ hostnamectl status   --pretty
```

Let's use **hostnamectl** to set a boring *static* hostname for the server:

```
[master@server ~]$ sudo hostnamectl set-hostname "fedora-server-01" --static
```

To use **hostnamectl** to set a transient hostname on a system running **systemd**, type the following:

```
[master@server ~]$ sudo hostnamectl  set-hostname \

 "fedora-server-01- temporary" --transient
```

Now we'll use **hostnamectl** to set a slightly more colorful *pretty* hostname that includes a smiley face emoji (☻ represented by Unicode code point **U+263B**):

```
 [master@server ~]$ sudo hostnamectl --pretty  set-hostname \
  "$(python3 -c "print('Kickass Server \u263b')")" --pretty
```

View all the three different classes of hostnames you just configured by typing this:

```
[master@server ~]$ hostnamectl status
```

To revert a hostname class to its default value, you can supply **""** (an empty string) as the argument to the **hostnamectl set-hostname** command. For example, to clear the current static hostname and reset it to its default value, type the following:

```
[master@server ~]$ sudo hostnamectl  set-hostname "" --static
```

Summary

In this chapter we covered how the **ifconfig**, **ip**, **nmcli**, and **route** commands can be used to configure the IP addresses (IPv4 and IPv6) and route entries (IPv4 and IPv6) on Linux-based systems. We looked at how this is done in Red Hat–like systems such as Fedora, CentOS, and RHEL, as well as in Debian-like systems such as Ubuntu. We explored how to use these commands together to build a simple Linux router. Brief mention was made of kernel modules in the context of network drivers. Remember that network interfaces don't follow the same method of access as most other devices with a **/dev** entry. We also looked at how IP/networking configuration can be done via a sample cloud (AWS) provider's API.

Keep in mind that when making IP address and routing changes, you should be sure to add any and all changes to the appropriate startup scripts and/or make sure that any network management stack (**networkd**, NM, and so on) in use is doing what you expect. You may want to schedule a reboot if you're on a production system, to make sure that the changes work as expected so that you don't get caught off guard later on.

If you're interested in more details on routing, it is worth taking a closer look at the next chapter and some of the advanced Linux routing features. Linux offers a rich set of functions that can be used to create powerful stand-alone appliances, routing systems, and networks.

For anyone interested in dynamic routing using Routing Information Protocol (RIP), Open Shortest Path First (OSPF), or Border Gateway Protocol (BGP), be sure to look into the Quagga project (www.nongnu.org/quagga). The project is dedicated to building and providing highly configurable dynamic routing systems/platforms that can share route updates with any standard router, including commercial hardware such as Cisco hardware. Another interesting network software emulation project is the GNS3 project found at https://www.gns3.com/.

CHAPTER

14 Linux Firewall (Netfilter)

Many, many moons ago, the Internet was a friendly place where people only wanted to sit around their mainframe computers while holding each other's hands, singing "Kum Ba Yah" and browsing the three or so available web sites. Everybody loved one another.

On a more serious note, the few original users of the Internet were focused on research and had better things to do than waste their time poking around other people's infrastructure. Any notion of security that was in place was largely to keep practical jokers from doing silly things. Many administrators made no serious effort to secure their systems. That was then, and this is now!

Unfortunately, as the Internet population grew, so did the threats. The need to put up barriers between the Internet and private networks started becoming increasingly commonplace.

The Linux networking, firewall, and packet filtering/mangling stacks have come a long way with these changes as well—from an initial implementation borrowed from Berkeley Software Distribution (BSD) through several major rewrites/updates (kernels 2.0, 2.2, 2.4, 2.6, 3.0, 4.0, and 5.0) and four user-level interfaces (`ipfwadm`, `ipchains`, `iptables`, and `nftables`). The current Linux packet filter and firewall infrastructure (both kernel and user tools) is referred to as Netfilter.

In this chapter, we start with a discussion of how Linux Netfilter works, follow up with a discussion of its relationship to Linux, and finish up with several configuration examples.

NOTE This chapter provides an introduction to the Netfilter system and demonstrates how firewalls work, with enough guidance to secure a simple network. Entire volumes have been written about firewalls. If you are interested in security beyond the scope of a basic configuration, you should definitely look into other dedicated resources that cover the subject in more depth.

How Netfilter Works

The idea behind Netfilter is simple: provide a simple means of making decisions on how a packet should flow. To make configuration easier, Netfilter supports command-line tools such as **`iptables`**, **`ip6tables`**, **`arptables`**, **`ebtables`**, **`nft`**, and so on.

The **`iptables`** tool manages Netfilter for Internet Protocol version 4 (IPv4). The **`iptables`** tool makes it easy to list, add, and remove rules as necessary from the system.

The command used to manage the IPv6 Netfilter subsystem is aptly named **`ip6tables`**. Most of the concepts about IPv4 Netfilter discussed in this chapter also apply to IPv6 Netfilter.

nftables is the new packet classification framework that will replace the **iptables** framework. It is implemented as a simple virtual machine that runs inside the kernel. In comparison to **iptables**, **nftables** offers faster firewall rule lookups, improved performance, a simpler code base, more efficient execution of filtering rules, more efficient storage of filtering rules, and so on.

nftables has support for existing **iptables** firewall rules via a compatibility layer that can provide translation of existing **iptables** firewall rules into their **nftables** equivalents. The user-space tool for interacting with **nftables** is called **`nft`**, and it can replace the functions previously served by several standalone **iptables** utilities/commands (such as **`iptables`**, **`ip6tables`**, **`arptables`**, and **`ebtables`**). Because **iptables** is probably the most widespread firewall implementation in the Linux world as of this writing, we will focus mostly on it, with brief mentions of **nftables** whenever appropriate.

In the Netfilter **iptables** implementation, all of the code that processes packets according to your configuration is actually run inside the kernel. To accomplish this, the Netfilter infrastructure breaks the task down into several distinct types of operations (tables): **nat** (network address translation), **mangle**, **raw**, **filter**, and **security**. Each operation has its own table of operations that can be performed based on administrator-defined rules; **nftables** has no such predefined tables.

The **nat** table is responsible for handling network address translation—that is, making or changing IP addresses to a particular source or destination IP address. The most common use for this is to allow multiple systems to access another network (typically the Internet) from a single IP address. When combined with connection tracking, network address translation is the essence of the Linux firewall.

The **mangle** table is responsible for altering or marking packets. The number of possible uses of the **mangle** table is enormous; however, it is also infrequently used. An example of its usage would be to change the ToS (Type of Service) bits in the TCP header so that Quality of Service (QoS) mechanisms can be applied to a packet.

The **raw** table is used mainly for dealing with packets at a very low level. It is used for configuring exemptions from connection tracking. The rules specified in the **raw** table operate at a higher priority than the rules in other tables.

The **filter** table is responsible for providing basic packet filtering. This can be used to allow or block traffic selectively according to whatever rules you apply to the system. An example of filtering is blocking all traffic except for that destined to port 22 (SSH) or port 25 (Simple Mail Transport Protocol, or SMTP).

Finally, the **security** table is used for managing Mandatory Access Control (MAC) networking rules.

A NAT Primer

Network address translation (NAT) allows you to hide hosts on both sides of a router so that each side can, for whatever reason, remain blissfully unaware of the other. NAT under Netfilter can be broken down into three categories: Source NAT (SNAT), Destination NAT (DNAT), and Masquerading.

SNAT is responsible for changing the source IP address and port to make a packet appear to be coming from an administrator-defined IP. This is most commonly used when a private network needs to use an externally visible IP address. To use SNAT, the administrator must know what the new source IP address is when the rule is being defined. If it is not known (for example, the IP address is dynamically defined by an Internet service provider [ISP]), the administrator should use Masquerading (explained later on). Another example of using SNAT is when an administrator wants to make a specific host on one network (typically private) appear as another IP address (typically public). SNAT, when used, needs to occur late in the packet-processing stages so that all of the other parts of Netfilter see the original source IP address before the packet leaves the system.

DNAT is responsible for changing the destination IP address and port so that a packet is redirected to another IP address. This is useful for situations in which administrators want to hide servers in a private network (typically referred to as a *demilitarized zone*, or *DMZ*, in firewall parlance) and map select external IP addresses to an internal address for incoming traffic. From a management point of view, DNAT makes it easier to manage policies, since all externally visible IP addresses are visible from a single host (also known as a *choke point*) in the network.

Finally, Masquerading is simply a special case of SNAT. This is useful for situations in which multiple systems inside a private network need to share a single dynamically assigned IP address to the outside world; this is the most common use of Linux-based firewalls. In such a case, Masquerading will make all the packets appear as though they have originated from the NAT-capable device's IP address, thus hiding the makeup of your private network. Using this method of NAT also allows your private network to use private IP spaces, such as 192.168.0.0/16, 172.16.0.0/12, 10.0.0.0/8, and so on.

Examples of NAT

Figure 14-1 shows a simple example where a host (192.168.1.2) is trying to connect to a server (200.1.1.1). Using SNAT or Masquerading in this case would apply a transformation to the packet so that the source IP address is changed to the NAT-capable device's external IP address (100.1.1.1). From the server's point of view, it is communicating with the NAT-capable device, not the host directly. From the host's point of view, it has unobstructed access to the public Internet. If multiple clients were behind the NAT-capable device (say, 192.168.1.3 and 192.168.1.4), the NAT engine would transform all their packets to appear as though they originated from 100.1.1.1 as well.

Alas, this raises a small problem. The server is going to send back some packets—but how does the NAT-capable device know which packet to send to whom? Herein lies the magic: The NAT engine maintains an internal list of client connections and associated server connections called *flows*. Thus, in the first example, the NAT engine is maintaining a record that "192.168.1.1:1025 converts to 100.1.1.1:49001, which is communicating with 200.1.1.1:80." When 200.1.1.1:80 sends a packet back to 100.1.1.1:49001, the NAT engine

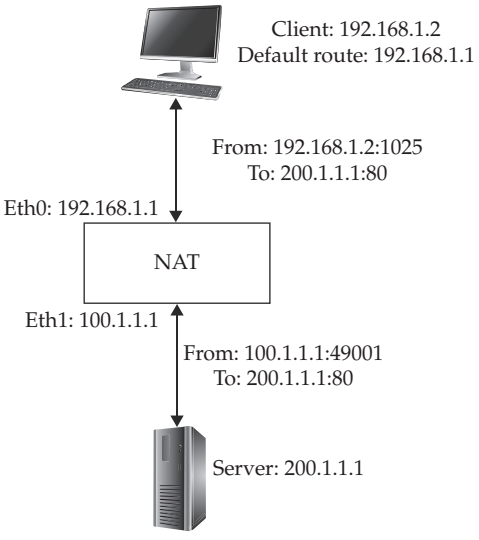


Figure 14-1 Using SNAT on a connection

automatically alters the packet so that the destination IP is set to 192.168.1.1:1025 and then passes it back to the client on the private network.

In its simplest form, a NAT engine is tracking only flows. Each flow is kept open so long as it sees traffic. If the NAT engine does not see traffic on a given flow for some time, the flow is automatically removed. These flows have no idea about the content of the connection itself, only that traffic is passing between two end point, and it is the job of NAT to ensure that the packets arrive as each end point expects.

Now let's look at the reverse case, as shown in Figure 14-2: a client from the Internet wants to connect to a server on a private network through a NAT-capable device. Using DNAT in this situation, we can make it the NAT engine's responsibility to accept packets on behalf of the server, transform the destination IP of the packets, and then deliver them to the server. When the server returns packets to the client, the NAT engine must look up the associated flow and change the packet's source IP address so that it reads from the NAT device rather than from the server itself. Turning this into the IP addresses shown in Figure 14-2, we see a server on 192.168.1.5:80 and a client on 200.2.2.2:1025. The client connects to the NAT IP address, 100.1.1.1:80, and the NAT engine transforms the packet so that the destination IP address is 192.168.1.5. When the server sends a packet back, the NAT engine does the reverse, so the client thinks that it is talking to 100.1.1.1. (Note that this particular form of NAT is also referred to as port address translation, or PAT.)

Connection Tracking and NAT

Although on the surface NAT appears to be a great way to provide security, unfortunately it is not enough. The problem with NAT is that it doesn't understand the contents of the flows and whether a packet should be blocked because it is in violation of the protocol. For example, assume that we have a network set up, as in Figure 14-2. When a new connection arrives for

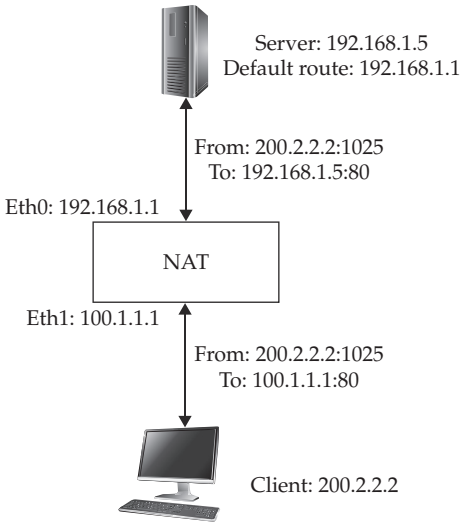


Figure 14-2 Using DNAT on a connection

the web server, we know that it must be a TCP SYN packet. There is no other valid packet for the purpose of establishing a new connection. With a blind NAT, however, the packet will be forwarded, regardless of whether or not it is a TCP SYN.

To make NAT more useful, Linux offers *stateful connection tracking*. This feature allows NAT to examine a packet's header intelligently and determine whether it makes sense from a TCP protocol level. Thus, if a packet arrives for a new TCP connection that is not a TCP SYN, stateful connection tracking will reject the packet without putting the server itself at risk. Even better, if a valid connection is established and a malicious person tries to spoof a random packet into the flow, stateful connection tracking will drop the packet unless it matches all of the criteria to be a valid packet between the two end point (a difficult feat, unless the attacker is able to sniff the traffic ahead of time).

As we discuss NAT throughout the remainder of this chapter, keep in mind that wherever NAT can occur, stateful connection tracking can occur.

Chains

For each table there exists a series of *chains* that a packet goes through. A chain is simply a list of rules that act on a packet flowing through the system.

There are five predefined chains in the **iptables** Netfilter implementation: PREROUTING, FORWARD, POSTROUTING, INPUT, and OUTPUT. Their relationship to one another is shown in Figure 14-3. You should note, however, that the relationship between TCP/IP and Netfilter, as shown in the figure, is purely logical.

Each of the predefined chains can invoke rules that are in one of the predefined tables (**nat**, **mangle**, or **filter**). Not all chains can invoke any rule in any table; each chain can invoke rules only in a defined list of tables. We will discuss what tables can be used from each chain when we examine what each of the chains does in the sections that follow.

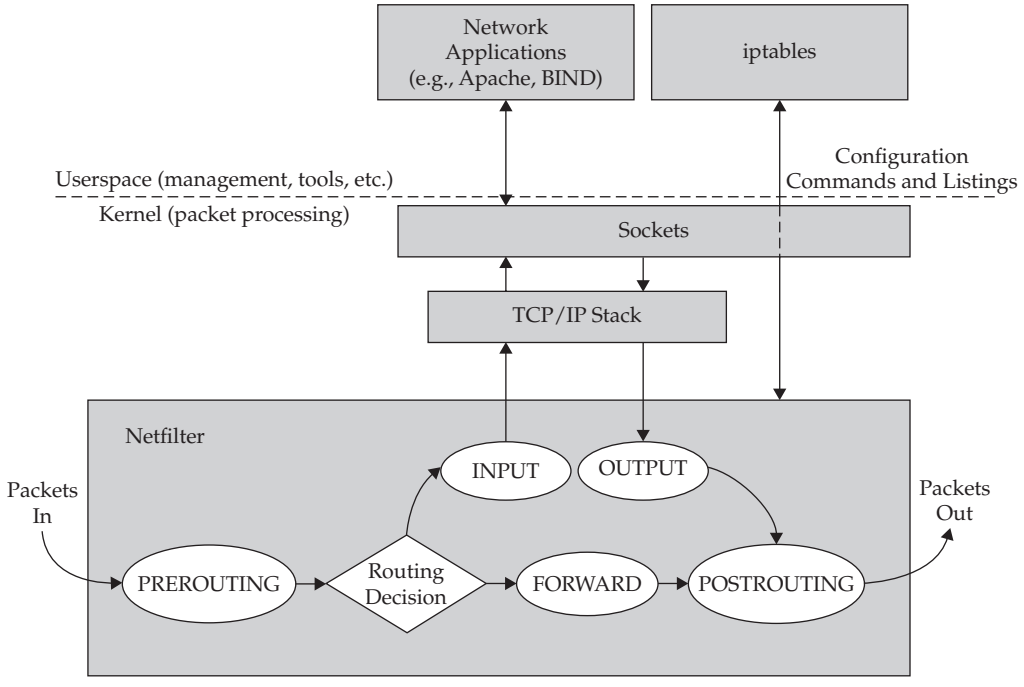


Figure 14-3 The relationship between the predefined chains in Netfilter

Administrators (or applications) can add more chains to the system if they want. A packet matching a rule can then, in turn, invoke another administrator-defined chain of rules. This makes it easy to repeat a list of rules multiple times from different chains. You will see examples of this kind of configuration later in the chapter.

All of the predefined chains are **members** of the mangle table. This means that at any point along the path, it is possible to mark or alter the packet in an arbitrary way. The relationship among the other tables and each chain, however, varies by chain. A visual representation of all of the relationships is shown in Figure 14-4. We'll cover each of these chains in more detail to help you understand these relationships.

There are no predefined chains in the **nftables** Netfilter implementation, but you have the convenience of being able to select from so-called *base chains*, which are chains that are registered in the **iptables** Netfilter hooks (PREROUTING, INPUT, FORWARD, OUTPUT, and POSTROUTING).

PREROUTING

The PREROUTING chain is the first thing a packet hits when entering the system. This chain can invoke rules in one of three tables: **nat**, **raw**, of **mangle**. From a NAT perspective, this is the ideal point at which to perform a Destination NAT (DNAT), which changes the destination IP address of a packet.

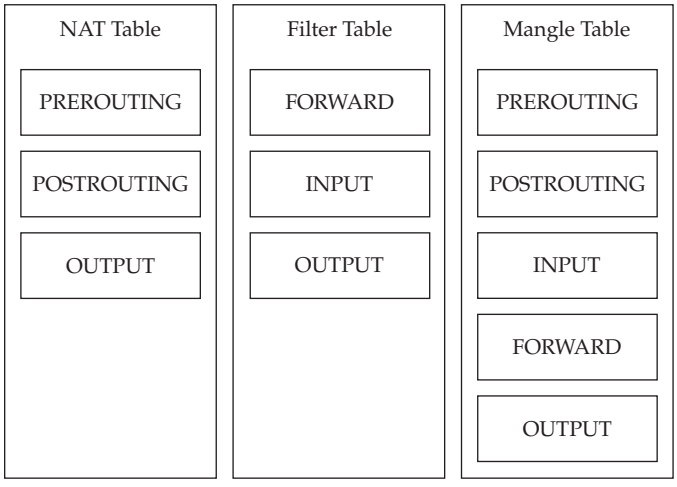


Figure 14-4 The relationship among predefined chains and predefined tables

Administrators looking to track connections for the purpose of a firewall should start the tracking here, since it is important to track the original IP addresses along with any NAT address from a DNAT operation.

FORWARD

The FORWARD chain is invoked only in the case when IP forwarding is enabled *and* the packet is destined for a system other than the host itself. For example, if the Linux system has the IP address 172.16.1.1 and is configured to route packets between the Internet and the 172.16.1.0/24 network, and a packet from 1.1.1.1 is destined to 172.16.1.10, the packet will traverse the FORWARD chain.

The FORWARD chain calls rules in the **filter** and **mangle** tables. This means that the administrator can define packet-filtering rules at this point that will apply to any packets to or from the routed network.

INPUT

The INPUT chain is invoked only when a packet is destined for the host itself. The rules that are run against a packet happen before the packet goes up the stack and arrives at the application. For example, if the Linux system has the IP address 172.16.1.1, the packet has to be destined to 172.16.1.1 in order for any of the rules in the INPUT chain to apply. If a rule drops all packets destined to port 80, any application listening for connections on port 80 will never see any such packets. The INPUT chain calls on rules in the **filter** and **mangle** tables.

OUTPUT

The OUTPUT chain is invoked when packets are sent from applications running on the host itself. For example, if a local user tries to use Secure Shell (SSH) on the command-line interface (CLI) to connect to a remote system, the OUTPUT chain will see the first packet

of the connection. The packets that return from the remote host will come in through PREROUTING and INPUT.

In addition to the filter and mangle tables, the OUTPUT chain can call on rules in the **nat** table. This allows administrators to configure NAT transformations to occur on outgoing packets that are generated from the host itself. Although this is atypical, the feature does enable administrators to do PREROUTING-style NAT operations on packets. (Remember, if the packet originates from the host, it never has a chance to go through the PREROUTING chain.)

POSTROUTING

The POSTROUTING chain can call on the **nat** and **mangle** tables. In this chain, administrators can alter source IP addresses for the purposes of Source NAT (SNAT). This is also another point at which connection tracking can happen for the purpose of building a firewall.

Installing Netfilter

The good news is that if you have a modern distribution of Linux, you should already have **`iptables`** Netfilter installed, compiled, and working. A quick check is to try running the **`iptables`** command, like so:

```
[root@server ~]# iptables -L
```

On an Ubuntu system, you would run this command instead:

```
master@ubuntu-server:~$ sudo iptables -L
```

A quick check to see the IPv6 equivalent is to use this command:

```
[root@server ~]# ip6tables -L
```

If the command gave you a list of chains and tables, you know that Netfilter is installed. In fact, there is a good chance that the OS installation process enabled some filters already! Some distros even provide an option to configure or customize a basic firewall at installation time. With Netfilter already present, you don't have much else to do besides actually configuring and using it.

The following sections offer some useful information about some of the options that can be used when setting up (from scratch) a vanilla kernel that does not already have Netfilter enabled. The complete process of installing Netfilter is actually two parts: enabling features during the kernel compilation process and compiling the administration tools. Let's examine the first part.

Enabling Netfilter in the Kernel

Most of Netfilter's code actually lives inside of the kernel and ships with the standard kernel.org distribution of Linux. On the off chance that Netfilter isn't already enabled in your distribution's kernel, you can enable it by turning on the right options during the kernel configuration step of compiling a kernel. If you are not familiar with the process of compiling a kernel, see Chapter 10 for details.

Netfilter has a lot of additional options that might not already be built into or enabled in your distro's vanilla kernel that supports standard Netfilter. So just in case you are in the market for building your kernel from scratch and want to use Netfilter's advanced option, just follow this truncated recipe: Download the kernel source, configure the kernel (via `make menuconfig` or similar) and then navigate to Networking Support → Networking Options → Network Packet Filtering Framework (Netfilter) and select or enable the desired options.

To enable support for the new `nftables` in your kernel, you will need the following to be enabled during kernel configuration: Network Packet Filtering, **nf_tables**, **nf_tables conntrack**, **IPv4 nf_tables**, and **IPv6 nf_tables**. Network Packet Filtering can be found under the Networking Support → Networking Options → Network Packet Filtering Framework (Netfilter) menu.

Core `nftable` support can be found under the Networking Support → Networking Options → Network Packet Filtering Framework (Netfilter) → Core Netfilter Configuration → Netfilter nf_tables Support menu.

Optional but Sensible Kernel Options

With the options just named compiled into the kernel, you technically have enough to make Netfilter work for most applications. However, a few more options can make life easier, provide additional security, and support some common protocols. For all practical purposes, you should consider these options as requirements. All of the following can be compiled as modules so that only those in active use are loaded into memory:

- **Connection State Match** With this, connection tracking gains the stateful functionality that was discussed in the section "Connection Tracking and NAT" earlier in the chapter. To reiterate, it allows the matching of packets based on their relationship to previous packets. This should be considered a requirement for anyone configuring a system as a firewall.
- **Packet Filtering** This option is required if you want to provide packet-filtering options.
- **REJECT Target Support** This option is related to the Packet Filtering option in that it provides a way of rejecting a packet based on the packet filter by sending an Internet Control Message Protocol (ICMP) error back to the source of a packet instead of just dropping it.
- **LOG Target Support** With this option, you can configure the system to log a packet that matches a rule. For example, if you want to log all packets that are dropped, this option makes it possible.
- **Full NAT** This option is a requirement to provide NAT functionality in Netfilter.
- **MASQUERADE Target Support** This option is a requirement to provide an easy way to hide a private network through NAT. This module internally creates a NAT entry.
- **REDIRECT Target Support** This option allows the system to redirect a packet to the NAT host itself. Using this option allows you to build transparent proxies, which are useful when it is not feasible to configure every client in your network with

proper proxy settings or if the application itself is not conducive to connecting to a proxy server.

- **NAT of Local Connections** This option allows you to apply DNAT rules to packets that originate from the NAT system itself. If you are not sure whether you'll need this later on, you should go ahead and compile it in.
- **Packet Mangling** This option adds the mangle table. If you think you'll want the ability to manipulate or mark individual packets for options such as Quality of Service, you should enable this module.

Configuring Netfilter

There is a good chance that your Linux distro of choice has already configured some Netfilter firewall rules for you out of the box, especially if you are using a relatively recent distribution. Although well intended, these default rules may not be to your liking and you will need to tweak them.

When it comes to tweaking, creating, or customizing firewall rules, some of your choices include using the good ol' command-line tools that are built for the job, using a GUI application, and learning how to manage the system using the existing set of scripts.

If you choose to go the GUI route, be aware that multiple GUIs are available for Linux in addition to the one that might have shipped with your system. The key to your decision, however, is that once you have made up your mind, you're going to want to stick to it. Although it is possible to switch between the GUI and CLI, it is not recommended, unless you know how to manage the configuration files that the GUI tools generates by hand.

Managing the system using the existing set of scripts requires the least amount of changing from a startup/shutdown script point of view, since you are using the existing framework; however, it also means getting to know how the current framework is configured and learning how to edit those files.

Finally, ignoring the existing scripts and going with your own might mean you need to start from scratch using command-line tools built for the job; you will have the benefit of knowing exactly how it works, when it starts, and how to manage it. The downside is that you will need to create all of the start and stop infrastructures as well. Because of the importance of the firewall functionality, it may not be enough to add the firewall rules to the end of a **/etc/rc.d/rc.local** (or similar) script that runs at the end of system bootup. Because of the time to boot, the window between starting a service and starting the firewall offers too much time for a potential attack to happen.

In the following sections we look at some front-end command-line tools (`iptables`, `nft`, `firewall-cmd`, and so on) that help with managing the back-end Netfilter stack.

Saving Your Netfilter Configuration

As you go through this chapter, you will create some custom firewall rules using the `iptables` commands, possibly tweak some settings in the **/proc** file system, and load additional kernel modules at boot time. To make these changes persistent across multiple reboots, you will need to save each of these components so that they start as you expect them to at boot time.

Saving under Fedora and other Red Hat–type Linux distributions is quite straightforward. Simply take the following steps:

1. Save your Netfilter rules to a sample plain-text file named **FIREWALL_RULES_FILE.txt** by running the following command:

   ```
   [root@fedora-server ~]# iptables-save  > FIREWALL_RULES_FILE.txt
   ```

   On a Fedora distro, the equivalent of our sample **FIREWALL_RULES_FILE.txt** file is **/etc/sysconfig/iptables** and so the command to run is this:

   ```
   [root@fedora-server ~]# iptables-save  > /etc/sysconfig/iptables
   ```

2. Add the appropriate modules to the **IPTABLES_MODULES** variable in the **/etc/sysconfig/iptables-config** file. For example, to add **ip_conntrack_ftp** and **ip_nat_ftp**, make the **IPTABLES_MODULES** line read as follows:

   ```
   IPTABLES_MODULES="ip_conntrack_ftp ip_nat_ftp"
   ```

TIP The configuration options for the IPv6 firewall (**ip6tables**) are stored in the **/etc/sysconfig/ip6tables-config** file. For example, the IPv6 equivalent of the **IPTABLES_MODULES** IPv4 directive is **IP6TABLES_MODULES** in the **ip6tables-config** file.

3. Make any changes to the kernel parameters as needed using the **sysctl** utility. For example, to enable IP forwarding, you would run the following command:

   ```
   [root@fedora-server ~]# sysctl -w net.ipv4.ip_forward=1 >> \
   /etc/sysctl.d/99-sysctl.conf
   ```

NOTE Some distributions already have commonly used kernel parameters defined (but disabled) in the **sysctl.conf** file, so you might simply need to change the existing variables to the desired value. So make sure you examine the file for the presence of the setting you want to change and tweak that value, instead of appending to the file as we did previously.

For other distributions, the methods discussed here may vary. If you aren't sure about how your distribution works, or if it's proving to be more headache than it is worth, simply disable the built-in scripts from the startup sequence and add your own. If you choose to write your own script, you can use the following outline:

```
#!/bin/sh
## Define where iptables and modprobe is located.
IPT="/sbin/iptables"
MODPROBE="/sbin/modprobe"
## Add your insmod/depmod lines here.
$MODPROBE ip_tables
$MODPROBE ipt_state
$MODPROBE iptable_filter
$MODPROBE ip_conntrack
$MODPROBE ip_conntrack_ftp
$MODPROBE iptable_nat
$MODPROBE ip_nat_ftp
## Flush the current chains, remove non-standard chains,
```

```
## and zero counters.
$IPT -t filter -F
$IPT -t filter -X
$IPT -t filter -Z
$IPT -t mangle -F
$IPT -t mangle -X
$IPT -t mangle -Z
$IPT -t nat -F
$IPT -t nat -X
$IPT -t nat -Z
## Add your rules here. Here is a sample one to get you started.
$IPT -A INPUT -i lo -j ACCEPT
## Add any /proc settings here. For example:
echo "1" > /proc/sys/net/ipv4/tcp_syncookies
```

The iptables Command

The **iptables** (and **ip6tables** for IPv6) command is the key to configuring the Netfilter system. A quick glance at its online help with the **iptables -h** command shows an impressive number of configuration options. In this section, we will walk through some of those options and learn how to use them.

At the heart of the command is the ability to define individual rules that are made a part of a rule chain. Each individual rule has a packet-matching criterion and a corresponding action. As a packet traverses a system, it will traverse the appropriate chains, as you saw in Figure 14-3 earlier in the chapter. Within each chain, each rule will be executed on the packet in order. When a rule matches a packet, the specified action is taken on the packet. These individual actions are referred to as *targets.*

Managing Chains

The format of the command varies by the desired action on the chain. These are the possible actions:

iptables -t *table* -A *chain* *rule-spec* [*options*]	Append ***rule-spec*** to ***chain***.
iptables -t *table* -D *chain* *rule-spec*	Delete ***rule-spec*** from ***chain***.
iptables -t *table* -I *chain* [*rulenum*] *rule-spec* [*options*]	Insert ***rule-spec*** at ***rulenum***. If no rule number is specified, the rule is inserted at the top of the chain.
iptables -t *table* -R *chain* *rulenum* *rule-spec* [*options*]	Replace ***rulenum*** with ***rule-spec*** on ***chain***.
iptables -t *table* -L *chain* [*options*]	List the rules on ***chain***.
iptables -t *table* -F *chain* [*options*]	Flush (remove all) the rules on ***chain***.

iptables -t *table* -Z *chain* *[options]*	Zero all the counters on ***chain***.
iptables -t *table* -N *chain*	Define a new chain called ***chain***.
iptables -t *table* -X *[chain]*	Delete ***chain***. If no chain is specified, all nonstandard chains are deleted.
iptables -t *table target* -P *chain*	Define the default policy for a chain. If no rules are matched for a given chain, the default policy sends the packet to ***target***.
iptables -t *table* -E *chain* *[new-chain]*	Rename ***chain*** to ***new-chain***.

Recall that there are several built-in tables (**nat**, **filter**, **mangle**, and **raw**) and five built-in chains (PREROUTING, POSTROUTING, INPUT, FORWARD, and OUTPUT), and that Figure 14-4 shows their relationships. As rules become more complex, however, it is sometimes necessary to break them up into smaller groups. Netfilter lets you do this by defining your own chain and placing it within the appropriate table.

When traversing the standard chains, a matching rule can trigger a jump to another chain in the same table. For example, let's create a chain called to_net10 that handles all the packets destined to the 10.0.0.0/8 network that is going through the FORWARD chain:

```
[root@server ~]# iptables -t filter -N to_net10
[root@server ~]#iptables -t filter -A FORWARD -d 10.0.0.0/8 -j to_net10
[root@server ~]# iptables -t filter -A to_net10 -j RETURN
```

In this example, the to_net10 chain doesn't do anything but return control back to the FORWARD chain.

TIP Every chain should have a default policy; that is, it must have a default action to take in the event a packet fails to meet any of the rules. When you are designing a firewall, the safe approach is to set the default policy (using the **-P** option in **iptables**) for each chain to be dropped and then explicitly insert ALLOW rules for the network traffic that you do want to allow.

To create a sample table named **to_net10** for the IPv6 firewall, we would use this:

```
[root@server ~]# ip6tables -t filter -N to_net10
```

To create a table named **to_net10** using the new **nftables**, you'd type this:

```
[root@server ~]# nft add table to_net10
```

TIP The **filter** table is the default table used whenever a table name is not explicitly specified with the **iptables** command. Therefore, the rule

```
# iptables -t filter -N example_chain
```

can also be written as

```
# iptables -N example_chain
```

Defining the Rule-Specification

The table in the preceding section referred several times to ***rule-spec*** (rule-specification). The rule-spec is the list of rules that are used by Netfilter to match on a packet. If the specified rule-spec matches a packet, Netfilter will apply the desired action on it. The following **iptables** parameters make up the common rule-specs.

- **-p** ***[!] protocol*** This specifies the IP protocol to compare against. You can use any protocol defined in the **/etc/protocols** file, such as **tcp**, **udp**, or **icmp**. A built-in value for "all" indicates that all IP packets will match. If the protocol is not defined in **/etc/protocols**, you can use the protocol number here. For example, **47** represents Generic Routing Encapsulation (GRE). The exclamation mark (**!**) negates the check. Thus, specifying **-p ! tcp** means all packets that are *not* TCP. If this option is not provided, Netfilter will assume "all." The **--protocol** option is an alias for this option. Here's an example of its usage:

  ```
  [root@server ~]# iptables -t filter -A INPUT -p tcp \
   --dport 80 -j ACCEPT
  ```

  For ip6tables, use this:

  ```
  [root@server ~]# ip6tables -t filter -A INPUT -p tcp \
   --dport 80 -j ACCEPT
  ```

  These rules will accept all packets destined to TCP port 80 on the INPUT chain.

- **-s** ***[!] address [/mask]*** This option specifies the source IP address to check against. When combined with an optional netmask, the source IP address can be compared against an entire netblock. As with **-p**, the use of the exclamation mark (**!**) inverts the meaning of the rule. Thus, specifying **-s ! 10.13.17.2** means all packets not from 10.13.17.2. Note that the address and netmask can be abbreviated. Here's an example of its usage:

  ```
  [root@server ~]# iptables -t filter -A INPUT -s 172.16/16 -j DROP
  ```

  This rule will drop all packets from the 172.16.0.0/16 network. This is the same network as 172.16.0.0/255.255.0.0.
  To use **ip6tables** to drop all packets from the IPv6 network range 2001:DB8::/32, we would use a rule like this:

  ```
  [root@server ~]#ip6tables -t filter -A INPUT -s 2001:DB8::/32 -j DROP
  ```

- **-d** ***[!] address [/mask]*** This option specifies the destination IP address to check against. When combined with an optional netmask, the destination IP address can be compared against an entire netblock. As with **-s**, the exclamation mark negates the rule, and the address and netmask can be abbreviated. Here's an example of its usage:

  ```
  [root@server ~]# iptables -t filter -A FORWARD \
  -d 10.100.93.0/24 -j ACCEPT
  ```

  This rule will allow all packets going through the FORWARD chain that are destined for the 10.100.93.0/24 network.

- **`-j target`** This option specifies an action to "jump" to. These actions are referred to as *targets* in **`iptables`** parlance. The targets that we've seen so far have been ACCEPT, DROP, and RETURN. The first two accept and drop packets, respectively. The third is related to the creation of additional chains.

  As you saw in the preceding section, you can create your own chains to help keep things organized and to accommodate more complex rules. If **`iptables`** is evaluating a set of rules in a chain that is not built-in, the RETURN target will tell **`iptables`** to return to the parent chain. Using the earlier to_net10 example, when **`iptables`** reaches **`-j RETURN`**, it goes back to processing the FORWARD chain where it left off. If **`iptables`** sees the RETURN action in one of the built-in chains, it will execute the default rule for the chain.

  Additional targets can be loaded via Netfilter modules. For example, the REJECT target can be loaded with **`ipt_REJECT`**, which will drop the packet and return an ICMP error packet back to the sender. Another useful target is REDIRECT (loaded with **`ipt_REDIRECT`**), which can make a packet be destined to the NAT host itself even if the packet is destined for somewhere else.

- **`-i interface`** This option specifies the name of the interface on which a packet was received. This is handy for instances when special rules should be applied if a packet arrives from a physical location, such as a DMZ interface. For example, if eth1 is your DMZ interface and you want to allow it to send packets to the host at 10.4.3.2, you can use this:

  ```
  [root@server ~]# iptables -A FORWARD -i eth1 -d 10.4.3.2 -j ACCEPT
  ```

- **`-o interface`** This option can also specify the name of the interface on which a packet will leave the system. Here's an example:

  ```
  [root@server ~]# iptables -A FORWARD -i enp2s0 -o eth1 -j ACCEPT
  ```

  In this example, any packets coming in from enp2s0 and going out to eth1 are accepted.

- *`[!]`* **`-f`** This option specifies whether a packet is an IP fragment or not. The exclamation mark negates this rule. Here's an example:

  ```
  [root@server ~]# iptables -A INPUT -f -j DROP
  ```

  In this example, any IP fragments coming in on the INPUT chain are automatically dropped. The same rule with negation logic is shown here:

  ```
  [root@server ~]# iptables -A INPUT ! -f -j ACCEPT
  ```

- **`-c PKTS BYTES`** This option allows you to set the counter values for a particular rule when inserting, appending, or replacing a rule on a chain. The counters correspond to the number of packets and bytes that have traversed the rule, respectively. For most administrators, this is a rare need. Here's an example of its usage:

  ```
  [root@server ~]# iptables -I FORWARD -f -j ACCEPT -c 10 10
  ```

  In this example, a new rule allowing packet fragments is inserted into the FORWARD chain, and the packet counters are set to 10 packets and 10 bytes.

- **-v** This option will display any output of **iptables** (usually combined with the **-L** option) to show additional data. Here's an example:

  ```
  [root@server ~]# iptables -L -v
  ```

- **-n** This option will display any hostnames or port names in their numeric form. Normally, **iptables** will do Domain Name System (DNS) resolution for you and show hostnames instead of IP addresses and show protocol names (such as SMTP) instead of port numbers (25). If your DNS system is down, or if you do not want to generate any additional packets, this is a useful option.

  Here's an example:

  ```
  [root@server ~]# iptables -L -n
  ```

- **-x** This option will show the exact values of a counter. Normally, **iptables** will try to print values in "human-friendly" terms and thus perform rounding in the process. For example, instead of showing "10310," **iptables** will show "10k."

  Here's an example:

  ```
  [root@server ~]# iptables -L -x
  ```

- ***-line-numbers*** This option will display the line numbers next to each rule in a chain. This is useful when you need to insert a rule in the middle of a chain and need a quick list of the rules and their corresponding rule numbers.

  Here's an example of its usage:

  ```
  [root@server ~]# iptables -L --line-numbers
  ```

  For IPv6 firewall rules, use this:

  ```
  [root@server ~]# ip6tables -L --line-numbers
  ```

Rule-Spec Extensions with Match

One of the most powerful aspects of Netfilter is the fact that it offers a "pluggable" design. For developers, this means that it is possible to make extensions to Netfilter using an application programming interface (API) rather than having to dive deep into the kernel code and hack away. This means a wide variety of extensions are available beyond the basic feature set.

These extensions are accomplished with the Match feature in the **iptables** command-line tool. By specifying a desired module name after the **-m** parameter, **iptables** will take care of loading the necessary kernel modules and then offer an extended command-line parameter set. These parameters are used to offer richer packet-matching features.

In this section, we discuss the use of a few of these extensions that, as of this writing, have been sufficiently well tested so that they are commonly included with Linux distributions.

TIP To get help for a match extension, simply specify the extension name after the **-m** parameter and then give the **-h** parameter. For example, to get help for the ICMP module, use this:

```
# iptables -m icmp -h
```

icmp This module provides an extra match parameter for the ICMP protocol:

```
icmp-type [!]typename
```

Here, ***typename*** is the name or number of the ICMP message type. For example, to block a ping packet, use the following:

```
# iptables -t filter -p icmp -A INPUT -m icmp --icmp-type  echo-request
```

For a complete list of supported ICMP packet types, see the module help page with the **-h** option.

limit This module provides a method of limiting the packet rate. It will match so long as the rate of packets is under the limit. A secondary "burst" option matches against a momentary spike in traffic but will stop matching if the spike sustains. The two parameters are

```
limit rate
limit-burst number
```

The ***rate*** is the sustained packet-per-second count. The ***number*** in the second parameter specifies how many back-to-back packets to accept in a spike. The default value for ***number*** is 5. You can use this feature as a simple approach to slowing down a SYN flood:

```
[root@server ~]# iptables -N syn-flood
[root@server ~]# iptables -A INPUT -p tcp --syn -j syn-flood
[root@server ~]# iptables -A syn-flood -m limit --limit 1/s -j RETURN
[root@server ~]# iptables -A syn-flood -j DROP
```

This will limit the connection rate to an average of one per second, with a burst up to five connections. This isn't perfect, and a SYN flood can still deny legitimate users with this method; however, it will help keep your server from spiraling out of control.

state This module allows you to determine the state of a TCP connection through the eyes of the conntrack module. It provides one additional option:

```
state state
```

Here, ***state*** is **INVALID**, **ESTABLISHED**, **NEW**, or **RELATED**. A state is **INVALID** if the packet in question cannot be associated to an existing flow. If the packet is part of an existing connection, the state is **ESTABLISHED**. If the packet is starting a new flow, it is considered **NEW**. Finally, if a packet is associated with an existing connection (such as an FTP data transfer), it is **RELATED**.

To use this feature to make sure that new connections have only the TCP SYN bit set, we do the following:

```
# iptables -A INPUT -p tcp ! --syn -m state --state NEW -j DROP
```

Reading this example, we see that for a packet on the INPUT chain that is TCP, which does not have the SYN flag set, and the state of a connection is **NEW**, we drop the packet. (Recall that legitimate new TCP connections must start with a packet that has the SYN bit set.)

tcp This module allows you to examine multiple aspects of TCP packets. We have seen some of these options (such as **--syn**) already. Here is a complete list of options:

- **--source-port [!]** ***port: [port]*** This option examines the source port of a TCP packet. If a colon followed by a second port number is specified, a range of ports is checked. For example, "6000:6010" means "all ports between 6000 and 6010, inclusive." The exclamation mark negates this setting. For example, **--source-port ! 25** means "all source ports that are not 25." An alias for this option is **--sport**.
- **--destination-port [!]** ***port: [port]*** Similar to the **--source-port** option, this examines the destination port of a TCP packet. Port ranges and negation are supported. For example, **--destination-port ! 9000:9010** means "all ports that are not between 9000 and 9010, inclusive." An alias for this option is **--dport**.
- **[!] --tcp-flags mask comp** This checks the TCP flags that are set in a packet. The **mask** parameter tells the option what flags to check, and the **comp** parameter tells the option what flags must be set. Both **mask** and **comp** can be a comma-separated list of flags. Valid flags are **SYN**, **ACK**, **FIN**, **RST**, **URG**, **PSH**, **ALL**, and **NONE**, where **ALL** means all flags and **NONE** means none of the flags. The exclamation mark negates the setting. For example, to use **--tcp-flags ALL SYN,ACK** means that the option should check all flags and only the SYN and ACK flags must be set.
- **[!] --syn** This checks whether the SYN flag is enabled. It is logically equivalent to **--tcp-flags SYN,RST,ACK SYN**. The exclamation point negates the setting.

The following example using this module checks whether a connection to DNS port 53 originates from port 53, does not have the SYN bit set, and has the URG bit set, in which case it should be dropped. Note that DNS will automatically switch to TCP when a request is greater than 512 bytes.

```
# iptables -A INPUT -p tcp  --sport 53 \
 --dport 53  ! --tcp-flags SYN URG  -j DROP
```

udp Like the tcp module, the udp module provides extra parameters to check for a packet. Two additional parameters are provided:

- **--source-port [!]** ***port:[port]*** This option checks the source port of a User Datagram Protocol (UDP) packet. If the port number is followed by a colon and another number, the range between the two numbers is checked. If the exclamation point is used, the logic is inverted.
- **--destination-port [!]** ***port:[port]*** Like the **source-port** option, this option checks the UDP destination port.

Here's an example:

```
[root@server ~]# iptables -I INPUT -p udp --destination-port 53 -j ACCEPT
```

This example will accept all UDP packets destined for port 53. This rule is typically set to allow traffic to DNS servers.

firewalld

firewalld is a dynamic firewall manager often used on Red Hat–based distributions like Fedora, CentOS, RHEL, and so on. Underneath its hood, it is actually just another nice front-end utility for managing Netfilter rules. It uses the concept of zones to assign trust levels to network interfaces. And because it is nicely integrated with NetworkManager (see Chapter 13), it is also able to assign trust levels to connections.

Through the magic of the system D-Bus, **firewalld** provides an interface (org.fedoraproject.FirewallD1) for other system services or applications to add firewall rules directly, as needed.

The primary configuration files for **firewalld** are stored under **/usr/lib/firewalld/** and **/etc/firewalld/**. The CLI utility for managing **firewalld** is **`firewall-cmd`**, and its GUI counterpart is called **`firewall-config`**. We'll examine some core **firewalld** concepts next.

Zones

In **firewalld** parlance, a *zone* defines the trust level for a connection. A connection can be assigned to only one zone at a time, but a zone can be used by many connections. The zone embodies the features of the firewall that are enabled, such as predefined services, ports and protocols, ICMP blocks, masquerading, and so on.

Even though you can create new zones as needed, **firewalld** conveniently ships with some predefined zones with features that are suitable for various environments. Some of the predefined zones are detailed here:

- **drop** This is the strictest or most untrusted zone. All incoming network packets are dropped. Only allows outgoing network connections.
- **block** Also a very strict/untrusted zone. Only network connections initiated within the system are permitted. The difference between the block zone and the drop zone is that the block zone offers the courtesy of responding with appropriate icmp-host-prohibited messages to incoming network connections.
- **public** Good for use in public areas. The other computers are generally not trusted. Only selected incoming connections are accepted.
- **external** For use on external networks with masquerading enabled, especially for routers. The other computers are generally not trusted. Only selected incoming connections are accepted.
- **dmz** For computers in so-called demilitarized zones that need to be publicly accessible with limited access to the internal network. Only selected incoming connections are accepted.

- **work** Designated for use in work areas. The other computers on networks are mostly trusted to not harm your system. Only selected incoming connections are accepted.
- **home** Earmarked for use in home areas. The other systems on networks are mostly trusted. Only selected incoming connections are accepted.
- **internal** For use on internal networks. You mostly trust the other computers on the networks to not harm your computer. Only selected incoming connections are accepted.
- **trusted** This is the least strict and most trusted zone. All network connections are accepted.

Services

A *service* is a predefined list of local ports, destinations, or firewall helper modules. The use of predefined services makes it easier to enable and disable access to a service. Sample services are **ssh**, **dhcp**, **dhcpv6**, **tftp**, **tftp-client**, **mysql**, **http**, **https**, and so on.

ICMP Types

The Internet Control Message Protocol (ICMP) suite is used mostly for diagnostic purposes in IP-based network devices and communications. There are several types of ICMP messages, and **firewalld** can use the predefined ICMP types to limit how much diagnostic information a system provides to potentially hostile systems. Sample ICMP types are **destination-unreachable**, **echo-reply**, **echo-request parameter-problem**, **redirect**, **router-advertisement**, and so on.

firewalld Examples

In this section, we'll interact with the **firewalld** daemon using the **`firewall-cmd`** utility.

First check that the **firewalld** daemon is up and running:

```
[master@fedora-server ~]$ firewall-cmd --state
```

If it isn't running, start it by running the following:

```
[master@fedora-server ~]$ sudo systemctl   start firewalld.service
```

Next, list the predefined zones:

```
[master@fedora-server ~]$ firewall-cmd --get-zones
```

To query **firewalld** for the current default zone, type this:

```
[master@fedora-server ~]$ firewall-cmd  --get-default-zone
```

To view the list of currently active zones with the interfaces assigned to them, type this:

```
[master@fedora-server ~]$ firewall-cmd --get-active-zones
```

Find out the zone that the **ens10** interface is assigned to:

```
[master@fedora-server ~]$ firewall-cmd --get-zone-of-interface=ens10
```

List the predefined services:

```
[master@fedora-server ~]$ firewall-cmd  --get-services
```

List the predefined ICMP types:

```
[master@fedora-server ~]$ firewall-cmd --get-icmptypes
```

Query **firewalld** for the services in the drop zone:

```
[master@fedora-server ~]$sudo firewall-cmd --list-services --zone=drop
```

List the services in the home zone:

```
[master@fedora-server ~]$sudo firewall-cmd --list-services --zone=home
```

List everything enabled in the default zone:

```
[master@fedora-server ~]$ sudo  firewall-cmd --list-all
```

List everything enabled in the external zone:

```
[master@fedora-server ~]$sudo firewall-cmd --list-all  --zone=external
```

Assuming the **https** service is currently not enabled in the home zone, use **`firewall-cmd`** to permanently enable it for that zone:

```
[master@fedora-server ~]$ sudo firewall-cmd –permanent \
 --add-service=https --zone=home
```

Reload the firewall rules without disrupting any current connections by running the following:

```
[master@fedora-server ~]$ sudo firewall-cmd --reload
```

Undo the previous command by removing the **https** service from the home zone:

```
[master@fedora-server ~]$sudo firewall-cmd  --remove-service=https \
  --zone=home
```

Forward all traffic on port 80 on your server to port 443 on another server whose IP address is 10.0.0.10 by running this:

```
[master@fedora-server ~]$ sudo firewall-cmd --zone=external  \
 --add-forward-port=port=80:proto=tcp:toaddr=10.0.0.10:toport=443
```

TIP Debian-based distributions such as Ubuntu can use a front-end program called Uncomplicated FireWall (**`ufw`**) for managing the **`iptables`**/Netfilter firewall stack. As its name implies, **`ufw`** is designed to make managing **`iptables`** rules easy (that is, uncomplicated).

Cookbook Solutions

Congratulations for making it this far into this chapter! We know it's a lot to take in initially—so many options, so many things to do, so little time!

Not to worry though, because we have your back. This section offers some cookbook solutions to common uses of the Linux Netfilter system that you can learn from and put to immediate use. Even if you didn't read the chapter and just landed here, you'll find some usable cookbook solutions. However, taking the time to understand what the commands are doing, how they are related, and how you can change them is worthwhile. You can also extend the few examples into endless possibilities.

With respect to saving the examples for use on a production system, you will want to add the **modprobe** commands to your startup scripts. In Fedora, CentOS, RHEL, and other Red Hat–type systems, add the module name to the **IPTABLES_MODULES** variable in **/etc/sysconfig/iptables-config**.

On Ubuntu/Debian-based Linux distros, you can add **modprobe** directives to the firewall configuration file **/etc/default/ufw**.

Fedora users can save their currently running **iptables** rule using the following **iptables-save** command:

```
[master@fedora-server ~]$ sudo iptables-save  > /etc/sysconfig/iptables
```

This will write the currently running **iptables** rules to the **/etc/sysconfig/iptables** configuration file.

The IPv6 equivalent of the command to write out the IPv6 firewall rules to the configuration file is shown here:

```
[master@fedora-server ~]$sudo ip6tables-save  > /etc/sysconfig/ip6tables
```

Other Linux distributions with Netfilter also have the **iptables-save** and **ip6tables-save** commands. The only trick is to find the appropriate startup file in which to write the rules.

Simple NAT: iptables

A common use for Linux firewalls is to make a network of systems available to the Internet via a single IP address. This is a common configuration in home and office networks where the Internet service provider (ISP) usually only allocates one IP address per customer. We walk through how to implement this simple solution in this section.

Assuming that you want to use your eth0 interface as your connection to the world and use your other interfaces (such as eth1) to connect to the inside network, run the following commands:

```
[master@server ~]$ sudo modprobe iptable_nat
[master@server ~]$ sudo iptables -t nat -A POSTROUTING \
-o eth0 -j MASQUERADE
[master@server ~]$ echo 1  | sudo tee  /proc/sys/net/ipv4/ip_forward
```

This set of commands will enable a basic NAT to the Internet. To add support for active FTP through this gateway, run the following:

```
[master@server ~]$ sudo modprobe ip_nat_ftp
```

If you are using Fedora, RHEL, or CentOS and want to make the **iptables** configuration part of your startup script, run the following:

```
[master@fedora-server ~]$ sudo iptables-save > /etc/sysconfig/iptables
```

TIP The **iptables-restore** and **ip6tables-restore** commands conversely allow you to easily restore your **iptables** rules from saved configurations files.

Simple NAT: nftables

NAT capabilities can be easily enabled on your Linux system using **nftables** and its configuration utility, **nft**. Say, for example, you want to use your enp0s3 interface as your connection to the world and use your other interface (such as enp0s8) to connect to the inside network.

First, list all the existing tables to check if the `nat` table already exists:

```
# nft list tables
```

Assuming the **nat** table is not shown as an output in the previous command, create the `nat` table by running the following:

```
# nft add table nat
```

Create a new chain named POSTROUTING under the **nat** table. This chain will use the postrouting hook.

```
# nft add chain nat postrouting \{ type nat hook postrouting priority  0 \; \}
```

Next add the new rule that will do the actual NATing. We want to do NAT for all traffic from the 10.11.11.0/24 network sent to our server and send it out through our enp0s3 interface with the source IP 192.168.99.1.

```
# nft add rule nat postrouting ip saddr 10.11.11.0/24 \
meta oif enp0s3 snat 192.168.99.1
```

List all the rules in the POSTROUTING chain under the `nat` table:

```
# nft list chain nat postrouting
```

Simple Firewall: iptables

In this section, we start with a deny-all firewall for two cases: a simple network where no servers are configured, and the same network, but with some servers configured. In the first case, we assume a simple network with two sides: inside on the 10.1.1.0/24 network (eth1) and the Internet (eth0). Note that by "server," we are referring to any node that can accept a connection made *to* it in order to provide service. This could, for example, mean a Linux system running an **ssh** daemon or a Windows system running a web server.

Let's start with the case where there are no servers to support.

First, we need to make sure that the NAT module is loaded and that FTP support for NAT is loaded. We do that with the **modprobe** commands:

```
[master@server ~]$ sudo modprobe iptable_nat
[master@server ~]$ sudo modprobe ip_nat_ftp
```

With the necessary modules loaded, we define the default policies for all the chains. For the INPUT, FORWARD, and OUTPUT chains in the filter table, we set the destinations to be DROP, DROP, and ACCEPT, respectively. For the POSTROUTING and PREROUTING chains, we set their default policies to ACCEPT. This is necessary for NAT to work.

```
[master@server ~]$ sudo iptables -P INPUT DROP
[master@server ~]$ sudo iptables -P FORWARD DROP
[master@server ~]$ sudo iptables -P OUTPUT ACCEPT
[master@server ~]$ sudo iptables -t nat -P POSTROUTING ACCEPT
[master@server ~]$ sudo iptables -t nat -P PREROUTING ACCEPT
```

With the default policies in place, we need to define the baseline firewall rule. What we want to accomplish is simple: Let users or nodes on the inside network (eth1) make connections to the Internet, but don't let the Internet make connections back. To accomplish this, we define a new chain called "block" that we use for grouping our state-tracking rules together. The first rule in that chain simply states that any packet that is part of an established connection or that is related to an established connection is allowed through. The second rule states that in order for a packet to create a new connection, it cannot originate from the eth0 (Internet-facing) interface. If a packet does not match against either of these two rules, the final rule forces the packet to be dropped.

```
[master@server ~]$ sudo iptables -N block
[master@server ~]$ sudo iptables -A block -m state \
--state ESTABLISHED,RELATED -j ACCEPT
[master@server ~]$ sudo iptables -A block -m state \
--state NEW ! -i  eth0 -j ACCEPT
[master@server ~]$ sudo iptables -A block -j DROP
```

With the blocking chain in place, we need to call on it from the INPUT and FORWARD chains. We aren't worried about the OUTPUT chain, since only packets originating from the firewall itself come from there. The INPUT and FORWARD chains, on the other hand, need to be checked. Recall that when doing NAT, the INPUT chain will not be hit, so we need to

have FORWARD do the check. If a packet is destined to the firewall itself, we need the checks done from the INPUT chain.

```
[master@server ~]$ sudo iptables -A INPUT -j block
[master@server ~]$ sudo iptables -A FORWARD -j block
```

Finally, as the packet leaves the system, we perform the MASQUERADE function from the POSTROUTING chain in the NAT table. All packets that leave from the eth0 interface go through this chain.

```
[master@server ~]$ sudo iptables -t nat -A POSTROUTING \
 -o eth0 -j MASQUERADE
```

With all the packet checks and manipulation behind us, we enable IP forwarding (a must for NAT to work) and SYN cookie protection, plus we enable the switch that keeps the firewall from processing ICMP broadcast packets (Smurf attacks)l

```
[master@server ~]$ echo 1 | sudo tee /proc/sys/net/ipv4/ip_forward
[master@server ~]$ echo 1 | sudo tee /proc/sys/net/ipv4/tcp_syncookies
[master@server ~]$ echo 1 | sudo tee \
/proc/sys/net/ipv4/icmp_echo_ignore_broadcasts
```

At this point, we have a working firewall for a simple environment without any servers that need to be accessible from outside. We can save this configuration and be done.

On the other hand, let's assume we have two applications that we want to make work through this firewall: a Linux system on the inside network that we need SSH access to from remote locations, and a Windows system from which we want to run BitTorrent. Let's start with the SSH case first.

To make a port available through the firewall, we need to define a rule that says, "If any packet on the eth0 (Internet-facing) interface is TCP and has a destination port of 22, change its destination IP address to 172.16.1.3." This is accomplished by using the DNAT action on the PREROUTING chain, since we want to change the IP address of the packet before any of the other chains see it.

The second problem we need to solve is how to *insert* a rule on the FORWARD chain that allows any packet whose destination IP address is 172.16.1.3 and destination port is 22 to be allowed. The key word is insert (**-I**). If we append the rule (**-A**) to the FORWARD chain, the packet will instead be directed through the block chain, because the rule **iptables -A FORWARD -j block** will apply first. To insert the proper rule, type the following:

```
$ sudo iptables -t nat -A PREROUTING -i eth0 -p tcp \
--dport 22 -j DNAT –to-destination  172.16.1.3
$ sudo iptables -I FORWARD -p tcp -d 172.16.1.3 --dport 22 -j ACCEPT
```

We can apply a similar idea to make BitTorrent work. Let's assume that the Windows machine that is going to use BitTorrent is 172.16.1.2. The BitTorrent protocol uses ports

6881–6889 for connections that come back to the client. Thus, we use a port range setting in the **iptables** command like so:

```
$ sudo iptables -t nat -A PREROUTING -i eth0 \
-p tcp --dport 6881:6889 -j DNAT --to-destination 172.16.1.2
$ sudo iptables -I FORWARD -p tcp -d 172.16.1.2 --dport 6881:6889 -j ACCEPT
```

Ta-da! We now have a working firewall and support for an SSH server and a BitTorrent user on the inside of our network.

Summary

In this chapter we discussed the ins and outs of the Linux firewall, Netfilter. In particular, we discussed the usage of the **iptables**, **ip6tables**, and **nft** (**nftables**) commands. With this information, you should be able to build, maintain, and manage a Linux-based firewall.

At the risk of stating the obvious, Netfilter is an impressively complex and rich system! People have written entire books dedicated to Netfilter and firewalls alone; therefore, you should consider our coverage of this subject in this chapter a gentle intro only—a way to whet your appetite. You can learn a lot more about Netfilter at the project web site found at www.netfilter.org.

CHAPTER

15 Local Security

We frequently hear about newly discovered attacks (or vulnerabilities) against various operating systems. A sometimes overlooked detail in discussions about these new attacks is the exploit vector. Exploit vectors can be of two types: those in which the vulnerability is exploitable over a network and those in which the vulnerability is exploitable locally. Local security and network security considerations require two different approaches. In this chapter, we focus on security from a local security perspective.

Local security addresses the problem of attacks that require the attacker to be able to do something on the system itself for the purpose of gaining elevated privileges.

Systems that lack proper local security controls can pose a real problem and invite attacks. Educational environments and schools are often ripe for these types of attacks: students may need access to servers to complete assignments and perform other academic work, but such a situation can be a threat to the system because when students get bored, they may test the bounds of their access and their own creativity, or they may sometimes not think about the consequences and impact of their actions.

Local security issues can also be triggered by network security issues. If a network security issue results in an attacker being able to invoke any program or application on the server, the attacker can use a local security–based exploit not only to give herself full access to the server, but also to escalate her own privileges to the root user. "Script kiddies"—attackers who use other people's attack programs because they are incapable of creating their own—are known to use these kinds of methods to gain unauthorized access to systems or more colloquially to *own* other people's systems.

This chapter addresses the fundamentals of keeping your system secure against common local security attacks. Keep in mind, however, that a single chapter on this topic will not make you an expert. Security is a field that is constantly evolving, and as such you should endeavor to also keep yourself abreast of latest developments and techniques in this space.

In this chapter, you will notice two recurring themes: mitigating risk and a "simpler is better" mantra. The former is another way of allocating your investment (both in time and money), given the risk you're willing to take on and the risk that a system or server poses

if compromised. And keep in mind that because you cannot prevent all attacks, you have to accept a certain level of risk—and the level of risk you are willing to accept will drive the investment in both time and money. So, for example, a web server dishing up your vacation pictures on a low-bandwidth link is a lower risk than a server handling large financial transactions for Wall Street!

The "simpler is better" mantra stems from Engineering 101—simple systems are less prone to problems, easier to fix, easier to understand, and inevitably more reliable. Keeping your servers simple is a desirable goal.

Common Sources of Risk

Security is the mitigation of risk. Along with every effort of mitigating risk comes an associated cost. Costs are not always necessarily financial; they can take the form of restricted access, loss of functionality, or loss of time. Part of your job as an administrator is to balance the costs of mitigating risk with the potential damage that an exploited risk can cause.

Consider a web server, for example. The risk of hosting a service that can be probed, poked at, and possibly exploited is inherent in running a public-facing or publicly accessible web server. However, you may find that the risk of exposure is low so long as the web server is maintained and immediately patched when security issues arise. If the benefit of running a web server is great enough to justify your cost of maintaining it, then it is a worthwhile endeavor. In this section, we look at common sources of risk and examine what you can do to mitigate those risks.

SetUID Programs

SetUID programs are executables that have a special attribute (flag) set in their permissions that allows users to run the executable in the context of the executable's owner. This enables administrators to make selected applications, programs, or files available with higher privileges to normal users, without having to give those users any administrative rights. An example of such a program is **`ping`**. Because the creation of raw network packets is restricted to the root user (the ability to create raw packets can allow the application to inject potentially bad payload within the packet), the **`ping`** application must run with the SetUID bit enabled and the owner set to root. Thus, for example, even though user *yyang* may start the **`ping`** program, the program can be run in the context of the *root* user for the purpose of placing an Internet Control Message Protocol (ICMP) packet onto the network. The **`ping`** utility in this example is said to be "SetUID root."

Developers of programs that need to run with root privileges have an obligation/responsibility to be extra security conscious. It should not be possible for a normal user to do something dangerous on the system by using such programs. This means many checks need to be written into the program and potential bugs must be carefully removed. Ideally, these programs should be small and single-purposed. This makes it easier to evaluate the code for potential bugs that can harm the system or allow for a user to gain privileges that he or she should not have.

From a day-to-day perspective, it is in the administrator's best interest to keep as few SetUID root programs on the system as possible. The risk balance here is the availability of features/functions to users versus the potential for bad things to happen. For some common programs such as **mount**, **traceroute**, and **su**, the risk is low for the value they bring to the system. Some well-known SetUID programs, such as the X Window System, pose a low to moderate risk; however, given X Window System's exposure, it is unlikely to be the root of any problems. If you are running a pure server environment and you do not need X Window System, it never hurts to remove it.

SetUID programs executed by web servers are almost always a bad thing. Use great caution with these types of applications and look for alternatives. The exposure is much greater, since it is possible for network input (which can come from anywhere) to trigger this application and affect its execution. If you find that you must run an application SetUID with root privileges, another alternative is to find out whether it is possible to run the application in a **chroot** environment (discussed later in this chapter in the section "chroot").

TIP An alternative to SetUID programs is a feature called *capabilities* provided by the kernel.

As an example, it is possible to make the popular **ping** program *non*-SetUID by instead assigning it just the precise capabilities (**CAP_NET_RAW**) that it needs to function. See the man pages for the following to learn more about capabilities: getcap, setcap, capabilities, and getpcaps. To view any special capabilities required by **ping** on any modern Fedora distro, type the following:

```
# getcap   `which ping`
```


Finding and Creating SetUID Programs

A SetUID program has a special file attribute that the kernel uses to determine whether it should override the default permissions granted to an application. A simple file system listing (**ls -l**) will show the permissions on a file and reveal this little fact. Here's an example:

```
[master@server ~]$  ls -l /usr/bin/mount
-rwsr-xr-x. 1 root root 44248 Oct 24 07:00 /usr/bin/mount
```

If the fourth character in the permissions field is **s**, the application is SetUID. If the file's owner is root, then the application is SetUID root. In the case of the **mount** binary, we can see that it will execute with root permissions available to it.

TIP You can use the **stat** utility to view the octal mode representation of file permissions. For example, to view the octal permission mode (4755) of the **mount** command, type the following:

```
$  stat -c '%A %a %n' `which mount`
-rwsr-xr-x   4755  /bin/mount
```

Another example is the **passwd** utility, shown here:

```
[master@server ~]$ ls -l  /usr/bin/passwd
-rwsr-xr-x. 1 root root 27864 Aug 17 17:50 /usr/bin/passwd
```

As with **mount**, we see that the fourth character of the permissions is **s** and the owner is root. The **passwd** program is, therefore, SetUID root.

To determine whether a running process is SetUID, you can use the **ps** command to see both the actual user of a process and its effective user, like so:

```
[master@server ~]$  ps ax -o pid,euser,ruser,comm
```

This will output all of the running programs with their process ID (**pid**), effective user (**euser**), real user (**ruser**), and command name (**comm**). If the effective user is different from the real user, it is likely a SetUID program.

NOTE Some applications that are (necessarily) started by the root user can *give up* their permissions to run as a less-privileged user to improve security. The Apache web server, for example, might be started by the root user to allow it to bind to TCP port 80 (recall that only privileged users can bind to ports lower than 1024), but it then gives up its root permissions and starts all of its threads as an unprivileged user (typically the user "nobody," "apache," "www-data," or "www").

Very rarely, you may need to make a program run as SetUID. To do this, use the **chmod** command. Prefix the desired permissions with a 4 to turn on the SetUID bit. (Using a prefix of 2 will enable the SetGID bit, which is similar to SetUID, but offers group permissions instead of user permissions.)

For example, if we have a program called **myprogram** and we want to make it SetUID root, we would do the following:

```
[master@server ~]$ chmod 4755 myprogram && sudo chown root myprogram
[master@server ~]$  ls -l myprogram
-rwsr-xr-x. 1 root  master 43160 Jul 13  2026   myprogram
```

Ensuring that a system has only the absolute minimum and necessary SetUID programs can be a good housekeeping measure. A typical Linux distribution can easily have several files and executables that are *unnecessarily* SetUID. Going from directory to directory to find SetUID programs can be tiresome and error-prone. So instead of doing that manually, you can use the **find** command, like so:

```
$  sudo  find /  -perm -4000 -ls
```

To search for files that are SetGID instead, type the following:

```
$  sudo  find / -type f  -perm -2000 -ls
```

To search for both SetUID and SetGID files with one single **find** command and view the octal mode for the file permissions using the **stat** command, type this:

```
$  sudo  find / -type f \( -perm -4000 -o -perm -2000 \) | \
 xargs stat -c '%A %a %n'
```

Unnecessary Processes

When looking through the system's boot or startup sequence, you may have noticed that a standard-issue Linux system starts with several (familiar and unfamiliar) processes running.

The underlying security issue always goes back to the question of risk: Is the risk of running an application worth the value it brings you? If the value a particular process brings is zero because you're not using it, then no amount of risk is worth it. Looking beyond security, there is the practical matter of stability and resource consumption. If a process brings zero value, even a benign process that does nothing but sit in an idle loop uses memory, processor time, and kernel resources. If a bug were to be found in that process, it could threaten the stability of your server. The bottom line is this: *if you don't need it, don't run it!*

If your system is running as a server, you should reduce the number of processes that are run. For example, if there is no reason for the server to connect to a printer, disable the print services. If there is no reason the server should accept or send e-mail, turn off the mail server component. If no services are run from **xinetd**, then **xinetd** should be turned off. No printer? Turn off Common UNIX Printing System (CUPS). Not a file server? Turn off Network File System (NFS) and Samba.

Fully thinned down, the server should be running the bare minimum it needs to provide the services required of it.

A Real-Life Example: Thinning Down a Server

Let's take a look at a real-life deployment of a Linux server handling web and e-mail access outside of a firewall and a Linux desktop/workstation behind a firewall with a trusted user. The two configurations represent extremes: tight configuration in a hostile environment (the Internet) and a loose configuration in a well-protected and trusted environment (a local area network, or LAN).

The Linux server runs the latest Fedora distro. With unnecessary processes thinned down, the server has 10 programs running, with 18 processes when no one is logged in. Of the 10 programs, only SSH, Apache, and Postfix are externally visible on the network. The rest handle basic management functions, such as logging (**rsyslog**) and scheduling (**cron**). Removing nonessential services (for example, Squid proxy server), the running program count can be reduced to seven (**init**/**systemd**, **rsyslog**, **cron**, **SSH**, **Postfix**, **Getty**, and **Apache**), with 13 processes running, five of which are Getty to support logins on serial ports and the keyboard.

By comparison, a Fedora system configured for workstation usage by a trusted user that has not been thinned down can have as many as 100 processes that handle everything from the X Window System, to printing, to basic system management services.

For workstation-style systems where the risk is mitigated (for example, where the workstation sits behind a firewall and the users are trusted), the benefits of having a lot of these applications running might well be worth the risk. Trusted users appreciate the ability to print easily and enjoy having access to a nice GUI, for example. For a server, however, the risk would be too great to have unnecessary programs running!

Picking the Right Runlevel

A Linux system with a GUI desktop environment provides a nice startup screen, a login menu, a mild learning curve, general familiarity, and an overall positive desktop experience. For a server, however, the trade-off is probably not worth it.

Most modern **systemd**-enabled Linux distros that are configured to boot and load the X Window (GUI) subsystem will boot to **graphical.target** target (also referred to as runlevel 5 in the SysV Init world). In such distros, changing the default boot target to **multi-user.target** (also referred to as runlevel 3 in the SysV init world) will turn off the GUI subsystem.

Modern **systemd**-enabled Linux distros use the **`systemctl`** utility, as well as a series of file system elements (soft links) to control and manage the system's default boot target (runlevel). Chapters 7 and 9 cover **systemd** in detail as well as show how to change the default boot target.

TIP You can see what runlevel you're in by using any of the following commands:

```
$ runlevel
```

On a **systemd**-enabled system, you can alternatively run this:

```
$ systemctl get-default
```

Nonhuman User Accounts

User accounts on a server do not always correspond to actual human users. Recall that every process running on a Linux system must have an owner. Running the **`ps auxww`** command on your system will show all of the process owners in the leftmost column of the output. On your desktop system, for example, you could be the only human user, but a look at the **/etc/passwd** files shows that there are several other accounts on the system.

For an application to drop its root privileges, it must be able to run as another user. This is where those extra users come into play: Each application that gives up its root privileges can be assigned another dedicated (and less privileged) user profile on the system. This other user typically owns all the application's files (including executable, libraries, configuration, and data) and the application processes. By having each application that drops privileges use its own user, the risk of a compromised application having access to other application configuration files is mitigated. In essence, an attacker is limited by what files the application can access, which, depending on the application, may be quite uninteresting.

Limited Resources

To better control the resources available to processes started by the shell, the **`ulimit`** facility can be used. System-wide defaults can be configured using the **/etc/security/limits.conf** file. **`ulimit`** options can be used to control such things as the number of files that may be open, how much memory they may use, how much CPU time they may use, how many processes they may open, and so on. The settings are read by the PAM (Pluggable Authentication Module) libraries when a user starts up. Incidentally, some servers that run/host mission-critical applications such as databases also use facilities like **`ulimit`** for performance-tuning purposes.

The key to choosing **`ulimit`** values is to consider the purpose of the system. For example, in the case of an application server, if the application is going to require a lot of processes to

run, then the system administrator needs to ensure that **ulimit** caps don't cripple the functionality of the system. Other types of single-purpose applications, such as a Domain Name System (DNS) server, should not need more than a small handful of processes.

Note a caveat here: PAM must have a chance to run to set the settings before the user does something. If the application starts as root and then drops permissions, PAM is not likely to run. From a practical point of view, this means that having individual per-user settings is not likely to do you a lot of good in most server environments. What will work are global settings that apply to both root and normal users. This detail turns out to be a good thing in the end; having root under control helps keep the system from spiraling away both from attacks and from broken applications.

TIP A Linux kernel feature known as *control groups* (cgroups) also provides the ability to manage and allocate various system resources such as CPU time, network bandwidth, memory, and so on. For more on cgroups, see Chapter 11.

The Fork Bomb

A popular prank that users like to play on one another on multiuser workstations is known as a "fork bomb." This is triggered by a program that simply creates so many processes that it overwhelms the system and brings it to a grinding halt. For the victim, this is merely annoying. For a production server, it can be fatal! Here's a simple shell-based fork bomb using the Bourne Again Shell (Bash):

```
[yyang@server ~]$   while true; do sh -c sh & done
```

If you don't have protections in place, *this script will eventually crash your server.*

The interesting thing about fork bombs is that not all of them are intentional. Broken applications, systems under denial-of-service (DoS) attacks, and sometimes just simple typographical errors while entering commands can lead to bad things happening. By using the limits described in this chapter, you can mitigate the risk of a fork bomb by restricting the maximum number of processes that a single user can invoke. While the fork bomb can still cause your system to become highly loaded, it will likely remain responsive enough to allow you to log in and deal with the situation, all the while (hopefully) maintaining the other services offered. It's not perfect, but it is a reasonable balance between dealing with the malicious and not being able to do anything at all.

The format of each line in the **/etc/security/limits.conf** file is as follows:

```
<domain>     <type>     <item>      <value>
```

Any line that begins with a pound sign (#) is a comment. The ***domain*** value holds the login name of a user or the name of a group; it can also be a wildcard (*). The ***type*** field refers to the type of limit, as in **soft** or **hard**.

The ***item*** field refers to what the limit applies to. The following is a sample of some items that an administrator might find useful:

Item	Description	Common Default Values
fsize	Maximum file size	Unlimited
nofile	Maximum number of open files	Unlimited
cpu	Maximum amount of time (in minutes) a CPU can be used	Unlimited
nproc	Maximum number of processes that a user can have	Unlimited
maxlogins	Maximum number of logins for a user	Unlimited

A reasonable tweak on a server is to restrict the number of processes for users. The same should be done for other settings too—after duly considering their effects. Remember, **ulimit** is not a cure-all for restricting or managing all classes of system resources. You have to use the proper tool for the job. So, for example, if you need to control total disk usage for a user, you should use disk quotas instead.

To mitigate problems that can be caused by users' ability to spawn too many processes that can quickly use up system resources (like in the *fork bomb* example), we can implement a **ulimit** setting for limiting the number of processes to 512 (for example) for each user. We can do this by creating an entry like the one shown here in the **/etc/security/limits.conf** file:

```
*           hard        nproc       512
```

If you log out and log in again, you can see the limits take effect by running the **ulimit** command with the **-a** option, as shown next, to see what the limits are. The highlighted **max user processes** entry in the following sample output shows the change.

```
[master@server ~]$  ulimit   -a
core file size          (blocks, -c) 0
...<OUTPUT TRUNCATED>...
cpu time               (seconds, -t) unlimited
max user processes              (-u) 128
virtual memory          (kbytes, -v) unlimited
```

Mitigating Risk

Once you know what the risks are, mitigating them becomes easier. You might find that the risks you see are sufficiently low, such that no additional securing is necessary. For example, a Microsoft Windows desktop system used by a trusted, well-experienced user is a low risk for running with administrator privileges. The risk that the user downloads and executes something that can cause damage to the system is low. This well-experienced user may find that being able to run some additional tools and having raw access to the system are well worth the risk of running with administrator privileges. Like any nontrivial risk, the list of caveats is long.

chroot

The **chroot()** system call (pronounced "cha-root") allows a process and all of its child processes to redefine what they perceive the root directory to be. For example, if you were to run **chroot("/www")** and start a shell, you could find that using the **cd** command would leave you at **/www**. The program would believe it is a root directory, but in reality, that would not be the case. This restriction applies to all aspects of the process's behavior: where it loads configuration files, shared libraries, data files, and so on. The restricted environment is also sometimes referred to as a "jail."

When the perceived root directory of the system is changed, a process has a restricted view of what is on the system. Access to other directories, libraries, and configuration files is not available. Because of this restriction, it is necessary for a target application to have all of the files necessary for it to work completely contained within the **chroot** environment. This includes any passwd files, libraries, binaries, and data files.

Most major applications have their own set of configuration files, libraries, and executables and thus the directions for making an application work in a **chroot** environment vary. However, the principle remains the same: make it all self-contained under a single directory with a faux root directory structure.

CAUTION A **chroot** environment will protect against accessing files outside of the directory, but it does not protect against system utilization, memory access, kernel access, and interprocess communication. This means that if there is a security vulnerability that someone can take advantage of by sending signals to another process, it will be possible to exploit it from within a **chroot** environment. In other words, **chroot** is not a perfect cure, but more a deterrent.

An Example chroot Environment

As an example, let's create a **chroot** environment for the Bash shell. We begin by creating the directory into which we want to put everything. Because this is just an example, we'll create a directory in **/tmp** called **myroot**:

```
[master@server ~]$   mkdir /tmp/myroot
[master@server ~]$   cd /tmp/myroot
```

Let's assume we need only two programs: **bash** and **ls**. Let's create the **bin** directory under **myroot** and copy the binaries over there:

```
[master@server myroot]$  mkdir bin
[master@server myroot]$ cp  $(type -P bash) $(type -P ls)  bin/
```

With the binaries there, we now need to check whether these binaries need any libraries. We use the **ldd** command to determine what (if any) libraries are used by these two programs.

CAUTION The following copy (**cp**) commands were based strictly on the output of the **ldd $(type -P bash)** and **ldd $(type -P ls)** commands on *our* sample system. You might need to modify the names and versions of the files that you are copying over to the **chroot** environment to match the exact filenames that are required on *your* system/platform.

We run **ldd** against **/bin/bash**, like so:

```
[master@server myroot]$  ldd  $(type -P bash)
    linux-vdso.so.1 (0x00007ffe68add000)
        libtinfo.so.6 => /lib64/libtinfo.so.6 (0x00007f5adaf70000)
        libdl.so.2 => /lib64/libdl.so.2 (0x00007f5adaf69000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f5adada0000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f5adb0e5000)
```

We also run **ldd** against `/bin/ls`, like so:

```
[master@server myroot]$  ldd  $(type -P ls)

linux-vdso.so.1 (0x00007ffd26bac000)
libselinux.so.1 => /lib64/libselinux.so.1 (0x00007f5d1f3c9000)
libcap.so.2 => /lib64/libcap.so.2 (0x00007f5d1f3c2000)
libc.so.6 => /lib64/libc.so.6 (0x00007f5d1f1f9000)
libpcre2-8.so.0 => /lib64/libpcre2-8.so.0 (0x00007f5d1f16b000)
libdl.so.2 => /lib64/libdl.so.2 (0x00007f5d1f164000)
/lib64/ld-linux-x86-64.so.2 (0x00007f5d1f434000)
libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f5d1f142000)
```

Now that we know what libraries need to be in place, we create the **lib64** directory and copy the 64-bit libraries over (because we are running a 64-bit operating system).

First, we create the **/tmp/myroot/lib64** directory:

```
[master@server myroot]$  mkdir /tmp/myroot/lib64/
```

Next, we'll copy over the shared libraries that **/bin/bash** needs:

```
[master@server myroot]$  cp /lib64/libtinfo.so.6   lib64/
[master@server myroot]$  cp /lib64/libdl.so.2   lib64/
[master@server myroot]$  cp /lib64/libc.so.6   lib64/
[master@server myroot]$  cp /lib64/ld-linux-x86-64.so.2   lib64/
```

And for **/bin/ls**, we need to run the following commands to get the *other* needed library files that we don't *already* have:

```
[master@server myroot]$  cp  /lib64/libselinux.so.1   lib64/
[master@server myroot]$  cp  /lib64/libcap.so.2  lib64/
[master@server myroot]$  cp /lib64/libpcre2-8.so.0    lib64/
[master@server myroot]$  cp /lib64/libpthread.so.0   lib64/
```

Most Linux distros conveniently include a powerful little program called **chroot** that can invoke the **chroot()** system call for us. The program takes two parameters: the directory that we want to make the root directory and the command that we want to run in the **chroot** environment. We want to use **/tmp/myroot** as the directory and start **/bin/bash**, so we run the following:

```
[master@server myroot]$  sudo chroot /tmp/myroot   /bin/bash
```

Because there is no **/etc/profile** or **/etc/bashrc** to change/customize our prompt, the prompt will change to something like **bash-<VERSION_NUMBER>#**. Now try running the **/bin/ls** command in your **chroot** environment:

```
bash-*#  /bin/ls
bin  lib64
```

Next, try a **pwd** to view the current working directory:

```
bash-*#  pwd
/
```

NOTE You may be wondering where the **pwd** command that we ran in the previous command hails from. We didn't need to explicitly copy over the **pwd** command, because **pwd** is one of the many Bash built-in commands. It comes with the Bash program that we already copied over!

Since we don't have an **/etc/passwd** or **/etc/group** file in the **chroot**-ed environment (to help map numeric user IDs to usernames), an **ls -l** command will show the raw user ID (UID) values for each file. Here's an example:

```
bash-*#  cd lib64/
bash-*#  /bin/ls -l
-rwxr-xr-x. 1 1000  1000   228072  Mar  7 12:44 ld-linux-x86-64.so.2
-rwxr-xr-x. 1 1000  1000   2786376 Mar  7 12:44 libc.so.6
...<OUTPUT TRUNCATED>...
```

With limited commands/executables in our sample **chroot** environment, the environment isn't terribly useful for practical work, which is what makes it great from a security perspective; we allow only the absolute minimum files necessary for an application to work, thus minimizing our exposure in the event the application gets compromised. Keep in mind that not all **chroot** environments need to have a shell and an **ls** command installed—for example, *if* the Berkeley Internet Name Domain (BIND) DNS server software needs only its own executable, libraries, and zone files installed, then that's all you need!

To quit the **chroot** environment, use the **exit** command:

```
bash-*# exit
```

NOTE A popular implementation of OS-level virtualization (or software/application containers) makes extensive use of the basic principles of **chroot**-ing as well as resource isolation and partitioning facilities of Linux (cgroups). LinuX Containers (LXC), Docker, CoreOS, Rocket, FreeBSD Jail (non-Linux), and so on are popular implementations of these concepts. Chapter 31 covers this topic in more detail.

SELinux

Traditional Linux security is based on a *Discretionary Access Control (DAC)* model. The DAC model allows the owner of a resource (objects) to control which users or groups (subjects) can access the resource. It is called "discretionary" because the access control is based on the discretion of the owner.

Another type of security model is the *Mandatory Access Control (MAC)* model. Unlike the DAC model, the MAC model uses predefined policies to control user and process interactions. The MAC model restricts the level of control that users have over the objects that they create. SELinux is an implementation of the MAC model in the Linux kernel.

The U.S. government's National Security Agency (NSA) has taken an increasingly public role in information security, especially due to the growing concern over information security attacks that could pose a serious threat to the world's ability to function. With Linux being a major component of enterprise computing, the NSA set out to create a set of patches to increase the security of Linux. The patches have all been released under the GNU Public License (GPL) with full source code and are thus subject to the scrutiny of the world—an important consideration given Linux's worldwide presence and developer community. The patches are collectively known as "SELinux," short for "Security-Enhanced Linux." The patches have been integrated into the Linux kernel using the Linux Security Modules (LSM) framework. This integration has made the patches and improvements far-reaching and an overall benefit to the Linux community.

SELinux makes use of the concepts of subjects (users, applications, processes, and so on), objects (files and sockets), labels (metadata applied to objects), and policies (which describe the matrix of access permissions for subjects and objects). Given the extreme granularity of objects, it is possible to express rich and complex rules that dictate the security model and behavior of a Linux system. Because SELinux uses labels, it requires a file system that supports extended attributes.

The full gist of SELinux is well beyond the scope of a single section in this book. To learn more about SELinux, visit the SELinux Fedora Wiki project page at http://fedoraproject.org/wiki/SELinux.

TIP As useful as SELinux is, you may find that it is the cause of some hard-to-debug issues that prevent some applications or subsystems from working properly. In such situations, if you need to quickly eliminate SELinux as being the issue, you can temporarily disable it by running

```
[master@fedora-server ~]$ sudo  setenforce 0
```

When you are ready to re-enable it, run

```
[master@fedora-server ~]$ sudo  setenforce 1
```

AppArmor

AppArmor is another implementation of the MAC security model on Linux-based systems. It is SUSE's alternative to SELinux (which is used mainly in Red Hat–derived distros such as Fedora, CentOS, and RHEL). AppArmor's backers tout it as being easier to manage and configure than SELinux. AppArmor's implementation of the MAC model focuses more on protecting individual applications—hence the name Application Armor—instead of attempting a blanket security model that applies to the entire system, as in SELinux. AppArmor's security goal is to protect systems from attackers exploiting vulnerabilities in specific applications that are running on the system. AppArmor is file system independent. It is integrated into and used mostly in openSUSE, SUSE Linux Enterprise (SLE), as well as some Debian-based distros. And, of course, it can also be installed and used in other Linux distributions.

Monitoring Your System

As you become familiar with Linux, your servers, and their day-to-day operation, you'll find that you start getting a "feel" for what is normal. This might sound peculiar, but in much the same way you learn to feel when your car isn't running quite right, you'll know when your server is not *acting* quite the same.

Part of your getting a feel for the system requires basic system monitoring. For local system behavior, you need to trust your underlying system as not having been compromised in any way. If your server does get compromised and a "root kit" that bypasses monitoring systems is installed, it can be difficult to see what is happening. For this reason, a mix of on-host monitoring and remote-host–based monitoring is a good idea.

Logging

By default, most of your log files will be stored in the **/var/log** directory, with the **`logrotate`** program automatically rotating (archiving) the logs on a regular basis. Although it is handy to be able to log to the local disk, it is often a better idea to also have logs sent to a dedicated log server. With remote logging enabled, you can be certain that any logs sent to the log server before an attack are most likely guaranteed not to be tampered with.

Because of the volume of log data that can be generated, you might find it prudent to learn some basic scripting skills so that you can easily parse through the log data and automatically highlight and e-mail or notify the administrator of anything that is peculiar or should warrant suspicion. This allows the administrator to track both normal and erroneous activity without having to read through a significant number of log messages every day. We discussed some log-filtering techniques and utilities (**`journalctl`**) in Chapter 9 that can help with this.

Using ps and netstat

You should periodically review the output of the **`ps auxww`** command. Future deviations from any established baseline output should catch your attention. As part of monitoring, you may find it useful to periodically list what processes are running and make sure that any processes you don't expect are there for a *reason*. Be especially suspicious of any packet-capture programs, such as **`tcpdump`**, that you did not start yourself!

The same can be said about the output of the **`netstat -an`** command (admittedly, **`netstat`**'s focus is more from a network security standpoint). Once you have a sense of what represents normal traffic and normally open ports, any deviations from that output should trigger interest into why the deviation is there. It might help answer questions such as these: Did someone change the configuration of the server? Did the application do something that was unexpected? Is there threatening activity on the server?

Between **`ps`** and **`netstat`**, you should have a fair handle on the goings-on with your network and process list.

Watch That Space (Using df)

The **df** command shows the available space on each of the disk partitions that is mounted. Running **df** on a regular basis to see the rate at which disk space gets used is a good way to look for any questionable activity. A sudden change in disk utilization should spark your curiosity. For example, a sudden increase could be because users are using their home directories to store vast quantities of MP3 files, movies, and so on. Legal issues aside, there are also other pressing concerns and repercussions for such unofficial use, such as backups and DoS issues.

The backups might fail because the backup medium ran out of space storing someone's music files instead of the key files necessary for the business. From a security perspective, if the sizes of the web or FTP directories grow significantly without reason, it may signal trouble looming with unauthorized use of your server. A server whose disk becomes full unexpectedly is also a potential source of a local (and/or remote) DoS attack. A full disk might prevent legitimate users from storing new data or manipulating existing data on the server. The server may also have to be temporarily taken offline to rectify the situation, thereby denying access to other services that the server should be providing.

Automated Monitoring

Most of the popular automated system-monitoring solutions specialize in monitoring network-based services and daemons. However, most of these also have extensive local resource-monitoring capabilities to monitor such things as disk usage, CPU usage, process counts, changes in file system objects, and so on. Some examples include sysinfo, Nagios, Tripwire, Munin, sysstat utilities (sar, iostat, and sadf), Beats (via Elasticsearch, Logstash, and Kibana aka ELK), Icinga, and so on.

Staying in the Loop (Mailing Lists)

As part of managing your system's security, you should be subscribed to key security mailing lists, such as BugTraq (www.securityfocus.com). BugTraq is a moderated mailing list that generates only a small handful of e-mails a day, most of which may not pertain to the software you are running. However, this is where critical issues are likely to show up first and be dealt with in real time.

In addition to BugTraq, any security lists for software for which you are responsible are musts. Also look for announcement lists for the software you use. Most Linux distributions also maintain announcement lists for security issues that pertain to their specific distros. Major software vendors also maintain their own lists. Although this may seem like a lot of e-mail, consider that most of the lists that are announcement-based are extremely low volume. In general, you likely won't need to deal with significantly more e-mail than you already do!

Summary

In this chapter you learned about securing your Linux system and mitigating risk, and you learned what to look for when making decisions about how to balance function with the need to be secure. We touched on local security concepts and techniques at a very high level. Specifically, we discussed SetUID programs, mitigating risk through the use of **`chroot`** environments, popular MAC security models (SELinux and AppArmor), and things that should be monitored as part of daily system housekeeping.

In the end, you will find that maintaining a reasonably secure environment is akin to maintaining good hygiene. Keep your server clean of unnecessary applications, make sure the environment for each application is minimized so as to limit exposure, and patch your software as security issues are brought to light. Try to keep as up to date as possible with security-related news for software that you run. With these basic commonsense practices, you'll find that your servers will be quite reliable and secure.

This page intentionally left blank

CHAPTER

16 Network Security

In Chapter 15, you learned that exploit vectors are of two types: those in which the vulnerability is exploitable locally and those in which the vulnerability is exploitable over a network. The latter case is covered in this chapter.

Network security addresses the problem of attackers sending malicious network traffic to your system with the intent of either making your system unavailable (denial-of-service, or DoS, attack) or exploiting weaknesses in your system to gain access or control of the system or other related systems. Network security is not a substitute for the good local security practices discussed in the previous chapter. Both local and network best security practices are necessary to keep things working in the expected manner.

This chapter covers four aspects of network security: tracking services, monitoring network services, handling attacks, and tools for testing. The following sections should be used as accompaniment with the information in Chapters 14 and 15.

TCP/IP and Network Security

The following discussion assumes you have experience configuring a system for use on a TCP/IP network. Because the focus here is on network security and not an introduction to networking, this section discusses only those parts of TCP/IP affecting your system's security. If you're curious about TCP/IP's internal workings, read Chapter 12.

The Importance of Port Numbers

Every host on an IP-based network has at least one IP address. In addition, every Linux-based host has many individual processes running. Each process has the potential to be a network client, a network server, or both. With potentially more than one process being able to act as a server on a single system, using an IP address alone to uniquely identify a network connection is not enough.

To solve this problem, TCP/IP adds a component to uniquely identify a TCP or UDP connection, called a *port*. Every connection from one host to another has a *source port* and a *destination port*. Each port is labeled with an integer between 0 and 65535.

To identify every unique connection possible between two hosts, the operating system keeps track of four pieces of information: the source IP address, the destination IP address, the source port number, and the destination port number. The combination of these four values is guaranteed to be unique for all host-to-host connections. (Actually, the operating system tracks a myriad of connection information, but only these four elements are needed for uniquely identifying a connection.)

The host initiating a connection specifies the destination IP address and port number. Obviously, the source IP address is already known. But the source port number, the value that will make the connection unique, is assigned by the source operating system. It searches through its list of already open connections and assigns the next available port number.

By convention, this number is always greater than 1024 (port numbers from 0 to 1023 are reserved for system uses and well-known services). Technically, the source host can also select its source port number. To do this, however, another process cannot have already taken that port. Generally, most applications let the operating system pick the source port number for them.

Given this arrangement, we can see how source *host A* can open multiple connections to a single service on destination *host B*. Host B's IP address and port number will always be constant, but host A's port number will be different for every connection. The combination of source and destination IPs and port numbers is, therefore, unique, and both systems can have multiple independent data streams (connections) between each other.

For a typical server application to offer services, it would usually run programs that listen on specific port numbers. Many of these port numbers are used for well-known services and are collectively referred to as *well-known ports,* because the port number associated with a known service is an approved standard. For example, port 80 is the well-known service port for HTTP.

In the upcoming "Using the netstat Command" section, we'll look at the **`netstat`** command as an important part of your network security arsenal. When you have a firm understanding of what port numbers represent, you'll be able to identify and interpret the network security statistics provided by tools like the **`netstat`** (and similar) commands.

Tracking Services

The services provided by a server are what make it a server. The ability to provide the service is accomplished by processes that bind to network ports and listen to the requests coming in. For example, a web server might start a process that binds to port 80 and listens for requests to download the pages of a site it hosts. Unless a process exists to listen on a specific port, Linux will simply ignore packets sent to that port.

This section discusses the usage of the **`netstat`** command, a tool for tracking and debugging network connections (among other things) in your system.

Using the netstat Command

To track what ports are open and what ports have processes listening to them, we use the **netstat** command. Here's an example:

```
# netstat  -natu
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address    Foreign Address            State
tcp        0      0 0.0.0.0:22      0.0.0.0:*                  LISTEN
tcp        0      0 192.168.1.4:22  192.168.1.135:52248        ESTABLISHED
tcp        0      0 :::111          :::*                       LISTEN
udp        0      0 0.0.0.0:56781   0.0.0.0:*
```

By default (with no parameters), **netstat** will provide all established connections for both network and domain sockets. That means we'll see not only the connections that are actually working over the network, but also the interprocess communications (which, from a security monitoring standpoint, might not be immediately useful). So in the command just illustrated, we have asked **netstat** to show us all ports (**-a**), regardless of whether they are listening or actually connected, for TCP (**-t**) and UDP (**-u**). We have told **netstat** not to spend any time resolving IP addresses to hostnames (**-n**).

In the **netstat** output, each line represents either a TCP or UDP network port, as indicated by the first column of the output. The Recv-Q (receive queue) column lists the number of bytes received by the kernel but not read by the process. Next, the Send-Q (send queue) column tells us the number of bytes sent to the other side of the connection but not acknowledged.

The fourth, fifth, and sixth columns are the most interesting in terms of system security. The Local Address column tells us our server's IP address and port number. Remember that our server recognizes itself as 127.0.0.1 and 0.0.0.0, as well as its normal IP address. In the case of multiple interfaces, each port being listened to will usually show up on all interfaces and, thus, as separate IP addresses. The port number is separated from the IP address by a colon (:). In the output, the Ethernet device has the IP address 192.168.1.4.

The fifth column, Foreign Address, identifies the other side of the connection. In the case of a port that is being listened to for new connections, the default value will be 0.0.0.0:*. This IP address initially means nothing, since we're still waiting for a remote host to connect to us!

The sixth column tells us the state of the connection. The man page for **netstat** lists all of the states, but the two you'll see most often are LISTEN and ESTABLISHED. The LISTEN state means that a process on your server is listening on the port number and ready to accept new connections. The ESTABLISHED state means just that—a connection is established between a client and server.

Security Implications of netstat's Output

By listing all of the available connections, you can get a snapshot of what the system is doing. You should be able to explain and account for *all* ports listed. If your system is listening to a port that you cannot explain, this should raise suspicions.

Just in case you haven't yet memorized all the well-known services and their associated port numbers (all 25 zillion of them!), you can look up the matching information you need in the **/etc/services** file. However, some services (most notably those that use the portmapper) don't have set port numbers but are valid services. To see which process is associated with a port, use the **`-p`** option with **`netstat`**. Be on the lookout for odd or unusual processes using the network sockets. For example, if the Bourne Again Shell (Bash) is listening to a network port, you can be fairly certain that something odd is going on!

Finally, remember that you are mostly interested in the destination port of a connection; this tells you which service is being connected to and whether it is legitimate. The source address and source port are, of course, important, too—especially if somebody or something has opened up an unauthorized back door into your system. Unfortunately, **`netstat`** doesn't explicitly tell you who originated a connection, but you can usually figure it out if you give it a little thought. Of course, becoming familiar with the applications that you do run and their use of network ports is the best way to determine *who* originated a connection to *where*. In general, you'll find that the rule of thumb is that the side whose port number is greater than 1024 is the side that originated the connection. Obviously, this general rule doesn't apply to absolutely all services. Some oddball services run on ports higher than 1024, such as the X Window System (port 6000).

Binding to an Interface

A common approach to improving the security of a service running on your server is to make it such that it binds only to a specific network interface. By default, applications will bind to all interfaces (seen as 0.0.0.0 in the **`netstat`** output). This will allow a connection to that service from any interface—so long as the connection makes it past any Netfilter rules (built-in Linux Kernel firewall stack) you may have configured. However, if you need a service to be available only on a particular interface, you should configure that service to bind to the specific interface.

For example, let's assume that there are three interfaces on your server:

- eno1, with the IP address 192.168.1.4
- eno2, with the IP address 172.16.1.1
- lo, with the IP address 127.0.0.1

Let's also assume that your server does not have IP forwarding (**/proc/sys/net/ipv4/ip_forward**) enabled. In other words, machines on the 192.168.1.0/24 (eno1) side cannot communicate with machines on the 172.16/16 side. The 172.16/16 (eno2) network represents the "safe" or "inside" network, and, of course, 127.0.0.1 (lo or loopback) represents the host itself.

If the application binds itself to 172.16.1.1, then only those hosts on the 172.16/16 network will be able to reach the application and connect to it. If you do not trust the hosts on the 192.168.1/24 side (for example, because it is designated as a demilitarized zone, or DMZ), this is a safe way to provide services to one segment without exposing yourself to another. For even less exposure, you can bind an application to 127.0.0.1. By doing so, you can almost guarantee

that connections will have to originate from the server itself to communicate with the service. For example, if you need to run the MySQL database for a web-based application and the application runs on the server, then configuring MySQL to accept only connections from 127.0.0.1 means that any risk associated with remotely connecting to and exploiting the MySQL service is significantly mitigated. The attacker would have to compromise your web-based application and somehow make it query the database on the attacker's behalf (perhaps via a SQL injection attack) in order to circumvent this setup.

SSH Tunneling Tricks

If you need to temporarily provide a service to a group of users across the Internet, binding the service to the loopback address (localhost) and then forcing the group to use SSH tunnels is a great way to provide authenticated and encrypted access to the service.

For example, if you have a Post Office Protocol 3 (POP3) service running on your server, you can bind the service to the localhost address. This, of course, means nobody will be able to connect to the POP3 server via a regular interface/address. But if you run an SSH server on the system, authenticated users can connect via SSH and set up a port-forwarding tunnel for their remote POP3 e-mail client. Here's a sample command to do this from the remote SSH client:

```
ssh -l <username> -L 1110:127.0.0.1:110  <REMOTE_SSH_and_POP3_SERVER>
```

After successfully establishing an SSH connection to the server using the previous sample command, you can configure any POP3 e-mail client to connect to the POP3 server at the IP address 127.0.0.1 via port 1110 (127.0.0.1:1110).

Shutting Down Services

Striking a reasonable balance between ease of installation/manageability of Linux servers and providing a secure out-of-box experience in Linux distros can be a delicate process. One of the side effects of this can been seen in some distributions that aim to oversimplify things for end users by adopting unsafe default settings, so as to provide convenience for the end user. The task of seeking out these unsafe default settings is thus left to you as the system administrator.

While evaluating which services should stay or go, answer the following questions:

- *Do we need the service?* The answer to this question is important. In most situations, you should be able to disable a great number of services that start up by default.
- *If we do need the service, is the default setting secure?* This question can also help you eliminate some services—if they aren't secure and they can't be made secure, then chances are they should be removed. For example, if remote login is a requirement and Telnet is the service enabled to provide that function, then an alternative such as SSH should be used instead, due to Telnet's default inability to encrypt login information over a network.

- *Are the developers of the software providing the service still actively maintaining the software with security patches?* All software needs updates from time to time. This is partly because as features get added, new security/bugs problems creep in. So be sure to remember to track the server software's development and get updates as necessary.

Shutting Down xinetd and inetd Services

To shut down a service that is started via the **xinetd** program, edit the service's configuration file under the **/etc/xinetd.d/** directory and set the value of the **disable** directive to **Yes**.

For traditional System V–based services, you can also use the **chkconfig** command to disable the service managed by **xinetd**. For example, to disable the echo service, type the following:

```
[master@server ~]$  sudo chkconfig echo off
```

On modern Linux distributions running **systemd**, you can alternatively disable a service using the **systemctl**. For example, to disable the **xinetd** service, use the following:

```
[master@fedora-server ~]$ sudo systemctl disable xinetd.service
```

On legacy Debian-based systems such as Ubuntu, you can use the **sysv-rc-conf** command (install it with the **apt-get** command if you don't have it installed) to achieve the same effect. For example, to disable the echo service in Ubuntu, you could run the following:

```
master@ubuntu-server:~$  sudo sysv-rc-conf  echo  off
```

Shutting Down Non-xinetd Services

If a service is not managed by **xinetd**, then a separate process or script that is started at boot time is running it. If the service in question was installed by your distribution, and your distribution offers a nice tool for disabling a service, you may find that to be the easiest approach.

On modern Linux distributions running **systemd**, you can stop a service using the **systemctl** command. For example, to stop the **rpcbind** service, type the following:

```
[master@fedora-server ~]$ sudo systemctl  stop rpcbind.service
```

Similarly, to permanently disable the **rpcbind** service unit from starting up during the next system boot, using the **systemctl** utility, type this:

```
[master@fedora-server ~]$  sudo systemctl disable rpcbind.service
```

Alternatively, on legacy (non-**systemd**) Linux distros, the **chkconfig** program provides an easy way to enable and disable individual services. For example, to disable the **rpcbind** service from starting in runlevels 3 and 5 on such systems, simply run the following:

```
[master@server ~]$   sudo chkconfig --level 35 rpcbind  off
```

Note that using **chkconfig** doesn't actually turn an *already running* service on or off; instead, it defines what will happen at the next startup time. To stop the running process, use the control script in the **/etc/init.d/** directory or the **service** command. In the case of **rpcbind**, we would stop it with the following:

```
[master@server ~]$ sudo service rpcbind stop
```

Monitoring Your System

Several free and open source commercial-grade applications exist that perform monitoring and are well worth checking out. Here, we'll take a look at a variety of excellent tools that help with system monitoring. Some of these tools already come installed with your Linux distributions; others don't. All are free and easily acquired.

Making the Best Use of syslog

In Chapter 9, we explored **rsyslogd**, the system logger, as well as the **systemd-journald** service (**journald**), both of which help manage and collect log messages from various programs. By now, you've probably seen the types of log messages you get with **rsyslogd** and have played with the **journalctl** utility as well. These include security-related messages, such as who has logged into the system, when they logged in, and so forth.

As you can imagine, it's possible to analyze these logs to build a time-lapse image of the utilization of your system services. This data can also point out questionable activity. For example, why was the host crackerboy.nothing-better-to-do.net sending so many web requests in such a short period of time? Has he found a hole (vulnerability) in the system?

Log Parsing

Doing periodic checks on the system's log files is an important part of maintaining a good security posture. Unfortunately, scrolling through an entire day's worth of logs is a time-consuming task that might reveal few meaningful events. To ease the drudgery, pick up a text on a scripting language (such as Python) and write small scripts to parse out the logs. A well-designed script should ignore what it recognizes as normal behavior and show everything else. This can reduce thousands of log entries for a day's worth of activities down to a manageable few dozen. This is an effective way to detect attempted break-ins and possible security gaps.

Hopefully, it'll become entertaining to watch the script kiddies trying and failing to break down your walls. Several canned solutions exist that can also help make parsing through log files easier. Examples of such programs that you might want to try out are **journalctl**, **logwatch**, **gnome-system-log**, **ksystemlog**, ELK (www.elastic.co), and Splunk.

Storing Log Entries

Unfortunately, log parsing may not be enough. If someone breaks into your system, it's likely that your log files will be promptly erased—which means all those wonderful scripts won't be able to tell you a thing. To get around this, consider dedicating a single host on your network to storing log entries. Configure your local logging daemon to send all of its messages to a separate/central loghost, and configure the central host appropriately to accept logs from

trusted or known hosts. In most instances, this should be enough to gather, in a centralized place, the evidence of any bad things happening.

If you're *really* feeling paranoid, consider attaching another Linux host to the loghost using a serial port and using a terminal emulation package, such as **minicom**, in log mode and then feeding all the logs to the serially attached machine. Using a serial connection between the hosts helps ensure that one of the hosts does not need network connectivity. The logging software on the loghost can be configured to send all messages to **/dev/ttyS0** if you're using COM1, or to **/dev/ttyS1** if you're using COM2. And, of course, do *not* connect the *other* system to the network! This way, in the event the loghost also gets attacked, the log files won't be destroyed. The log files will be safely residing on the serially attached system, which is impossible to log into without physical access.

For an even (more ridiculous) higher degree of ensuring the sanctity of logs, you can connect a printer to another system and have the terminal emulation package echo everything it receives on the serial port to the printer. Thus, if the serial host system fails or is damaged in some way by an attack, you'll have a hard copy of the logs!

Monitoring Bandwidth with MRTG

Monitoring the amount of bandwidth being used on your servers produces some useful information. A common use for this is to justify the need for hardware upgrades, by being able to consistently demonstrate high system utilization levels. Your data can be easily turned into a graph, too—and everyone knows how much upper management folks like graphs and pretty pictures! Another useful benefit of monitoring bandwidth is to identify bottlenecks in the system, thus helping you balance the system load. But relative to the topic of this chapter, a useful aspect of graphing your bandwidth is to identify when things go wrong.

Once you've installed a package such as MRTG (Multi-Router Traffic Grapher, available at http://oss.oetiker.ch/mrtg/) to monitor bandwidth, you will quickly be able to establish a criterion for what "normal" looks like on your site. Investigate any inexplicable and substantial drop or increase in utilization, as it may indicate a failure or a type of attack. Other things to do are to check your logs, check the modification timestamps on configuration files to ensure that the modification times correspond to legitimate changes, look for configuration files with odd or unusual entries, and so on.

Handling Attacks

Part of security includes planning for the worst case: What happens if/when a break-in succeeds? At that point, knowing the details around the *how* and *when* is important, but possibly even more important is dealing with the aftermath of the event. Servers are doing things they shouldn't, information is leaking that shouldn't leak, or other mayhem is discovered by you or your team, and stakeholders are asking why you're spreading mayhem and not doing your job. Your pretty graphs won't be of much use to you in these situations!

Just as a facilities director plans for fires and a backup administrator plans for backing up and restoring data when needed, an IT security officer needs to plan for how to handle an attack. This section covers key points to consider with respect to Linux. For an excellent overview on handling attacks, visit the CERT web site at www.cert.org.

Trust Nothing (and No One)

The first thing you should do in the event of an attack is to *fire* everyone in the IT department. Absolutely *no* one is to be trusted. Everyone is guilty until proven innocent. *Just kidding!*

But, seriously, if an attacker has successfully broken into your systems, there is nothing that your servers can tell you about the situation that is completely trustworthy. Root kits (tool kits that attackers use to invade systems and then cover their tracks) can make detection difficult. With binaries replaced, you may find that there is nothing you can do to the server itself that helps. In other words, every server that has been successfully hacked may need to be completely rebuilt with a fresh installation. Before doing the reinstall, you should make an effort to look back at how far the attacker went so as to determine the point in the backup cycle when the data is certain to be trustworthy. Any data backed up after that should be closely examined to ensure that compromised data does not make it back into the system.

Change Your Passwords

If the attacker has gotten your root password or may have taken a copy of the password file (or equivalent), it is crucial that all of your passwords be changed. This is an incredible hassle; however, it is necessary to make sure that the attacker doesn't waltz back into your rebuilt server using the password without any resistance.

NOTE It is also a good idea to change the root password(s) as well as any other shared privileged account credentials following any staff changes. It may seem like everyone is leaving on good terms; however, later finding out that someone on your team had issues with the company can spell trouble.

Pull the Plug

Once you're ready to start cleaning up, you will need to stop any remote access to the system. You may find it necessary to stop all network traffic to the server until it is completely rebuilt with the latest patches before reconnecting it to the network.

This can be done by simply pulling the plug on whatever connects the box to the network. Putting a server back onto the network when it is still getting patches is an almost certain way to find yourself dealing with another attack.

Network Security Tools

Lots of tools exist to help monitor your systems, including Nagios (www.nagios.org), Icinga (www.icinga.org) and, of course, the various tools already mentioned in this chapter. But what do you use to poke at your system for basic sanity checks?

In this section, we review a few tools that you can use for testing your system. Note that no one single tool is enough, and no combination of tools is perfect—there is no secret "Hackers Testing Tool Kit" that security professionals use. The key to the effectiveness of most tools is how you use them and how you interpret the data gathered by the tools.

Some of the tools discussed here were originally created to aid in basic diagnostics and system management and later on evolved to also become useful as security tools. What makes these tools work well for Linux from a security perspective is that they offer deeper insight into what your system is doing. That extra insight often proves to be incredibly helpful.

nmap

The **nmap** program can be used to scan a host or a group of hosts to look for open TCP and UDP ports. **nmap** can go beyond scanning and can actually attempt to connect to the remote listening applications or ports so that it can better identify the remote application. This is a powerful and simple way for an administrator to take a look at what the system exposes to the network and is frequently used by both attackers and administrators to get a sense of what is possible against a host.

What makes **nmap** powerful is its ability to apply multiple scanning techniques. This is especially useful because each scanning technique has its pros and cons with respect to how well it traverses firewalls and the level of stealth desired.

Snort

An intrusion detection system (IDS) provides a way to monitor a point in the network surreptitiously and report on questionable activity based on packet traces. The Snort program (www.snort.org) is an open source IDS and intrusion prevention system (IPS) that provides extensive rule sets that are frequently updated with new attack vectors. Any questionable activity can be sent to a logging host, and several open source log-processing tools are available to help make sense of the information gathered (for example, the Basic Analysis and Security Engine, or BASE).

Running Snort on a Linux system that is located at a key entry/exit point in your network is a great way to track the activity without your having to set up a proxy for each protocol that you want to support.

Nessus and OpenVAS

The Nessus and OpenVAS applications take the idea behind **nmap** and extends it with deep application-level probes and a rich reporting infrastructure. Nessus is owned and managed by a commercial company, Tenable Network Security (www.tenable.com). The OpenVAS project (www.openvas.org) is a free and open source alternative to Nessus.

Running Nessus or OpenVAS against a server is a quick way to perform a sanity check on the server's exposure. Your key to understanding these systems is in understanding their output. The report will log numerous comments, from an informational level all the way up to a high level. Depending on how your application is written and what other services you offer on your Linux system, they may log false positives or seemingly scary informational notes. Take the time to read through each one of them and understand what the output is, as not all of the messages necessarily reflect your situation. For example, if the scanner detects that your system is at risk due to a hole in Oracle 25c but your server does not even run Oracle, more than likely, you have hit upon a false positive!

Wireshark/tcpdump

You learned about Wireshark and **tcpdump** in Chapter 12, where we used them to study the ins and outs of TCP/IP. Although those chapters used these tools only for troubleshooting, they are just as valuable for performing network security functions.

Raw network traces are the food devoured by all the tools listed in the preceding sections to gain insight into what your server is doing. However, these tools don't have quite the insight that you do into what your server is *supposed* to do. Thus, you'll find it useful to be able to take network traces yourself and read through them to look for any questionable activity. You may be surprised by what you see!

For example, if you are looking at a possible break-in, you may want to start a raw network trace from another Linux system that can see all of the network traffic of your questionable host. By capturing all the traffic over a 24-hour period, you can go back and start applying filters to look for anything that shouldn't be there. Extending the example, if the server is supposed to handle only web operations and SSH, you can take the packet trace with Domain Name System (DNS) resolution turned off and then apply the filter "not port 80 and not port 22 and not icmp and not arp." Any packets that show up in the output are suspect.

Summary

Using the information presented in this chapter, you should have the basic high-level knowledge you need to make an informed decision about the state of health of your server and decide what, if any, action is necessary to secure it.

It is important to take the time to know what constitutes normal or baseline behavior for the systems/services that you manage. Once you know what normal behavior is, unusual behavior will stick out like a sore thumb. For example, if you know that the Telnet service should not be running (or even installed) on your system normally, then seeing a system log entry for access via Telnet would mean that something is terribly wrong!

Security as a field is constantly evolving and requires keeping a watchful/careful eye toward new developments. Be sure to subscribe to the relevant mailing lists, keep an eye on relevant web sites, educate yourself with additional reading materials/books, and, most important, always apply common sense.

This page intentionally left blank

PART

IV

Internet Services

CHAPTER 17
Domain Name System (DNS)

CHAPTER 18
File Transfer Protocol (FTP)

CHAPTER 19
Apache Web Server

CHAPTER 20
Simple Mail Transfer Protocol (SMTP)

CHAPTER 21
Post Office Protocol and Internet Mail Access Protocol (POP and IMAP)

CHAPTER 22
Voice over Internet Protocol (VoIP)

CHAPTER 23
Secure Shell (SSH)

This page intentionally left blank

CHAPTER

17 Domain Name System (DNS)

The *Domain Name System (DNS)* provides a way to map or translate an unfriendly numerical IP address into a human-friendly format. Although this translation isn't mandatory, it does make the network much more useful and easier to work with for humans.

At inception of the Internet in the 1970s, IP-address-to-hostname mapping was done through the maintenance of a **hosts.txt** file that was distributed via FTP to all the machines on the Internet. As the number of hosts grew, it was soon clear that a single person maintaining a single file of all of those hosts was not a scalable way of managing the association of IP addresses to hostnames. To solve this problem, a distributed system was devised in which each site would maintain information about its own hosts. One host at each site would be considered authoritative, and that single host address would be kept in a master table that could be queried by all other sites. This is the essence of DNS.

If the information in DNS wasn't decentralized, as it now is, one other choice would be to have a central site maintaining a master list of all hosts (numbering in the tens of millions) and having to update those hostnames tens of thousands of times a day—an overwhelming alternative! Even more important to consider are the needs of each site. If you're stunned by the prospect of having to manage this for every host on the Internet, then you're getting the picture.

We will discuss DNS in depth, so you'll have what you need to configure and deploy your own DNS servers for whatever your needs might be.

NOTE In this chapter, you will see the terms "DNS server" and "name server" used interchangeably. Technically, "name server" is a little ambiguous because it can apply to any number of naming schemes that resolve a name to a number, and vice versa. In the context of this chapter, however, "name server" will always mean a DNS server unless otherwise stated.

The Hosts File

Not all sites run their own DNS servers—and not all sites need their own DNS servers! In sufficiently small sites with no Internet connectivity, it's reasonable for each host to keep its own copy of a table matching all of the hostnames on the local network with their corresponding IP addresses. In most Linux and UNIX systems, this table is stored in the **/etc/hosts** file.

The **/etc/hosts** file keeps its information in a simple tabular format in a plain-text file. The IP address is in the first column, and all the related hostnames are in the second column. The third column is typically used to store the short version of the hostname. Only white space separates the fields. Any *text* from the pound (#) symbol until the end of the line represents a comment and thus is ignored. Here's an example:

```
# This is a comment line. Just like all the others with # symbol
127.0.0.1     localhost.localdomain  localhost
::1           localhost6.localdomain6  localhost6
192.168.1.1   serverA.example.org   serverA # Comment - Linux server
192.168.1.2   serverB.example.org   serverB # Other Linux server
192.168.1.7   dikkog                       # Win 10 server
192.168.1.9   sassy            # FreeBSD box and Lunchroom Print server
```

In general, your **/etc/hosts** file should contain, at the very least, the necessary host-to-IP mappings for the loopback interface (127.0.0.1 for IPv4 and ::1 for IPv6) and the local hostname with its corresponding IP address.

A more robust and scalable naming service (than **/etc/hosts**) is the DNS system. The rest of this chapter will cover the use of the DNS name service.

How DNS Works

In this section, we'll explore some background material necessary to your understanding of the installation and configuration of a DNS server and client.

Domain and Host Naming Conventions

Until now, you've most likely referenced sites by their *fully qualified domain name (FQDN),* like this one: www.kernel.org. Each string between the periods (or dots) in this FQDN is significant. Starting from the right and moving to the left are the top-level domain component, the second-level domain component, and the third-level domain component. This is illustrated further in Figure 17-1 in the FQDN for a system (serverA.example.org) and is a classic example of an FQDN. Its breakdown is discussed in detail in the following section.

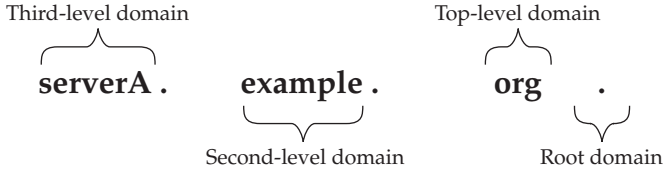


Figure 17-1 FQDN for serverA.example.org

The Root Domain

The DNS structure is like that of an inverted (upside-down) tree; this, therefore, means that the root of the tree is at the top and its leaves and branches are at the bottom! Funny sort of tree, eh?

At the top of the inverted domain tree is the highest level of the DNS structure, aptly called the *root domain* and represented by the simple dot (.).

This is the dot that's supposed to occur after every FQDN, but it is silently assumed to be present even though it is not explicitly written. Thus, for example, the proper FQDN for www.kernel.org is really www.kernel.org. (with the root dot at the end), and the FQDN for the popular web portal for Yahoo! is actually www.yahoo.com. (the trailing period is used here, too).

Coincidentally, this portion of the domain namespace is managed by a bunch of special servers known as the *root name servers.* At the time of this writing, a total of 13 names are used for distinguishing between the root name servers, which are managed by 13 providers. Many more physical servers and clusters are supported via *anycast* addressing.

Each provider may have multiple servers (or clusters) that are spread all over the world. The servers are distributed for various reasons, such as security and load balancing. Also at the time of this writing, all the root name servers fully support IPv6-type record sets. The root name servers are named alphabetically, with names like a.root-server.net, b.root-server .net, … m.root-server.net. The role of the root name servers will be discussed a bit later.

The Top-Level Domain Names

The top-level domains (TLDs) can be regarded as the first branches that we would meet on the way down from the top of our inverted tree structure.

The top-level domains provide the categorical organization of the DNS namespace. What this means in plain English is that the various branches of domain namespace have been divided into clear categories to fit different uses (examples of such uses could be geographical, functional, and so on). At the time of this writing, there are more than 1534 (and growing) top-level domains.

The TLDs can be broken down further:

- Generic top-level domain (such as .org, .com, .net, .mil, .gov, .edu, .int, .biz, and so on).
- Country-code top-level domains (such as .us, .uk, .ng, and .ca, corresponding to the country codes for the United States, the United Kingdom, Nigeria, and Canada, respectively), as well as their internationalized counterparts (such as .भारत for India, .中国 for China, .рф for Russia, and so on).
- Branded top-level domains. These allow organizations to create any TLDs with up to 64 characters. They can include generic words and brand names (such as .coke, .pepsi, .example, .linux, .microsoft, .caffenix, .who, .unicef, .companyname, .accountants, .beer, .io, and so on).
- Other special or infrastructure top-level domains (such as the .arpa domain).

The top-level domain in our sample FQDN (serverA.example.org.) is ".org."

The Second-Level Domain Names

The names at this level of the DNS make up the actual organizational boundary of the namespace. Companies, Internet service providers (ISPs), educational communities, nonprofit groups, and individuals typically acquire unique names within this level. Here are a few examples: redhat.com, ubuntu.com, fedoraproject.org, linuxserverexperts.com, kernel.org, and caffenix.com.

The second-level domain in our sample FQDN (serverA.example.org.) is "example."

The Third-Level Domain Names

Individuals and organizations that have been assigned second-level domain names can pretty much decide what to do with the third-level names. The convention, though, is to use the third-level names to reflect hostnames or other functional uses. It is also common for organizations to begin the subdomain definitions from here. An example of functional assignment of a third-level domain name is the "www" in the FQDN www.yahoo.com. The "www" here can be the actual hostname of a machine under the umbrella of the yahoo.com domain, or it can be an alias to a real hostname.

The third-level domain name in our sample FQDN (serverA.example.org.) is "serverA." Here, it simply reflects the actual hostname of our server.

By keeping DNS distributed in this manner, the task of keeping track of all the hosts connected to the Internet is delegated to each site, which takes care of its own information.

The inverted tree structure of DNS is shown in Figure 17-2.

Subdomain

"But I just saw the site www.support.example.org!" you say. "What's the hostname component, and what's the domain name component?" you might ask.

Welcome to the wild and mysterious world of *subdomains.* A subdomain exhibits all the properties of a domain, except that it has delegated a subsection of the domain instead of all the hosts at a site. Using the example.org site as an example, the subdomain for the support and help desk department of Example, Inc., is support.example.org. When the primary name server for the example.org domain receives a request for a hostname whose FQDN ends in support.example.org, the primary name server forwards the request down to the primary

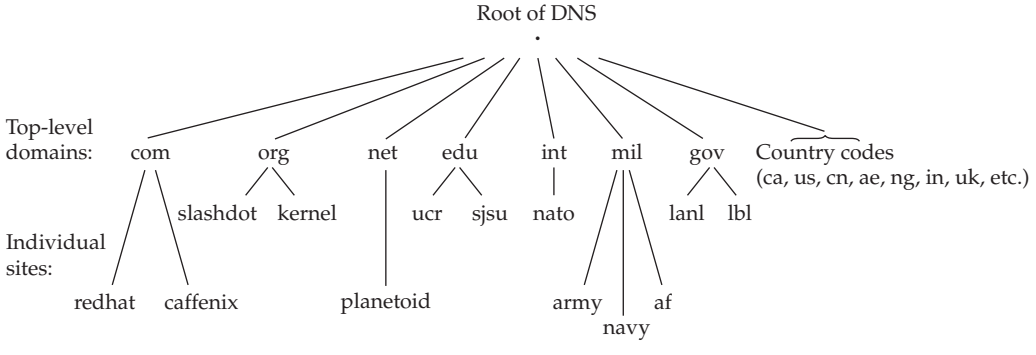


Figure 17-2 The DNS tree, two layers deep

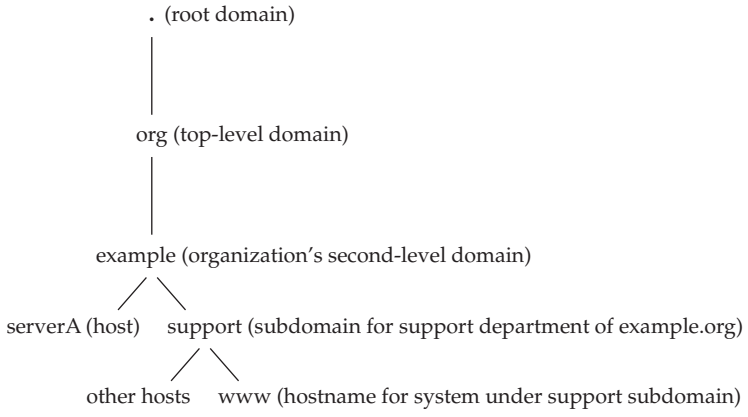


Figure 17-3 Concept of subdomains

name server for support.example.org. Only the primary name server for support.example.org knows all the hosts existing beneath it—hosts such as a system named "www" with the FQDN of www.support.example.org.

Figure 17-3 shows you the relationship from the root servers down to example.org and then to support.example.org. The "www" is, of course, the hostname.

To make this clearer, let's follow the path of a DNS request:

1. A client wants to visit a web site called "www.support.example.org."
2. The query starts with the top-level domain "org." Within "org." is "example.org."
3. Let's say one of the authoritative DNS servers for the "example.org" domain is named "ns1.example.org," or "ns1" for short. Since the host ns1 is authoritative for the example.org domain, we have to query it for all hosts (and subdomains) under it. So we query it for information about the host we are interested in: "www.support .example.org."
4. Now ns1.example.org's DNS configuration is such that for anything ending with "support.example.org," the server must contact another authoritative server called "dns2.example.org."
5. The request for "www.support.example.org" is then passed on to dns2.example.org, which returns the IP address for www.support.example.org—say, 192.168.1.10.

Note that when a site name appears to reflect the presence of subdomains, it doesn't mean subdomains in fact exist. Although the hostname specification rules do not allow periods, the Berkeley Internet Name Domain (BIND) name server has always allowed them. Thus, from time to time, you will see periods used in hostnames. Whether or not a subdomain exists is handled by the configuration of the DNS server for the site. For example, www.bogus .example.org does not automatically imply that bogus.example.org is a subdomain. Rather, it may also mean that www.bogus is the hostname for a system in the "example.org" domain.

The in-addr.arpa Domain

DNS allows resolution to work in both directions. *Forward resolution* converts names into IP addresses, and *reverse resolution* converts IP addresses back into hostnames. The process of reverse resolution relies on the in-addr.arpa domain, where arpa is an acronym for Address Routing and Parameters Area.

As explained in the preceding section, domain names are resolved by looking at each component from right to left, with the suffixing period indicating the root of the DNS tree. Following this logic, IP addresses must have a top-level domain as well. This domain is called in-addr.arpa for IPv4-type addresses. In IPv6, the domain is called ip6.arpa.

Unlike FQDNs, IP addresses are resolved from left to right once they're under the in-addr .arpa domain. Each octet further narrows down the possible hostnames. Figure 17-4 provides a visual example of reverse resolution of the IP address 138.23.169.15.

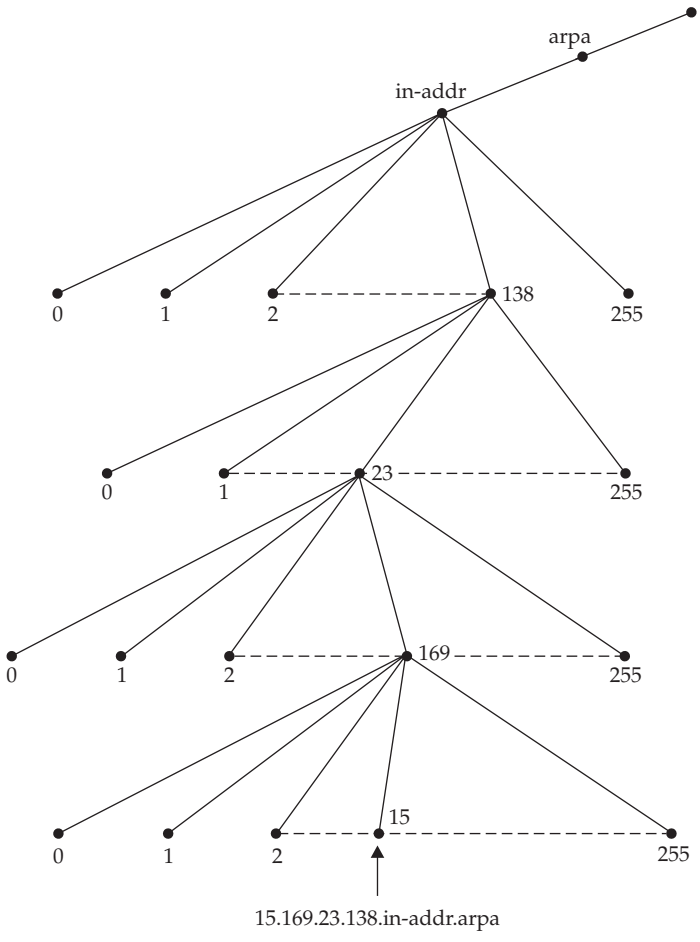


Figure 17-4 Reverse DNS resolution of 138.23.169.15

Types of Servers

DNS servers come in three flavors: primary, secondary, and caching. Another special class of name servers consists of the so-called "root name servers." Other DNS servers require the services provided by the root name servers every once in a while.

The three main flavors of DNS servers are discussed next.

Primary servers are considered authoritative for a particular domain. An *authoritative server* is the one on which the domain's configuration files reside. When updates to the domain's DNS tables occur, they are done on this server. A primary name server for a domain is simply a DNS server that knows about all hosts and subdomains existing under its domain.

Secondary servers work as backups and as load distributors for the primary name servers. Primary servers know of the existence of secondaries and send them periodic notifications/alerts of changes to the name tables. The secondary then initiates a zone transfer to pull in the actual changes.

When a site queries a secondary name server, the secondary responds with authority. However, because it's possible for a secondary to be queried before its primary can alert it to the latest changes, some people refer to secondaries as "not quite authoritative." Realistically speaking, you can generally trust secondaries to have correct information. (Besides, unless you know which is which, you cannot tell the difference between a query response from a primary and one received from a secondary.)

Root Name Servers

The root name servers act as the first port of call for the topmost parts of the domain namespace. These servers publish a file called the "root zone file" to other DNS servers and clients on the Internet. The root zone file describes or lists the authoritative servers for the DNS top-level domains (.com, .org, .ca, .ng, .hk, .uk, and so on).

Obviously, a list of such a critical nature is itself mirrored across multiple servers and multiple geographic regions. For example, an earthquake in one part of the world might destroy the root server(s) for that area, but all the root servers in other, unaffected parts of the world can take up the slack until the affected servers come back online. The only noticeable difference to users is likely to be a slightly higher latency in resolving domain names.

A root name server is simply an instance of a primary name server—it delegates every request it gets to another name server. You can build your own root server out of BIND—nothing terribly magical about it!

Caching servers are just that: caching servers. They contain no configuration files for any particular domain. Rather, when a client host requests a caching server to resolve a name, that server will check its own local cache first. If it cannot find a match, it will find the

primary server and ask it. This response is then cached. Practically speaking, caching servers work quite well because of the temporal nature of DNS requests. Their effectiveness is based on the premise that if you've asked for the IP address to example.org in the past, you are likely to do so again in the near future. Clients can tell the difference between a caching server and a primary or secondary server, because when a caching server answers a request, it answers it "non-authoritatively."

NOTE A DNS server can be configured to act with a specific level of authority for a particular domain. For example, a server can be primary for example.org but be secondary for domain.com. All DNS servers can act as caching servers, even if they are also primary or secondary for any other domains.

Installing a DNS Server

Several choices exist in the Linux/FOSS world when it comes to implementing DNS functionality. In this chapter, we will focus on the venerable BIND DNS server implementation. It is used on a vast majority of name-serving machines worldwide. BIND is currently maintained and developed by the Internet Systems Consortium (ISC). You can learn more about the ISC at www.isc.org. The ISC is also in charge of development of the ISC Dynamic Host Configuration Protocol (DHCP) server/client as well as other software.

NOTE Because of the timing between writing this book and the inevitable release of newer software, it is possible that the version of BIND discussed here will not be the same as the version to which you will have access; but you shouldn't worry at all, because most of the configuration directives, keywords, and command syntax have remained much the same between recent versions of the software.

Our sample target system runs the Fedora distribution of Linux, and, as such, we will be using the precompiled binary that ships with this OS. Software that ships with Fedora is supposed to be very recent software, so you can be sure that the version of BIND referred to here is close to the latest version that can be obtained directly from the www.isc.org site (the site even has precompiled binaries for the BIND program in the RPM format!).

The good news is that once BIND is configured, you'll rarely need to concern yourself with its operation. Nevertheless, keep an eye out for new releases. New bugs and security issues are discovered from time to time and should be corrected. Of course, new features are released as well, but unless you have a need for them, those releases are less critical.

If you have a working connection to the Internet, installing BIND on Red Hat–based distros like RHEL, CentOS, and Fedora can be as simple as running this command:

```
[master@fedora-server ~]$ sudo dnf -y install bind
```

Once this command finishes, you are ready to begin configuring the DNS server.

Downloading, Compiling, and Installing the ISC BIND Software from Source

If the ISC BIND software is not available in a prepackaged form for your particular Linux distribution, you can always build the software from source code available from the ISC site at www.isc.org. Another reason to build from source might be to take advantage of the most recent bug fixes available for the software, which your distribution may not have implemented yet. As of this writing, the most current stable version of the software is the 9.15.* series (substitute the asterisk for the highest available version number), which can be downloaded directly from the directory here:

http://ftp.isc.org/isc/bind9/cur/9.15/

Make sure that you have the **openssl-devel** and **libcap-devel** packages installed on your RPM-based distro before attempting to compile and build BIND from source. The equivalent packages in the Debian/Ubuntu world are **libssl-dev** and **libcap-dev**. The package ensures that you have the necessary library/header files available to support some of the advanced security features of BIND.

Once the package is downloaded, unpack the software as shown. For this example, we assume the source was downloaded into the **/usr/local/src/** directory. Unpack the tarball thus:

```
tar xvzf bind-9.15.*.tar.gz
```

Change to the **bind*** subdirectory created by the preceding command, and then take a minute to study any README file(s) that might be present.

Next, configure the package with the **`configure`** command. Assuming we want BIND to be installed under the **/usr/local/named/** directory, we'll run this:

```
# ./configure  --prefix=/usr/local/named
```

To compile and install **bind** under the directory specified by the **`prefix`** option, type the following:

```
# make ; make install
```

The version of ISC BIND software that we built from source installs the name server daemon (**named**) and some other useful utilities under the **/usr/local/named/sbin/** directory. The client-side programs (**`dig`**, **`host`**, **`nsupdate`**, and so on) are installed under the **/usr/local/named/bin/** directory.

What Was Installed

Many programs come with the main **bind** and **bind-utils** packages that were installed earlier using the distro package manager. We are interested in the following four tools:

- **/usr/sbin/named** The main DNS server program
- **/usr/sbin/rndc** The **bind** name server control utility
- **/usr/bin/host** A program that performs a simple query on a name server
- **/usr/bin/dig** A program that performs complex queries on a name server

The remainder of the chapter will discuss some of the programs/utilities listed here, as well as their configuration and usage.

Understanding the BIND Configuration File

The **named.conf** file is the main BIND configuration file. BIND consults this file to determine how it should behave and what additional configuration files, if any, must be read. This section covers what you need to know to set up a general-purpose DNS server. You'll find a complete guide to the new configuration file format in the **html** directory of BIND's documentation.

The general format of the **named.conf** file is as follows:

```
statement {
      clauses;  // comments
};
```

The **statement** keyword tells BIND we're about to describe a particular facet of its operation, and **clauses** are the specific commands/options applying to that statement. The curly braces are required so that BIND knows which clauses are related to which statements; a semicolon appears after every clause and after the closing curly brace.

An example of this follows:

```
options {
      directory "/var/named"; // put config files in /var/named
};
```

This BIND statement means that this is an **options** statement, and the **directory** clause here is the directive that specifies BIND's working directory—that is, the directory on the local file system that will hold the name server's configuration data.

The Specifics

This section documents the most common statements you will see in a typical **named.conf** file. The best way to tackle this is to skim it first and then treat it as a reference guide for later sections. If some of the directives seem bizarre or don't quite make sense to you during the first pass, don't worry. Once you see them in use in later sections, the hows and whys will quickly fall into place.

Comments

Comments can be in one of the following formats:

Format	Indicates
`//`	C++-style comments
`/*...*/`	C-style comments
#	Perl and UNIX shell script–style comments

In the case of the first and last styles (C++ and Perl/UNIX shell), once a comment begins, it continues until the end of the line. In regular C-style comments, the closing `*/` is required to indicate the end of a comment. This makes C-style comments easier for multiline comments. In general, however, you can pick the comment format that you like best and stick with it.

Statement Keywords

The following are some of the more frequently used **statement** keywords:

Keyword	Description
`acl`	Access Control List—determines what kind of access others have to your DNS server.
`include`	Allows you to include another file and have that file treated as part of the normal **named.conf** file.
`logging`	Specifies what information gets logged and what gets ignored. For logged information, you can also specify where the information is logged.
`options`	Specifies configuration directives that affect the overall server operations.
`controls`	Allows you to declare control channels for use by the **`rndc`** utility.
`server`	Sets server-specific configuration options.
`zone`	Defines a DNS zone.

We discuss four of the most often used keywords (**`include`**, **`logging`**, **`server`**, and **`zone`**) in more detail in the next sections.

The include Statement If you find that your configuration file is starting to grow unwieldy, you may want to consider breaking up the file into smaller components. The contents of each file can then be referenced in the main **named.conf** file via an **`include`** statement. Note that you cannot use the **`include`** statement inside another statement. Here's an example of an **`include`** statement:

```
include "/path/to/filename_to_be_included";
```

The logging Statement The **`logging`** statement is used to specify what information you want logged and where. When this statement is used in conjunction with the syslog facility, you get an extremely powerful and configurable logging system. The items logged include

various statistics about **named**. By default, they are logged to the **/var/log/messages** file. In their simplest form, the various types of logs have been grouped into predefined categories or channels; for example, there are categories for security-related logs, a general category, a default category, a resolver category, a queries category, and so on.

Unfortunately, the configurability of this **logging** statement comes at the price of some additional complexity, but the default logging set up by **named** is good enough for most uses. Here is a simple **logging** directive example:

```
1. logging {
2.     category default { default_syslog; };
3.     category queries { default_syslog; };
4.
5. };
```

NOTE Line numbers have been added to the preceding listing to aid readability.

The preceding logging specification means that all logs that fall under the default category will be sent to the system's **syslog** (the default category defines the logging options for categories where no specific configuration has been defined).

Line 3 in the listing specifies where all *queries* will be logged to; in this case, all queries will be logged to the system **syslog**.

The server Statement The **server** statement tells BIND specific information about other name servers it might be dealing with. The format of the server statement is as follows:

```
1. server ip-address {
2.     bogus yes/no;
3.     keys { string ; [ string ; [...]] } ; ]
4.     transfer-format one-answer/many-answers;
5.     ...<other options>...
6. };
```

Here, ***ip-address*** in line 1 is the IPv4 or IPv6 address of the remote name server.

The **bogus** option in line 2 tells the server whether the remote server is sending bad information. This is useful if you are dealing with another site that may be sending you bad information due to a misconfiguration. The **keys** clause in line 3 specifies a **key_id** defined by the **key** statement, which can be used to secure transactions when talking to the remote server. This key is used in generating a request signature that is appended to messages exchanged with the remote name server. The item in line 4, **transfer-format**, tells BIND whether the remote name server can accept multiple answers in a single query response.

A sample **server** entry might look like this:

```
server 192.168.1.12 {
      bogus no;
      transfer-format many-answers;
};
```

The Zone Statement The **zone** statement allows you to define a DNS zone—the definition of which is often confusing. Here is the fine print: A DNS zone is not the same thing as a DNS domain. The difference is subtle, but important.

Let's review: Domains are designated along organizational boundaries. A single organization can be separated into smaller administrative subdomains. Each subdomain gets its own zone. All of the zones collectively form the entire domain.

For example, .example.org is a domain. Within it are the subdomains .engr.example.org, .marketing.example.org, .sales.example.org, and .admin.example.org. Each of the four subdomains has its own zone. Also, .example.org has some hosts within it that do not fall under any of the subdomains; thus, it has a zone of its own. As a result, the example.org *domain* is actually composed of five zones in total.

In the simplest model, where a single domain has no subdomains, the definition of zone and domain are the same in terms of information regarding hosts, configurations, and so on.

Next we discuss setting up zones in the **named.conf** file.

Configuring a DNS Server

Earlier, you learned about the differences between primary, secondary, and caching name servers. To recap: Primary name servers contain the databases with the latest DNS information for a zone. When a zone administrator wants to update these databases, the primary name server gets the update first, and the rest of the world asks it for updates. Secondaries explicitly keep track of primaries, and primaries notify the secondaries when changes occur. Primaries and secondaries are considered equally authoritative in their answers. Caching name servers have no authoritative records, only cached entries.

Defining a Primary Zone in the named.conf File

The most basic syntax for a **zone** entry is as follows:

```
zone domain-name {
      type master;
      file path-name;
};
```

The ***path-name*** refers to the file containing the database information for the zone in question. For example, to create a zone for the domain example.org, where the database file is located in **/var/named/example.org.db**, you would create the following zone definition in the **named.conf** file:

```
zone "example.org" {
      type master;
      file "example.org.db";
};
```

Note that the **directory** option specified in the **named.conf** file will automatically prefix the **example.org.db** filename. So if you specified ***directory /var/named***, the server software will automatically look for example.org's information in **/var/named/example.org.db**.

The zone definition created here is just a *forward reference*—that is, the mechanism by which others can look up a name and get the IP address for a system under the example.org domain that your name server manages. It's also proper Internet etiquette to supply an IP-to-hostname mapping (also necessary if you want to send e-mail to some sites). To do this, you provide an entry in the in-addr.arpa domain.

The format of an in-addr.arpa entry is the first three octets of your IP address, *reversed*, followed by `in-addr.arpa`. Assuming that the network address for example.org is 192.168.1, the in-addr.arpa domain would be `1.168.192.in-addr.arpa`. Thus, the corresponding **`zone`** statement in the **named.conf** file would be as follows:

```
zone "1.168.192.in-addr.arpa" {
      type master;
      file "example.org.rev";
};
```

Note that the filenames (**example.org.db** and **example.org.rev**) used in the zone sections here are completely arbitrary. You are free to choose your own naming convention as long as it makes sense to you.

The exact placement of our sample example.org zone section in the overall **named.conf** file will be shown later on in the "DNS Server Setup Walkthrough" section.

Additional Options

Primary domains can also use the following configuration choices from the **`options`** statement: **`check-names`**, **`allow-update`**, **`allow-query`**, **`allow-transfer`**, **`notify`**, and **`also-notify`**. Use of any of these options in a zone configuration affects only that zone.

Defining a Secondary Zone in the named.conf File

The zone entry format for secondary servers is similar to that of master servers. For forward resolution, here is the format:

```
zone domain-name {
      type slave;
      masters IP-address-list; ;
      file path-name;
};
```

Here, ***`domain-name`*** is the exact same zone name as specified on the primary name server, ***`IP-address-list`*** is the list of IP addresses where the primary name server for that zone exists, and ***`path-name`*** is the full path location of where the server will keep copies of the primary's zone files.

Additional Options

A secondary zone configuration can also use some of the configuration choices from the **`options`** statement, such as **`check-names`**, **`allow-update`**, **`allow-query`**, **`allow-transfer`**, and **`max-transfer-time-in`**.

Defining a Caching Zone in the named.conf File

A caching configuration is the easiest of all configurations. It's also required for every DNS server configuration, even if you are running a primary or secondary server. This is necessary for the server to search the DNS tree recursively to find other hosts on the Internet.

For a caching name server, we define three zone sections. Here's the first entry:

```
zone "." IN {
      type hint;
      file "root.hints";
};
```

The first zone entry here is the definition of the root name servers. The line **`type hint;`** specifies that this is a caching zone entry, and the line **`file "root.hints";`** specifies the file that will prime the cache with entries pointing to the root servers. You can always obtain the latest root hints file from www.internic.net/zones/named.root.

The second zone entry (defined in the **/etc/named.rfc1912.zones** file via the **`Include`** directive on Red Hat–like distros) defines the name resolution for the localhost. The second zone entry is as follows:

```
zone "localhost" IN {
      type master;
      file "localhost.db";
};
```

The third zone entry defines the reverse lookup for the localhost. This is the reverse entry for resolving the localhost address (127.0.0.1) back to the local hostname:

```
zone "0.0.127.in-addr.arpa" IN {
      type master;
      file "127.0.0.rev";
};
```

Putting these zone entries into **/etc/named.conf** is sufficient to create a caching DNS server. But, of course, the contents of the actual database files (**localhost.db**, **127.0.0.rev**, **example.org.db**, and so on) referenced by the **`file`** directive are also important. The following sections will examine the makeup of the database file more closely.

DNS Records Types

This section discusses the makeup of the *name server database files*—the files that store specific information that pertains to each zone the server hosts. The database files consist mostly of record types; therefore, you need to understand the meaning and use of the common record types for DNS: SOA, NS, A, PTR, CNAME, MX, TXT, and RP.

SOA: Start of Authority

The SOA record starts the description of a site's DNS entries. Here is the format of this entry:

```
1.  domain.name. IN SOA ns.domain.name. hostmaster.domain.name. (
2.          1999080801          ; serial number
3.          10800               ; refresh rate in seconds (3 hours)
4.          1800                ; retry in seconds (30 minutes)
5.          1209600             ; expire in seconds (2 weeks)
6.          604800              ; minimum in seconds (1 week)
7.         )
```

NOTE Line numbers have been added to the preceding listing to aid readability.

The first line contains some details you need to pay attention to: **`domain.name`** is, of course, to be replaced with your domain name. This is usually the same name that was specified in the **`zone`** directive in the **/etc/named.conf** file. That last period at the end of **`domain.name.`** is supposed to be there—indeed, the DNS configuration files are extremely picky about it. The ending period is necessary for the server to differentiate relative hostnames from FQDNs; for example, it signifies the difference between serverA and serverA.example.org.

`IN` tells the name server that this is an Internet record. There are other types of records, but it's been years since anyone has had a need for them—you can thus safely ignore them.

`SOA` tells the name server this is the Start of Authority record.

`ns.domain.name.` is the FQDN for the name server for this domain (that would be the server where this file will finally reside). Again, watch out and don't miss that trailing period.

`hostmaster.domain.name.` is the e-mail address for the domain administrator. Notice the lack of an @ in this address. The leftmost period is replaced internally with the @ symbol. Thus, the e-mail address referred to in this example is hostmaster@domain.name. The trailing period is used here, too.

The remainder of the record starts after the opening parenthesis on line 1. Line 2 is the serial number. It is used to tell the name server when the file has been updated. Watch out—forgetting to increment this number when you make a change is a mistake frequently made in managing DNS records. (Forgetting to put a period in the right place is another common error.)

TIP To maintain serial numbers in a sensible way, use the date formatted in the following order: YYYYMMDD*xx*. The tail-end *xx* is an additional two-digit number, starting with 00, so if you make multiple updates in a day, you can still tell which is which.

Line 3 in the list of values is the refresh rate in seconds. This value tells the secondary DNS servers how often they should query the primary server to see if the records have been updated.

Line 4 is the retry rate in seconds. If the secondary server tries but cannot contact the primary DNS server to check for updates, the secondary server tries again after the specified number of seconds.

Line 5 specifies the expire directive. It is intended for secondary servers that have cached the zone data. It tells these servers that if they cannot contact the primary server for an update, they should discard the value after the specified number of seconds. One to two weeks is a good value for this interval.

The final value (line 6, the minimum) tells caching servers how long they should wait before expiring an entry if they cannot contact the primary DNS server. Five to seven days is a good guideline for this entry.

TIP Don't forget to place the closing parenthesis (line 7) after the final entry.

NS: Name Server

The NS record is used for specifying which name servers maintain records for this zone. If any secondary name servers exist that you intend to transfer zones to, they need to be specified here. The format of this record is as follows:

```
IN NS ns1.domain.name.
IN NS ns2.domain.name.
```

You can have as many backup name servers as you'd like for a domain—at least two is good.

A and AAAA: Address Records

This is probably the most common type of record found in the wild. The "A" record is used to provide a mapping from hostname to IP address. The format of an A record is simple:

```
Host_name  IN  A  IP-Address
```

For example, an A record for the host serverB.example.org, whose IP address is 192.168.1.2, would look like this:

```
serverB  IN  A  192.168.1.2
```

The equivalent of the IPv4 A resource record in the IPv6 world is called the "AAAA" (quad-A) resource record. For example, a quad-A record for the host serverB whose IPv6 address is 2001:DB8::2 would look like this:

```
serverB  IN  AAAA  2001:DB8::2
```

Note that any hostname is automatically suffixed with the domain name listed in the SOA record, unless this hostname ends with a period. In the foregoing example for serverB, if the SOA record prior to it is for example.org, then serverB is understood to be serverB .example.org. If you were to change this to serverB.example.org (without a trailing period), the name server would understand it to be serverB.example.org.example.org.—which is probably not what you intended! So if you want to use the FQDN, be sure to suffix it with a period.

PTR: Pointer Record

The PTR record is for performing reverse name resolution, thereby allowing someone to specify an IP address and determine the corresponding hostname. The format for this record is similar to the A record, except with the values reversed:

```
IP-Address   IN   PTR   Host_name
```

The ***IP-Address*** can take one of two forms: just the last octet of the IP address (leaving the name server to suffix it automatically with the information it has from the in-addr.arpa domain name) or the full IP address, which is suffixed with a period. The ***Host_name*** must have the complete FQDN. For example, the PTR record for the host serverB would be as follows:

```
192.168.1.2.   IN    PTR    serverB.example.org.
```

A PTR resource record for an IPv6 address in the ip6.arpa domain is expressed similarly to the way it is done for an IPv4 address, but in reverse order. Unlike in the normal IPv6 way, the address cannot be compressed or abbreviated; it is expressed in the so-called "reverse nibble format" (four-bit aggregation). Therefore, with a PTR record for the host with the IPv6 address 2001:DB8::2, the address will have to be expanded to its equivalent of 2001:0db8:0000:0000:0000:0000:0000:0002.

For example, here's the IPv6 equivalent for a PTR record for the host serverB with the IPv6 address 2001:DB8::2 (single line):

```
2.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.8.b.d.0.1.0.0.2. IN   PTR \
serverB.example.org.
```

MX: Mail Exchanger

The MX record is in charge of telling other sites about your zone's mail server. If a host on your network generates an outgoing mail message with its hostname on it, someone returning a message would *not* blindly send it back directly to that host. Instead, the replying mail server would look up the MX record for that site and send the message there.

For example, MX records are used when a user's desktop named pc.domain. sends a message using its PC-based e-mail client/reader, which cannot itself accept Simple Mail Transfer Protocol (SMTP) mail; it's important that the replying party have a reliable way of knowing the identity of pc.domain's mail server name.

The format of the MX record is as follows:

```
domainname.   IN   MX   weight   Host_name
```

Here, ***domainname.*** is the domain name of the site (with a period at the end, of course); the ***weight*** is the importance of the mail server (if multiple mail servers exist, the one with the smallest number has precedence over those with larger numbers); and the ***Host_name*** is, of course, the name of the mail server. It is important that the ***Host_name*** have an A record as well. Here's a sample entry:

```
example.org.    IN   MX   10   smtp1
                IN   MX   20   smtp2
```

Typically, MX records occur close to the top of DNS configuration files. If a domain name is not specified, the default name is pulled from the SOA record.

CNAME: Canonical Name

CNAME records allow you to create aliases for hostnames. A CNAME record can be regarded as an alias. This is useful when you want to provide a highly available service with an easy-to-remember name, but still give the host a real name.

Another popular use for CNAMEs is to "create" a new server with an easy-to-remember name without having to invest in a new server at all. Here's an example: Suppose a site has a web server with a real hostname of zabtsuj-content.example.org. It can be argued that zabtsuj-content.example.org is neither a memorable nor user-friendly name. Therefore, since the system is a web server, a CNAME record (or alias) of "www" might be appropriate and is thus created for the host. This will simply map the user-unfriendly name of zabtsuj-content .example.org to a more user-friendly name of www.example.org. This will allow all requests that go to www.example.org to be passed on transparently to the actual system that hosts the web content—that is, zabtsuj-content.example.org.

Here's the format for the CNAME record:

```
New_host_name   IN   CNAME   old_host_name
```

For our sample scenario, the complete CNAME entry will be the following:

```
zabtsuj-content   IN    A     192.168.1.111
www               IN    CNAME   zabtsuj-content
```

RP and TXT: The Documentation Entries

Sometimes it's useful to provide contact information as part of your database—not just as comments, but as actual records that others can query. This can be accomplished using the RP (Responsible Person) and TXT records.

A TXT record is a free-form text entry into which you can place whatever information you deem fit. Most often, you'll want to put only contact information in these records. Each TXT record must be tied to a particular hostname. Here's an example:

```
serverA.example.org.    IN   TXT   "Contact: Admin Guy"
                        IN   TXT   "SysAdmin/Android"
                        IN   TXT   "Voice: 999-999-9999"
```

The RP record was created as an explicit container for a host's contact information. This record states who the responsible person is for the specific host; here's an example:

```
serverB.example.org.  IN  RP  admin-address.example.org.  example.org.
```

As useful as these RP records may be, they are a rarity these days, because it is perceived that they give away too much information about the site that could lead to social engineering–based attacks. The TXT records, on the other hand, are gaining a resurgence in many applications. You may find these record types helpful (or even necessary) on your internal- or external-facing DNS servers.

Setting Up BIND Database Files

So now that you know enough about all the DNS record types to get started, it's time to create the actual database that will feed the server. The database file format is not too strict, but some conventions have jelled over time. Sticking to these conventions will make your life easier and will smooth the way for the administrator who takes over your creation.

TIP Remember to add descriptive comments liberally to the BIND configuration files.

The database files are your most important configuration files. It is easy to create the forward lookup databases; what usually gets left out are the reverse lookups. Some applications will routinely perform reverse lookups on IP addresses to see where requests are coming from, so it is a common courtesy to have this information.

Every database file should start with a **`$TTL`** entry. This entry tells BIND what the time-to-live value is for each individual record whenever it isn't explicitly specified. (The time-to-live, or TTL, in the SOA record is for the SOA record only.) After the **`$TTL`** entry is the SOA record and at least one NS record. Everything else is optional. (Of course, "everything else" is what makes the file useful!) You might find the following general format helpful:

```
$TTL
SOA  record
NS   records
MX   records
A  and  CNAME records
```

Let's walk through the process of building a complete configuration for a DNS server from start to finish to demonstrate how the information shown thus far comes together. For this example, we will build DNS servers for example.org that will accomplish the following goals:

- Establish two name servers: ns1.example.org and ns2.example.org.
- The name servers will be able to respond to queries for IPv6 records of which they are aware.
- Act as a slave server for the sales.example.org zone, where serverB.example.org will be the master server.
- Define A records for serverA, serverB, smtp, ns1, and ns2.
- Define AAAA records (IPv6) for serverA-v6 and serverB-v6.
- Define smtp.example.org as the mail exchanger (MX) for the example.org domain.
- Define www.example.org as an alternative name (CNAME) for serverA.example.org, and define ftp.example.org as an alternative name for serverB.example.org.
- Define contact information for serverA.example.org.

Okay, Mr. Bond, you have your instructions. Go forth and complete the mission. Good luck!

DNS Server Setup Walkthrough

To accomplish our goal of setting up a DNS server for example.org, we will need to take a series of steps. Let's walk through them one at a time.

TIP If you don't feel like doing too much typing, we've bundled all the files that you'll need to follow along and saved them for you at the following URL. You just need to download, uncompress, and copy the files to the proper directories stated in the steps, like so:

```
# pwd
# wget  http://linuxserverexperts.com/8e/chapter-17.tar.xz
# tar  xvJf  chapter-17.tar.xz
# cd chapter-17 && ls
```

1. Make sure that you have installed the BIND DNS server software as described earlier in the chapter. Depending on your distro, use the **rpm** or **dpkg** command to confirm this:

   ```
   [master@fedora-server ~]$ rpm  -q  bind
   ```

   And for a Debian-based distro, type:

   ```
   master@ubuntu-server:~$ dpkg  -l  bind9
   ```

NOTE If you only built and installed BIND from source, your distro package manager will not reveal anything because the RPM or dpkg database will not know anything about it. However, you would know what you installed and where.

2. Use any text editor you are comfortable with to create the main DNS server configuration file—the **/etc/named.conf** file. Enter the following text into the file:

```
options {
           listen-on      port 53 { any; };
           listen-on-v6   port 53 { any; };
           directory      "/var/named";
         dump-file       "/var/named/data/cache_dump.db";
         statistics-file "/var/named/data/named_stats.txt";
         notify    yes;
};
# The following zone definitions don't need any modification.
#  The 1st is the definition of the root name servers #  and sets up
our server as a caching-capable
# DNS server. The 2nd defines localhost. The 3rd zone
#  definition defines the reverse lookup for localhost.
zone "." in {
      type hint;
      file "root.hints";
};
zone "localhost" in {
      type master;
      file "localhost.db";
};
zone "0.0.127.in-addr.arpa" in {
      type  master;
      file "127.0.0.rev";
};
```

```
# The zone definition below is for the domain
# that our name server is authoritative for i.e.
# the example.org domain name.
zone "example.org" {
      type master;
      file "example.org.db";
};
# Below is the zone for the in-addr.arpa domain,
# for the example.org site.
zone "1.168.192.in-addr.arpa" {
      type master;
      file "example.org.rev";
};
# Below is the entry for the subdomain for which
# this server is a slave server
# IP address of sales.example.orgs master server is 192.168.1.2
zone "sales.example.org" {
       type slave;
       file "sales.example.org.bk";
       masters {192.168.1.2;};
};
# Below is the zone for the ip6.arpa domain for
# the example.org site. The zone will store its data in
# the same file as the 1.168.192.in-addr.arpa domain
zone "0.0.0.0.0.0.0.0.8.b.d.0.1.0.0.2.ip6.arpa"  {
         type master;
         file "example.org.rev";
};
```

3. Save the preceding file as **/etc/named.conf** and exit the text editor.
4. Next, we'll need to create the actual database files referenced in the file sections of the **/etc/named.conf** file. In particular, the files we want to create are **root.hints**, **localhost.db**, **127.0.0.rev**, **example.org.db**, and **example.org.rev**. All the files will be stored in BIND's working directory, **/var/named/**. We'll create them as they occur from the top of the **named.conf** file to the bottom.
5. Thankfully, we won't have to create the root hints file manually. Download the latest copy of the root hints file from the Internet. Use the **wget** command to download and copy it in the proper directory:

```
# wget -O /var/named/root.hints http://www.internic.net/zones/named.root
```

6. Use any text editor you are comfortable with to create the zone file for the local host. This is the **localhost.db** file. Enter the following text into the file:

```
$TTL 1W
@       IN SOA        localhost   root (
                      2006123100     ; serial
                      3H             ; refresh (3 hours)
                      30M            ; retry (30 minutes)
                      2W             ; expiry (2 weeks)
                      1W)            ; minimum (1 week)
        IN    NS      @
        IN    A       127.0.0.1
```

7. Save the preceding file as **/var/named/localhost.db** and exit the text editor.
8. Use any text editor to create the zone file for the reverse lookup zone for the local host. This is the **127.0.0.rev** file. Enter the following text into the file:

```
$TTL 1W
@      IN SOA    localhost.   root.localhost. (
                        2006123100      ; serial
                        3H              ; refresh
                        30M             ; retry
                        2W              ; expiry
                        1W )            ; minimum
       IN     NS        localhost.
1      IN     PTR       localhost.
```

TIP It is possible to use abbreviated time values in BIND. For example, **3H** means 3 hours, **2W** means 2 weeks, **30M** implies 30 minutes, and so on.

9. Save the preceding file as **/var/named/127.0.0.rev** and exit the text editor.
10. Next, create the database file for the main zone of concern—that is, the example.org domain. Use a text editor to create the **example.org.db** file and then input the following text into the file:

```
$TTL 1W
@     IN SOA       ns1.example.org.    root (
                  2009123100    ; serial
                  3H            ; refresh (3 hours)
                  30M           ; retry (30 minutes)
                  2W            ; expiry (2 weeks)
                  1W)           ; minimum (1 week)
                IN    NS        ns1.example.org.
                IN    NS        ns2.example.org.
                IN    MX   10   smtp.example.org.
ns1             IN    A     192.168.1.1   ;primary name server
ns2             IN    A     192.168.1.2   ;secondary name server
serverA         IN    A         192.168.1.1
serverB         IN    A         192.168.1.2
smtp            IN    A         192.168.1.25      ;mail server
www             IN    CNAME     serverA          ;web server
ftp             IN    CNAME     serverB          ;ftp server
serverA         IN    TXT       "Fax: 999-999-9999"
; IPv6 entries for serverA (serverA-v6) and
; serverB (serverB-v6) are below
serverA-v6      IN    AAAA      2001:DB8::1
serverB-v6      IN    AAAA      2001:DB8::2
```

11. Save the preceding file as **/var/named/example.org.db** and exit the text editor.

12. Create the reverse lookup zone file for the example.org zone. Use a text editor to create the **/var/named/example.org.rev** file and then input the following text into the file:

```
$TTL 1W
@    IN  SOA     ns1.example.org.   root (
            2009123100  ; serial
            3H    ; refresh (3 hours)
            30M   ; retry (30 minutes)
            2W    ; expiry (2 weeks)
            1W)   ; minimum (1 week)
            IN    NS    ns1.example.org.
            IN    NS    ns2.example.org.
1       IN    PTR   serverA.example.org.  ; Reverse info for serverA
2       IN    PTR   serverB.example.org.  ; Reverse info for serverB
25      IN    PTR   smtp.example.org.     ; Reverse for mailserver
; IPv6 PTR entries for serverA (serverA-v6) and serverB (serverB-v6) are below
$ORIGIN 0.0.0.0.0.0.0.0.8.b.d.0.1.0.0.2.ip6.arpa.
1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0  IN  PTR  serverA-v6.example.org.
2.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0  IN  PTR  serverB-v6.example.org.
```

13. We don't have to create any files to be secondary for sales.example.com. We need to add only the entries we already have in the **named.conf** file. (Although the log files will complain about not being able to contact the master, this is okay, since we have only shown how to set up the primary master for the zone for which our server is authoritative.)

    The next step will demonstrate how to start the **named** service. However, because the BIND software is so finicky about its dots and semicolons, and because you might have had to type in all the configuration files manually, chances are great that you invariably made some typos (or we made some typos ourselves). So your best bet will be to monitor the system log files carefully to view error messages as they are being generated in real time.

14. Use the **`tail`** command in another terminal window to view the logs and then issue the command in the next step in a separate window so that you can view both simultaneously. In your new terminal window, type the following:

```
[master@serverA named]$ sudo tail -f /var/log/messages
```

15. We are ready to start the **named** service at this point. Use the **`systemctl`** command to launch the service:

```
[master@serverA named]$ sudo systemctl start named.service
```

TIP On an openSUSE system, the equivalent command will be

```
[root@opensuse-serverA]#  rcnamed start
```

16. If you get a bunch of errors in the system logs, you will find that the logs will usually tell you the line number and/or the type of error. So fixing the errors shouldn't be too hard. Just go back and add the dots and semicolons where they ought to be. Another common error is misspelling the configuration file's directives—for example, writing "master" instead of "masters"; though both are valid directives, each is used in a different context.

TIP If you have changed BIND's configuration files (either the main **named.conf** or the database file), you will need to tell it to reread them by sending the named process an HUP signal. The quick and easy way to do this is by reloading or restarting the daemon using your system's service manager utility (such as **service**, **systemctl**, **rcnamed**, and so on).

17. Finally, you might want to make sure that your DNS server service starts up during the next system reboot. Use the **systemctl** command:

```
[master@serverA named]$ sudo systemctl enable named.service
```

The next section walks through the tools that can be used to test or query a DNS server.

The DNS Toolbox

This section describes a few tools that you'll want to get acquainted with as you work with DNS. They'll help you to troubleshoot problems more quickly.

host

The **host** tool is really a simple utility to use. Its functionality can, of course, be extended by using it with its various options. Its syntax and some of its common options are shown here:

```
host [-aCdlrTwv] [-c class] [-n] [-N ndots] [-t type] [-W time]
          [-R number] hostname [server]
    -a is equivalent to -v -t *
    -r disables recursive processing
    -R specifies number of retries for UDP packets
    -t specifies the query type
    -T enables TCP/IP mode
    -v enables verbose output
    -w specifies to wait forever for a reply
    -W specifies how long to wait for a reply
```

In its simplest use, **host** allows you to resolve hostnames into IP addresses from the command line. Here's an example:

```
[master@serverA ~]$ host internic.net
internic.net has address 192.0.43.9
...<OUTPUT TRUNCATED>...
```

You can also use **host** to perform reverse lookups. Here's an example:

```
[master@serverA ~]$ host 192.0.43.9
.43.0.192.in-addr.arpa domain name pointer 43-9.any.icann.org.
```

The **host** command can also be used to query for IPv6 records. For example, to query (on its listening IPv6 interface) a name server (::1) for the IPv6 address for the host serverB-v6 .example.org, you can run the following:

```
[master@serverA ~]$ host  serverB-v6.example.org  ::1
Using domain server:
Name: ::1
Address: ::1#53
Aliases:
serverB-v6.example.org has IPv6 address 2001:db8::2
```

To query for the PTR record for serverB-v6, you can use the following:

```
[master@serverA ~]$ host  2001:db8::2   ::1
Using domain server:
Name: ::1
Address: ::1#53
Aliases:
2.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.8.b.d.0.1.0.0.2.ip6.arpa \
domain name pointer serverB-v6.example.org.
```

dig

The domain information gopher (**dig**) is a great tool for gathering information about DNS servers. This tool has the BIND group's blessing and official stamp.

Its syntax and some of its options are shown here (see the **dig** man page for the meaning of the various options):

```
dig [@global-server] [domain] [q-type] [q-class] {q-opt}
            {global-d-opt} host [@local-server] {local-d-opt}
            [ host [@local-server] {local-d-opt} [...]]
Where:  domain    is in the Domain Name System
```

Here's **dig**'s usage summary:

```
dig   @<server>  domain   query-type
```

Here, <***server***> is the name of the DNS server you want to query, ***domain*** is the domain name you are interested in querying, and ***query-type*** is the name of the record you are trying to get (A, MX, NS, SOA, HINFO, TXT, ANY, and so on).

For example, to get the MX record for the example.org domain we established in the earlier project from the DNS server we set up, you would issue the **dig** command like this:

```
[master@serverA ~]$ dig @localhost example.org MX
```

To query our local DNS server for the A records for the yahoo.com domain, simply type the following:

```
[master@serverA ~]$ dig @localhost yahoo.com
```

NOTE Notice that for the preceding command, we didn't specify the query type—that is, we didn't explicitly specify an A-type record. The default behavior for **dig** is to assume you want an A-type record when nothing is specified explicitly. You might also notice that we are querying our DNS server for the yahoo.com domain. Our local server is obviously not authoritative for the yahoo.com domain, but because we also configured it as a caching-capable DNS server, it is able to obtain the proper answer for us from the appropriate DNS servers.

To query our local IPv6-capable DNS server for the AAAA record for the host serverB-v6 .example.org, type the following:

```
[master@serverA ~]$ dig  @localhost serverB-v6.example.org  -t AAAA
```

To reissue one of the previous commands, but this time suppress all verbosity using one of **dig**'s options (**+short**), type this:

```
[master@serverA ~]$ dig +short @localhost yahoo.com
```

To query the local name server for the reverse lookup information (PTR RR) for 192.168.1.1, type this:

```
[master@serverA ~]$ dig -x 192.168.1.1 @localhost
```

To query the local name server for the IPv6 reverse lookup information (PTR RR) for 2001:db8::2, type this:

```
[master@serverA ~]$ dig -x 2001:db8::2  @localhost
```

The **dig** program is incredibly powerful. Its options are too numerous to cover properly here. Read the man page installed with **dig** to learn how to use some of its advanced features.

resolvectl

resolvectl is another **systemd**-esque utility that's designed to play especially well with the **systemd-resolved** resolver service. It is used for resolving domain names, IPv4 and IPv6 addresses, DNS resource records (RR), and services (**openpgp**, **tlsa**, and so on) and for reconfiguring the DNS resolver. Its usage is simple and mimics the behavior of most of the other traditional DNS introspection toolkits. Here are the syntax and some common options:

```
resolvectl [OPTIONS...] {COMMAND}
Where possible Commands are:
 query HOSTNAME|ADDRESS  Resolve domain names, IPv4 and IPv6 addresses
 service [[NAME] TYPE] DOMAIN Resolve service (SRV)
 openpgp EMAIL@DOMAIN...      Query OpenPGP public key
 tlsa DOMAIN[:PORT]...        Query TLS public key
 status [LINK...]             Show link and server status
 statistics                   Show resolver statistics
 dns [LINK [SERVER...]]       Get/set per-interface DNS server address
 domain [LINK [DOMAIN...]]    Get/set per-interface search domain
 llmnr [LINK [MODE]]          Get/set per-interface LLMNR mode
```

Start the **systemd-resolved** service by running the following:

```
[master@serverA ~]$ sudo systemctl   start  systemd-resolved.service
```

Let's use **resolvectl** to get the MX record for the wikipedia.org domain:

```
[master@serverA ~]$ resolvectl  -t MX query wikipedia.org
```

Use **resolvectl** to do a reverse lookup of the 1.0.0.1 IP address:

```
[master@serverA ~]$ resolvectl  query 1.0.0.1
```

Retrieve the TLS key stored as a DNS record for the fedoraproject.org site:

```
[master@serverA ~]$ resolvectl tlsa tcp fedoraproject.org:443
```

nslookup

The **nslookup** utility is one of the tools that exists across various operating system platforms, so it is probably one of the most familiar tools for many. Its usage is simple, too. It can be used both interactively and noninteractively (that is, directly from the command line).

Interactive mode is entered when no arguments are provided to the command. Typing **nslookup** by itself at the command line will drop you to the **nslookup** shell. To get out of interactive mode, just type **exit** at the **nslookup** prompt.

Usage for the noninteractive mode is summarized here:

```
nslookup [ -option ] [ name | - ] [ server ]
```

For example, to use **nslookup** noninteractively to query our local name server for information about the host www.example.org, you'd type this:

```
[master@serverA ~]$ nslookup www.example.org localhost
```

NOTE The developers of BIND frown upon the use of the **nslookup** utility. Even though it is still very popular, it has officially been deprecated.

whois

The **whois** command is used for determining ownership of a domain. Information about a domain's owner isn't a mandatory part of its records, nor is it customarily placed in the TXT or RP records. Therefore, you'll need to gather this information using the **whois** technique, which reports the domain owner's actual owner, snail-mail address, e-mail address, and technical contact phone numbers. Let's try an example to get information about the example.com domain:

```
[master@serverA ~]$ whois example.com
Querying whois.verisign-grs.com]
...<OUTPUT TRUNCATED>...
% This query returned 1 object
domain:       EXAMPLE.COM
organisation: Internet Assigned Numbers Authority
created:      1992-01-01
```

nsupdate

An often-forgotten but powerful DNS utility is **nsupdate**. It is used to submit Dynamic DNS (DDNS) Update requests to a DNS server. It allows the resource records (RR) to be added or removed from a zone without your needing to edit the zone database files manually. This is especially useful because DDNS-type zones should not be edited or updated by hand since the manual changes are bound to conflict with the dynamic updates that are automatically maintained in journal files, which can result in zone data being corrupt.

The **nsupdate** program reads input from a specially formatted file or from standard input. Here's the syntax for the command:

```
nsupdate [ -d ] [[ -y keyname:secret ] [ -k keyfile ] ] [-v] [filename ]
```

The rndc Tool

The *remote name daemon control* utility is handy for controlling the name server and also debugging problems with the name server.

The **rndc** program can be used to manage the name server securely. A separate configuration file is required for **rndc**, because all communication with the server is authenticated with digital signatures that rely on a shared secret, which is typically stored in a configuration file named **/etc/rndc.conf**. You will need to generate the secret that is shared between the utility and the name server by using tools such as **rndc-confgen** (we don't discuss this feature here).

Following is the usage summary for **rndc**:

```
rndc [-c config] [-s server] [-p port] [-k key-file ]
[-y key] [-V] command
command is one of the following:
  reload       Reload configuration file and zones.
  reload zone [class [view]] Reload a single zone.
  refresh zone [class [view]] Schedule immediate maintenance for a zone.
  reconfig     Reload configuration file and new zones only.
  stats        Write server statistics to the statistics file.
  stop         Save pending updates to master files and stop the server.
  flush        Flushes all of the server's caches.
  status       Display status of the server.
```

You can use **rndc**, for example, to view the status of the DNS server:

```
[master@serverA ~]$ sudo rndc status
```

If, for example, you make changes to the zone database file (**/var/named/example.org.db**) for one of the zones under your control (such as example.org) and you want to reload just that zone without restarting the entire DNS server, you can issue the **rndc** command with the option shown here:

```
[master@serverA ~]$ sudo rndc reload example.org
```

CAUTION Remember to increment the serial number of the zone after making any changes to it!

Configuring DNS Clients

In this section, we'll delve into the wild and exciting process of configuring DNS clients! Okay, maybe it's not that exciting—but there's no denying the clients' significance to the infrastructure of any networked site.

The Resolver

So far, we've been studying servers and the DNS tree as a whole. The other part of this equation is, of course, the client—the host that's contacting the DNS server to resolve a hostname into an IP address.

NOTE You might have noticed earlier in the section "The DNS Toolbox" that most of the queries issued were being made against a DNS server called localhost. Localhost is, of course, the local system whose shell you are executing the query commands from. In our case, hopefully, this system is serverA.example.org! The reason we specified the DNS server to use was that, by default, the system will query whatever the host's default DNS server is. If it so happens that your host's DNS server is some random DNS server that your ISP has assigned you, some of the queries will fail, because your ISP's DNS server will not know about the zone you manage and control locally. Therefore, if we configure our local system to use our local DNS server to process all DNS-type queries, we won't have to specify "localhost" manually any longer.

Under Linux, the *resolver* handles the client side of DNS. This is actually part of a library of C programming functions that gets *linked* to a program when the program is started. Because all of this happens automatically and transparently, the user doesn't have to know anything about it. It's simply a little bit of magic that helps users, for example, browse the Internet—and remain blissfully unaware of how all the plumbing works!

From the system administrator's perspective, configuring the DNS client isn't magic, but it's straightforward. Traditionally, only two files are involved: **/etc/resolv.conf** and **/etc/nsswitch.conf**. We'll review the roles of these files next.

The /etc/resolv.conf File

The **/etc/resolv.conf** file traditionally used to be managed by the **`resolvconf`** program. The file and its behavior on modern Linux distros is now managed by a Network Name Resolution manager aptly named **systemd-resolved**.

The file contains the information necessary for the client to know what its local DNS server is. This file typically has two lines: the first indicates the default search domain, and the second indicates the IP address of the host's name server.

The default search domain applies mostly to sites that have their own local servers. When the default search domain is specified, the client side will automatically append this domain name to the requested site and check that first. For example, if you specify your default domain to be "yahoo.com" and then try to connect to the hostname "my," the client software will automatically try contacting "my.yahoo.com." Using the same default, if you try to contact the host "www.stat.net," the software will try "www.stat.net.yahoo.com" (a perfectly legal hostname), find that it doesn't exist, and then try "www.stat.net" alone (which does exist).

Of course, you can supply multiple default domains. However, doing so will slow the query process a bit, because each domain will need to be checked. For instance, if both example.org and stanford.edu are specified, and you perform a query on www.stat.net, you'll get three queries: www.stat.net.yahoo.com, www.stat.net.stanford.edu, and www.stat.net.

The format of the **/etc/resolv.conf** file is as follows:

```
search domainname
nameserver IP-address
```

Here, ***`domainname`*** is the default domain name to search, and ***`IP-address`*** is the IP address of your DNS server. For example, here's a sample **/etc/resolv.conf** file:

```
search example.org
nameserver 127.0.0.1
```

Thus, when a name lookup query is needed for serverB.example.org, only the host part is needed—that is, serverB. The example.org suffix will be automatically appended to the query. Of course, this is valid only at your local site, where you have control over how clients are configured!

The /etc/nsswitch.conf File

The **/etc/nsswitch.conf** file tells the system where it should look up certain kinds of configuration information (*services*). When multiple locations are identified, the **/etc/nsswitch.conf** file also specifies the order in which the information can best be found. The password, group, and hosts files/databases are sample back-ends that use **/etc/nsswitch.conf**.

The format of the **/etc/nsswitch.conf** file is simple. The service name comes first on a line (note that **/etc/nsswitch.conf** applies to more than just hostname lookups), followed by a colon. Next are the locations that contain the information. If multiple locations are identified, the entries are listed in the order in which the system needs to perform the search. Valid entries for locations are **`files`**, **`nis`**, **`dns`**, [**`NOTFOUND`**], and **`NISPLUS`**. Comments begin with a pound symbol (#).

For example, if you open the file for viewing, you might see a line similar to this:

```
hosts:      files nis dns myhostname
```

This line tells the system that all hostname lookups should first start with the **/etc/hosts** file. If the entry cannot be found there, NIS is checked. If the host cannot be found via NIS, DNS is checked, and then finally fall back to the *myhostname* **systemd** library (a mechanism for providing hostname resolution for locally configured system hostnames).

It's possible that NIS isn't running at your site and you want the system to check DNS records before it checks NIS records. In this case, you'd change and reorder the line to this:

```
hosts:     files dns myhostname nis
```

The only recommendation for this line is that the hosts file (**files**) should always come first in the lookup order.

And that's it. Save your file, and the system automatically detects the change.

Using [NOTFOUND=action]

In the **/etc/nsswitch.conf** file, you might see entries similar to **`[NOTFOUND=action]`**. This is a special directive that allows you to stop the process of searching for information after the system has failed all prior entries. The action can be either to return or continue. The default action is to continue. For example, if your file contains the line

```
hosts: files [NOTFOUND=return] dns nis
```

the system will try to look up host information in the **/etc/hosts** file only. If the requested information isn't found there, NIS and DNS won't be searched.

Configuring the Client (Traditional)

Let's walk through the traditional process of configuring a Linux client to use a DNS server. It's traditional because it doesn't rely on **systemd-resolved** to provide DNS resolver services and instead uses **`resolvconf`** (even if only in compatibility mode). We'll assume that we are using the DNS server on serverA and we are configuring serverA itself to be the client. This may sound a bit odd at first, but it is important for you to understand that just because a system runs the server does not mean it cannot also be a client! Think of it in terms of running a web server—just because a system runs Apache doesn't mean you can't run Firefox on the same machine and access the web sites hosted locally on the machine via the loopback address (127.0.0.1)!

Breaking out the steps to configuring the client, we'll do the following:

1. Let's first make sure that the main **/etc/resolv.conf** is not a link to the **systemd-resolved**-managed **/run/systemd/resolve/resolv.conf** by deleting the link:

   ```
   [master@serverA ~]$ sudo rm  /etc/resolv.conf
   ```

2. Edit or create **/etc/resolv.conf** and set the **nameserver** entry to point to your DNS server, per our example:

   ```
   search example.org
   nameserver 127.0.0.1
   ```

3. Look through the **/etc/nsswitch.conf** file to make sure that DNS is consulted for hostname resolutions:

   ```
   [master@serverA ~]$ grep "^hosts"  /etc/nsswitch.conf
   ```

   If you don't have **dns** listed, as in this output, use any text editor to include **dns** on the hosts line.

4. Test the configuration with the **`dig`** utility:

   ```
   [master@serverA ~]$ dig +short serverA.example.org
   192.168.1.1
   ```

Notice that you didn't have to specify explicitly the name server to use with **`dig`** (such as **`dig @localhost +short serverA.example.org`**) for the preceding query. This is because **`dig`** will by default use (that is, query) the DNS server specified in the local **/etc/resolv.conf** file.

TIP If you want to use **systemd-resolved** to provide resolver services, but still customize what it does, you can provide custom settings (DNS, search domain, FallbackDNS, LLMNR, and so on) in **/etc/systemd/resolved.conf** and then run **`systemctl restart systemd-resolved`** to restart the service.

Summary

This chapter covered all the information you'll need to get a basic DNS server infrastructure up and running. We discussed the following topics:

- Name resolution over the Internet
- Obtaining and installing the BIND name server and role of the **/etc/hosts** file

- Configuring DNS servers to act as primary, secondary, and caching servers
- Various DNS record types for IPv4 and IPv6
- Configuration options in the **named.conf** file
- Tools for use in conjunction with the DNS server to do troubleshooting
- Configuring a Linux client to use DNS

With the information available in the BIND documentation on how the server should be configured, along with the actual configuration files for a complete server presented in this chapter, you should be able to go out and perform a complete installation from start to finish.

Like any software, nothing is perfect, and problems can occur with BIND and the related files and programs discussed here. Don't forget to check out the main BIND web site (www.isc.org) as well as the various mailing lists dedicated to DNS and the BIND software for additional information.

This page intentionally left blank

CHAPTER

18 File Transfer Protocol (FTP)

The File Transfer Protocol (FTP) has been around since 1971. Remarkably, the underlying protocol itself has undergone little change since then. Clients and servers, on the other hand, have been almost constantly improved and refined. This chapter covers the *Very Secure FTP Daemon* (**vsftpd**) software package, which implements the FTP service.

The **vsftpd** program is a popular FTP server implementation and is used by major high-traffic FTP sites such as kernel.org, redhat.com, isc.org, and freebsd.org. The fact that these sites run the software adds to its "street cred." **vsftpd** was designed from the ground up to be fast, stable, and very secure.

NOTE Like most other services, **vsftpd** is only as secure as you make it. The authors of the program have provided all of the necessary tools to make the software as secure as possible out of the box, but a bad configuration can cause your site to become vulnerable. Remember to triple-check your configuration and test it out before going live.

This chapter will discuss how to obtain, install, and configure the latest version of **vsftpd**. We will show you how to configure it for private access as well as anonymous access. Also, you'll learn how to use an FTP client to test out your new FTP server.

The Mechanics of FTP

This section steps through the details of the FTP client/server interaction. Although this information isn't crucial to your being able to get an FTP server up and running, it is important when you need to troubleshoot FTP issues—especially issues that don't clearly manifest themselves as being related to FTP. These details might help you pinpoint issues such as whether the problem is with the network, the FTP server, or the FTP client.

Client/Server Interactions

The original design of FTP assumed something that was reasonable for a long time on the Internet: Internet users are a friendly, happy-go-lucky, do-no-evil bunch. This original design

and assumption require some rethinking given the more hostile Internet environment that we have today—an environment that necessitates the use of firewalls as a standard on networks.

Inasmuch as FTP facilitates the exchange of files between an FTP client and an FTP server, its design has some built-in nuances that are worthy of further mention.

One of its nuances stems from the fact that it uses two ports: a *control* port (port 21) and a *data* port (port 20). The control port serves as a communication channel between the client and the server for the exchange of commands and replies, and the data port is used purely for the exchange of data, which can be a file, part of a file, or a directory listing.

FTP can operate in two modes: *active FTP* mode and *passive FTP* mode.

Active FTP

Active **FTP** was traditionally used in the original FTP specifications. In this mode, the client connects from an ephemeral port (a port number greater than 1024) to the FTP server's command port (port 21). When the client is ready to transfer data, the server opens a connection from its data port (port 20) to the IP address and ephemeral port combination provided by the client. The key here is that the client does not make the actual data connection to the server but instead informs the server of its own port by issuing the **`PORT`** command; the server then connects back to the specified port. The server can be regarded as the active party (or the agitator) in this **FTP** mode.

From the perspective of an FTP client that is behind a firewall, the active **FTP** mode poses a slight problem: the firewall on the client side might frown upon (or disallow) connections originating or initiated from the Internet from a privileged service port (such as data port 20) to non-privileged service ports on the clients it is supposed to protect.

Passive FTP

The FTP client issues the **`PASV`** command to indicate that it wants to access data in passive mode, and the server then responds with an IP address and an ephemeral port number on itself to which the client can connect to transfer the data. The **`PASV`** command issued by the client tells the server to "listen" on a data port that is not its normal data port (that is, port 20) and to wait for a connection rather than initiate one. The key difference in comparison with active **FTP** mode is that in passive **FTP** mode, it is the client that initiates the connection to the port and IP address provided by the server. In this regard, the server can be considered the passive party in the data communication.

From the perspective of an FTP server that is behind a firewall, passive **FTP** mode is a little problematic because a firewall's natural instinct would be to disallow connections that originate from the Internet that are destined for ephemeral ports of the systems that it is supposed to protect. A typical symptom of this behavior occurs when a client appears to be able to connect to the server without a problem, but the connection seems to hang whenever an attempt to transfer data occurs.

To address some of the issues pertaining to FTP and firewalls, many firewalls implement application-level proxies for FTP that keep track of **FTP** requests and open up those high ports when needed to receive data from a remote site.

Obtaining and Installing vsftpd

The **vsftpd** package is the FTP server software that ships with most modern Linux distributions. The **vsftpd** project's official web site is https://security.appspot.com/vsftpd.html. You can easily install **vsftpd** directly from the distro software repository. In this section and the next, you will learn how to install/configure the software from the prepackaged binary. Let's start with the process of installing **vsftpd** on a Red Hat–like distro using **dnf**:

1. As a privileged user, use **dnf** to download and install **vsftpd** simultaneously:

```
[master@fedora-server ~]$ sudo dnf -y install vsftpd
```

2. Confirm that the software has been installed:

```
[master@fedora-server ~]$ rpm -q vsftpd
vsftpd-*
```

On a Debian-based distribution such as Ubuntu, **vsftpd** can be installed by typing the following:

```
master@ubuntu-server:~$ sudo apt-get -y install vsftpd
```

Configuring vsftpd

After we have installed the software, our next step is to configure it for use. When we installed the **vsftpd** software, we also installed other files and directories on the local file system. Some of the more important files and directories that come installed with the **vsftpd** program are discussed in Table 18-1.

File	Description
/usr/sbin/vsftpd	This is the main **vsftpd** executable. It is the daemon itself.
/etc/vsftpd/vsftpd.conf	This is the main configuration file for the **vsftpd** daemon. It contains the many directives that control the behavior of the FTP server.
/etc/vsftpd/ftpusers	Text file that stores the list of users *not* allowed to log into the FTP server. This file is referenced by the Pluggable Authentication Module (PAM) system.
/etc/vsftpd/user_list	Text file used either to allow or deny access to the users listed. Access is denied or allowed according to the value of the **userlist_deny** directive in the **vsftpd.conf** file.
/var/ftp	This is the FTP server's working directory.
/var/ftp/pub	This serves as the directory that holds files meant for anonymous access to the FTP server.

Table 18-1 vsftpd Configuration Files and Directories

The vsftpd.conf Configuration File

As stated, the main configuration file for the **vsftpd** FTP server is **vsftpd.conf**. Red Hat–like distros will usually place this file in the **/etc/vsftpd/** directory. On Debian-like systems, the configuration file is instead located at **/etc/vsftpd.conf**.

The file is quite easy to manage and understand, and it contains pairs of options (directives) and values that are in this simple format:

```
option=value
```

CAUTION The options and values in **vsftpd.conf** have finicky syntax! No space(s) should appear between the **option** directive, the equal sign (**=**), and the ***value***. Having any spaces therein can prevent the **vsftpd** daemon from starting up!

As with most other Linux/UNIX configuration files, comments in the file are denoted by lines that begin with the pound sign (#). To see the meaning of each of the directives, you should consult the **vsftpd.conf** man page, using the **man** command like so:

```
[master@server ~]$ man vsftpd.conf
```

NOTE **vsftpd** configuration files are located directly under the **/etc** directory on Debian-like systems. For example, the equivalent of **/etc/vsftpd/ftpusers** in Fedora is located at **/etc/ftpusers** in Ubuntu.

The options (or directives) in the **/etc/vsftpd/vsftpd.conf** file can be categorized according to the role they play. Some of these categories are discussed in Table 18-2.

NOTE The possible values of the options in the configuration file can also be divided into three categories: the Boolean options (such as YES, NO), the Numeric options (such as 007, 700), and the String options (such as root, /etc/vsftpd.chroot_list, and so on).

Starting and Testing the FTP Server

Because it comes with some default settings that allow it to hit the ground running, the **vsftpd** daemon is pretty much ready to run out of the box.

Of course, we'll need to start the service. Once you've learned how to start the daemon, the rest of this section will walk you through testing the FTP server by connecting to it using an FTP client.

So let's start a sample authenticated FTP session, but first we'll start the FTP service.

TIP The **ftp** daemon is automatically started right after installing the software in Ubuntu via **apt-get**. So check to make sure it isn't already running before trying to start it again. You can examine the output of the command

"ps -aux | grep vsftpd" to check this.

Type of Option	Description	Example
Daemon	These options control the general behavior of the **vsftpd** daemon.	**listen** | **listen_ipv6** When enabled, **vsftpd** will run in stand-alone mode instead of running under a superdaemon such as **xinetd** or **inetd**. **vsftpd** itself will then take care of listening for and handling incoming connections. Default value is **NO**.
Socket	These are the networking and port-related options.	**listen_address** Specifies the IP address on which **vsftpd** listens for network connections. This option has no default value.
		listen_port The port that **vsftpd** will listen on for incoming FTP connections. The default value is **21**.
		pasv_enable Enables or disables the **PASV** method of obtaining a data connection. The default value is **YES**.
		port_enable Enables or disables the **PORT** method of obtaining a data connection. The default value is **YES**.
Security	These options directly control the granting or denial of access to the server—the options offer a built-in access-control mechanism to the FTP server.	**anonymous_enable** Controls whether anonymous logins are permitted. If this option is enabled, both the usernames *ftp* and *anonymous* are recognized as anonymous logins. The default value is **YES**.
		local_enable Controls whether local logins are permitted. If this option is enabled, normal user accounts in **/etc/passwd** can be used to log in. Default value is **NO**.
		userlist_enable Loads a list of usernames from the filename specified by the **userlist_file** directive when this option is enabled. If a user tries to log in using a name in this file, that user will be denied access before even being prompted for a password. The default value is **NO**.
		userlist_deny Examined if the **userlist_enable** option is active. When its value is set to **NO**, users will be denied login unless they are explicitly listed in the file specified by **userlist_file**. When login is denied, the denial is issued before the user is asked for a password; this helps prevent users from sending cleartext across the network. The default value is **YES**.
		userlist_file This option specifies the name of the file to be loaded when the **userlist_enable** option is active. Depending on the distro, the default value is either **vsftpd.user_list** or **user_list**.
		cmds_allowed Specifies a list of allowed FTP commands. However, the post-login commands are always allowed—**USER**, **PASS**, **QUIT**; other commands are rejected. Sample usage: `cmds_allowed=PASV, RETR` This option has no default value.

Table 18-2 Configuration Options for vsftpd (*continued*)

Type of Option	Description	Example
File-transfer	These options relate to file transfers to and from the FTP server.	**download_enable** If set to **NO**, all download requests will be denied permission. The default value is **YES**.
		write_enable Controls whether any FTP commands that change the file system are allowed. These commands are **STOR**, **DELE**, **RNFR**, **RNTO**, **MKD**, **RMD**, **APPE**, and **SITE**. The default is **NO**.
		chown_uploads This option has the effect of changing the ownership of all anonymously uploaded files to that of the user specified in the setting **chown_username**. The default value is **NO**.
		chown_username Specifies the name of the user who is given ownership of anonymously uploaded files. The default value is **root**.
Directory	These options control the behavior of the directories served by the FTP server.	**use_localtime** When enabled, **vsftpd** will display directory listings with the time in the local system time zone. The default behavior is to display the time in Greenwich Mean Time (GMT). The default value is **NO**.
		hide_ids All directory listings will show **ftp** as the user and group for all files when this option is enabled. The default value is **NO**.
		dirlist_enable Enables or disables the ability to perform directory listings. If this option is set to **NO**, a permission-denied error will be given when a directory listing is attempted. The default is **YES**.
Logging	Controls how and where **vsftpd** logs information.	**vsftpd_log_file** Specifies the main **vsftpd** log file. The default value is **/var/log/ vsftpd.log**.
		xferlog_enable Tells the software to keep a log of all file transfers as they occur. The default is **NO**.
		syslog_enable If this option is enabled, any log output that would have gone to **/var/log/vsftpd.log** goes to the system log instead. Logging is done under the File Transfer Protocol Daemon (FTPD) facility. The default value is **NO**.

Table 18-2 Configuration Options for vsftpd

1. On Linux distributions running **systemd**, you can start the **vsftpd** daemon using the **systemctl** command:

```
[master@server ~]$ sudo systemctl start vsftpd.service
```

2. Launch the command-line FTP client program (install the **ftp** client package for your distro if you don't already have it) and connect to the local FTP server as a local user:

```
[master@server ~]# ftp localhost
Connected to localhost (127.0.0.1).
```

```
220 (vsFTPd 3.*.*)
Name (localhost:master):
```

3. Enter the name of a local user that exists on your system when prompted—we'll use the user account *yyang* to start an authenticated FTP session. Type **yyang** at the Name prompt:

```
Name (localhost:root): yyang
331 Please specify the password.
```

4. Enter **yyang's** password (*19gan19*) when prompted for the password:

```
Password: **********
230 Login successful.
Remote system type is UNIX.
```

5. Use the **ls** (or **dir**) FTP command to perform a listing of the files in the current directory on the FTP server:

```
ftp> ls
```

6. Use the **pwd** command to display our present working directory on the FTP server:

```
ftp> pwd
257 "/home/yyang" is the current directory
```

7. Using the **cd** command, try to change to a directory outside of yyang's FTP home directory; for example, change directory to the **/boot** directory of the local file system:

```
ftp> cd /boot
250 Directory successfully changed.
```

8. Log out of the FTP server using the **bye** FTP command:

```
ftp> bye
221 Goodbye.
```

The default **vsftpd** configuration on our sample system does the following things:

- **Local user logins** Permits all valid users on the local system with entries in the user database (the **/etc/passwd** file) to log into the FTP server using their normal usernames and passwords. This was demonstrated in the previous sample FTP session.
- **Anonymous FTP access** Prevents anonymous users from using or connecting to the FTP service.

Customizing the FTP Server

The default, out-of-the-box behavior of **vsftpd** might not be what you want for your production FTP server, so in this section we will walk through the process of customizing some of the FTP server's options to suit certain scenarios.

Setting Up an Anonymous-Only FTP Server

First we'll set up our FTP server so that it does *not* allow access to users who have regular user accounts on the system. This type of FTP server is useful for large sites that have files that should be available to the general public via FTP. In such a scenario, it is impractical to create an account for every single user when users can potentially number into the thousands.

Fortunately for us, **vsftpd** is almost ready to serve as an anonymous FTP server out of the box. We'll examine the configuration in the **vsftpd.conf** file and enable any options needed to ensure this and also disable the options that are not required.

With any text editor of your choice, open up the **/etc/vsftpd/vsftpd.conf** file for editing. Look through the file and make sure that, at a minimum, the directives listed next are present. (If the directives are present but commented out, you might need to remove the comment symbol [#] or change the value of the option.)

```
listen=NO
listen_ipv6=YES
xferlog_enable=YES
xferlog_std_format=YES
anonymous_enable=YES
local_enable=NO
write_enable=NO
```

You will find that these options are sufficient to enable your anonymous-only FTP server, so you may choose to overwrite the existing **/etc/vsftpd/vsftpd.conf** file (or **/etc/vsftpd.conf** on Ubuntu/Debian) and enter just the options shown. This will help keep the configuration file simple and uncluttered.

TIP Virtually all Linux systems come preconfigured with a user called "ftp." This account is supposed to be a non-privileged system account and is used especially for anonymous FTP-type access. You will need this account to exist on your system in order for anonymous FTP to work. To confirm the account exists, use the **getent** utility by typing the following:

```
# getent passwd ftp
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
```

If you don't get output similar to this, you can quickly create an FTP system account with the **useradd** command. To create a suitable ftp user, type this:

```
# useradd -c "FTP User" -d /var/ftp -r -s /sbin/nologin ftp
```

If you had to make any modifications to the **vsftpd.conf** file, you need to restart the **vsftpd** service. To restart **vsftpd on** distros using **systemd** as the service manager, type the following:

```
[master@server ~]$ sudo systemctl  restart vsftpd.service
```

You should test the server to make sure that only anonymous connections are allowed using the new configuration:

1. Launch the command-line FTP client program and connect to the local FTP server as an anonymous user:

```
[master@server ~]$ ftp localhost
```

2. Enter the name of the anonymous FTP user when prompted—that is, type **ftp**:

```
Name (localhost:master): ftp
331 Please specify the password.
```

TIP Most FTP servers that allow anonymous logins often also permit the implicit use of the username "anonymous." So instead of supplying **ftp** as the username to use to connect anonymously to our sample Fedora FTP server, we can instead use the popular username **anonymous**.

3. Enter anything at all when prompted for the password:

```
Password: <type-absolutely-anything-you-want-for-the-password>
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
```

4. Use the **pwd** command to display our present working directory on the FTP server:

```
ftp> pwd
257 "/"
```

5. Using the **cd** command, try to change to a directory outside of the allowed anonymous FTP directory (this should fail). For example, try to change the directory to the **/boot** directory of the local file system:

```
ftp> cd  /boot
550 Failed to change directory.
```

6. Log out of the FTP server using the **bye** FTP command:

```
ftp> bye
221 Goodbye.
```

That's it. We're done.

Setting Up an FTP Server with Virtual Users

Virtual users are users that do not actually exist—that is, these users do not have any privileges or functions on the system other than those for which they were created. This type of FTP setup serves as a midway point between enabling users with local system accounts to access the FTP server and enabling only anonymous users. If there is no way to guarantee the security of the network connection from the user end (FTP client) to the server end (FTP server), it would be foolhardy to allow users with local system accounts to log into the FTP server. This is because the FTP transaction between both ends usually occurs in plain text. Of course, this is relevant only if the server contains any data of value to its owners!

The use of virtual users will allow a site to serve content that should be accessible to untrusted users but still make the FTP service accessible to the general public. In the event

that the credentials of the virtual user(s) ever become compromised, you can at least be assured that only minimal damage can occur.

TIP It is also possible to set up **vsftpd** to encrypt all the communication between itself and any FTP clients by using Transport Layer Security/Secure Sockets Layer (TLS/SSL). This is quite easy to set up, but the caveat is that the clients' FTP application must also support this sort of communication. If security is a serious concern, you might consider using OpenSSH's **sftp** program instead for simple file transfers.

In this section we will create two sample virtual users named *ftp-user1* and *ftp-user2*. These users will not exist in any form in the system's user database (the **/etc/passwd** file). The following steps detail the process to achieve this:

TIP To save you some typing, copies of the configuration and/or script files used in this chapter are bundled at http://linuxserverexperts.com/8e/chapter-18.tar.xz. To follow along using this bundle, just download the file, extract its contents (decompress it), and copy the file(s) to the proper directories on your local system.

1. Create a plain-text file that contains the username and password combinations of the virtual users. Each username with its associated password will be on alternating lines in the file. For example, for the user **ftp-user1**, the password will be **user1**, and for the user **ftp-user2**, the password will be **user2**.

   Name the file **plain_vsftpd.txt**. Use a text editor of your choice to create the file. Here we use **vi**:

   ```
   [master@server ~]$ vi plain_vsftpd.txt
   ```

2. Enter this text into the file:

   ```
   ftp-user1
   user1
   ftp-user2
   user2
   ```

3. Save the changes to the file and then exit the text editor.

4. Convert the plain-text file that was created in Step 1 into a Berkeley DB format (db) that can be used with the **pam_userdb.so** library. The output will be saved in a file called **hash_vsftpd.db** stored under the /etc directory. Type the following:

   ```
   [master@server ~]$ sudo db_load -T -t hash -f plain_vsftpd.txt \
   /etc/hash_vsftpd.db
   ```

NOTE On Fedora systems, you need to have the **libdb-utils** package installed to have the **db_load** program. You can look for it on the installation media or quickly install it using **dnf** with the following command:

```
# dnf  -y install libdb-utils
```

The equivalent package in Ubuntu is called **db-util**.

5. Restrict access to the virtual users database file by giving it more restrictive permissions. This will ensure that it cannot be read by any casual user on the system. Type the following:

   ```
   [master@server ~]$ sudo chmod  600 /etc/hash_vsftpd.db
   ```

6. Next, create a PAM file that the FTP service will use as the new virtual users database file. We'll name the file **virtual-ftp** and save it under the **/etc/pam.d/** directory. Use any text editor to create the file.

```
[master@server ~]$ sudo vi /etc/pam.d/virtual-ftp
```

7. Enter this text into the file:

```
auth required /lib64/security/pam_userdb.so db=/etc/hash_vsftpd
account required /lib64/security/pam_userdb.so db=/etc/hash_vsftpd
```

   These entries tell the PAM system to authenticate users using the new database stored in the **hash_vsftpd.db** file.

8. Make sure that the changes have been saved into a file named **virtual-ftp** under the **/etc/pam.d/** directory.

9. Let's create a home environment for our virtual FTP users. We'll cheat and use the existing directory structure of the FTP server to create a subfolder that will store the files that we want the virtual users to be able to access. Type the following:

```
[master@server ~]$ sudo mkdir  -p  /var/ftp/private
```

TIP We cheated in Step 9 so that we won't have to go through the process of creating a guest FTP user that the virtual users will eventually map to, and also to avoid having to worry about permission issues, since the system already has an FTP system account that we can safely leverage. Look for the **guest_username** directive under the **vsftpd.conf** man page for further information (**`man vsftp.conf`**).

10. Now we'll create our custom **vsftpd.conf** file that will enable the entire setup.

    Open the **/etc/vsftpd/vsftpd.conf** file for editing. Look through the file and make sure that, at a minimum, the directives listed next are present. (If the directives are present but commented out, you may need to remove the comment sign or change the value of the option.) Comments have been added to explain the less-obvious directives.

```
listen=NO
listen_ipv6=YES
#We do NOT want to allow users to log in anonymously
anonymous_enable=NO
xferlog_enable=YES
xferlog_std_format=YES
connect_from_port_20=YES
seccomp_sandbox=NO
#This is for the PAM service that
# we created that was named virtual-ftp
pam_service_name=virtual-ftp
#Enable the use of the /etc/vsftpd.user_list file
userlist_enable=YES
#Do NOT deny access to users specified
# in the /etc/vsftpd.user_list file
userlist_deny=NO
userlist_file=/etc/vsftpd.user_list
local_enable=YES
#This activates virtual users.
```

```
guest_enable=YES
#Map all the virtual users to the real user called "ftp"
guest_username=ftp
#Make all virtual users root ftp directory on the server to be:
#/var/ftp/private/
local_root=/var/ftp/private/
```

TIP If you choose not to edit the existing configuration file and instead create one from scratch, you will find that the options specified here will serve your purposes with nothing additional needed. The **vsftpd** software will simply assume its built-in defaults for any option that you didn't specify in the configuration file!

11. We'll need to create (or edit) the **/etc/vsftpd.user_list** file that was referenced in the configuration in Step 10. To create the entry for the first virtual user, type this:

```
[master@server ~]$ echo ftp-user1  | \
 sudo tee -a  /etc/vsftpd.user_list
```

12. To create the entry for the second virtual user, type this:

```
[master@server ~]$ echo ftp-user2 | \
sudo tee -a  /etc/vsftpd.user_list
```

13. We are ready to fire up or restart the FTP server now. Type the following:

```
[master@server ~]$ sudo systemctl  restart  vsftpd.service
```

14. We will now verify that the FTP server is behaving the way we want it to by connecting to it as one of the virtual FTP users and running some simple FTP commands. Connect to the server as the user **ftp-user1** (remember that the FTP password for that user is **user1**).

```
[master@server ~]$ ftp localhost
...<OUTPUT TRUNCATED>...
Name (localhost:master): ftp-user1
331 Please specify the password.
Password:
230 Login successful.
...<OUTPUT TRUNCATED>...
ftp> ls -l
...<OUTPUT TRUNCATED>...
ftp> pwd
257 "/"
ftp> cd /boot
550 Failed to change directory.
ftp> bye
221 Goodbye.
```

15. We'll also test to make sure that anonymous users *cannot* log into the server:

```
[master@server ~]$ ftp localhost
...<OUTPUT TRUNCATED>...
Name (localhost:master): ftp
530 Permission denied.
Login failed.
```

16. We'll finally verify that local users (for example, the user yyang) *cannot* log into the server:

```
[master@server ~]$ ftp localhost
...<OUTPUT TRUNCATED>...
Name (localhost:master): yyang
530 Permission denied.
Login failed.
```

Everything looks fine.

TIP **vsftpd** is an IPv6-ready daemon. Enabling the FTP server to listen on an IPv6 interface is as simple as enabling the proper options in the **vsftpd** configuration file. However, note that the directives **listen** (for IPv4) and **listen_ipv6** (for IPv6) are mutually exclusive and cannot both be set to **YES** in the same configuration file. Therefore, one way to support both IPv4 and IPv6 is to set **listen_ipv6** and its value to **YES** and set **listen=NO**, like so:

```
listen=NO
listen_ipv6=YES
```

The other alternative to configure the **vsftpd** software to support IPv4 and IPv6 simultaneously is to spawn another instance of **vsftpd** and point it to its own config file to support the protocol version you want.

On Fedora and other Red Hat–type distros, the **vsftpd** startup scripts will automatically read (and start) all files under the **/etc/vsftpd/** directory that end with *.conf. So, for example, you can name one file **/etc/vsftpd/vsftpd.conf** and name the other file that supports IPv6 something like **/etc/vsftpd/vsftpd-ipv6.conf**.

Summary

The Very Secure FTP Daemon is a powerful FTP server offering all of the features you need for running a commercial-grade FTP server in a secure manner. In this chapter, we discussed the process of installing and configuring the **vsftpd** server on Fedora and Debian-like systems. Specifically, we covered the following information:

- Some important and often-used configuration options for **vsftpd**
- Details about the FTP protocol and its effects on firewalls
- How to set up anonymous FTP servers
- How to set up an FTP server that allows the use of virtual users
- How to use an FTP client to connect to the FTP server to test things out

This information is enough to keep your FTP server humming along for quite a while. Of course, like any printed media about software, this text will age and the information will slowly but surely become obsolete. Therefore, be sure to visit the **vsftpd** web site from time to time, not only to learn about the latest developments but also to obtain the latest documentation.

This page intentionally left blank

CHAPTER

19 Apache Web Server

Apache is a widely popular open source Hypertext Transfer Protocol (HTTP) server software. It is released under the Apache license. Apache continues to dominate the web server market in terms of use and deployments. It has thus maintained a certain level of acceptance and respect from the Internet community. Apache offers some of the following benefits and advantages:

- It is stable, flexible, and secure.
- It is used, backed, and supported by several major sites and organizations.
- The entire program and related components are open source.
- It works on most variants of Linux/UNIX and Microsoft Windows.

We'll walk through the process of installing and configuring the Apache HTTP server on a Linux server in this chapter. But before we get into the steps necessary to configure Apache, let's review some of the fundamentals of HTTP as well as some of the internals of Apache, such as its process ownership model. This information will help you understand why Apache is set up to work the way it does.

Understanding HTTP

HTTP traffic makes up a significant portion of the world's Internet traffic, and Apache is a server implementation of HTTP. Applications such as Firefox, Chrome, Opera, Curl, wget, Edge, Safari, and Internet Explorer are client implementations of HTTP.

As of this writing, HTTP is at version 2, but HTTP version 1.1 is still by far the most widely used version of the protocol out there. HTTP/1.1 is documented in RFCs 7230 through 7235.

Headers

When a web client connects to a web server, the client's default method of making this connection is to contact the server's TCP port 80. Once connected, the web server says nothing; it's up to the client to issue HTTP-compliant commands (also called *verbs* or *methods*) for its requests to the server. Along with each command comes a *request header* that includes information about the client. For example, when using the Firefox browser (the client) on a Linux box, a web server might receive the following information from the client:

```
GET / HTTP/1.1
Connection: Keep-Alive
User-Agent: Mozilla/* (X11;Linux x86_64;rv:*) Gecko/* Firefox/*
Host: localhost:80
Accept: text/xml, image/gif, image/jpeg, image/png...
Accept-Encoding: gzip,deflate
Accept-Language: en-us
Accept-Charset: iso-8859-1,*,utf-8
```

The first line contains the HTTP GET command, which asks the server to fetch a file. The remainder of the information makes up the header, which tells the server about the client, the kind of file formats the client will accept, and so forth. Many servers use this information to determine what can and cannot be sent to the client, as well as for logging purposes.

Along with the request header, additional headers may be sent. For example, when a client uses a hyperlink to get to the server site, an entry showing the client's originating address will also appear in the header.

When it receives a blank line, the server knows a request header is complete. Once the request header is received, it responds with the actual requested content, prefixed by a server header. The server header provides the client with information about the server, the amount of data the client is about to receive, the type of data coming in, and other information. For example, the request header just shown, when sent to an HTTP server, results in the following server response header:

```
HTTP/1.1 200 OK
Date: Sun, 07 Jun 2026 14:03:31 GMT
Server: Apache/2.4.* (Fedora)
Last-Modified: Sun, 07 Jun 2026 11:41:32 GMT
ETag: "7888-1f-b80bf300"
Accept-Ranges: bytes
Content-Length: 31
Connection: close
Content-Type: text/html; charset=UTF-8
```

A blank line and then the actual content of the transmission follow the response header.

Ports

The default port for HTTP requests is port 80, but you can also configure a web server to use a different (arbitrarily chosen) port that is not in use by another service. This is one of the mechanisms for running multiple web servers or sites on the same host, with each server or site on a different port. Some sites also use this arrangement for multiple configurations of their web servers to support various types of client requests.

When a site runs a web server on a nonstandard port, you can see that port number in the site's URL. For example, the web address www.example.com with the default port number (80) implicitly and explicitly displayed would read http://www.example.com and http://www.example.com:80, respectively. But serving the same site on a nonstandard port such as port 8080 will require the port number to be explicitly stated, as in http://www.example.com:8080.

Process Ownership and Security

Running a web server on a Linux platform follows the traditional Linux/UNIX permissions and ownership model. In terms of permissions, that means each process has an owner, and that owner has limited rights on the system.

Whenever a program (process) is started, it inherits the permissions of its parent process. For example, if you're logged in as root, the shell in which you're doing all your work has all the same rights as the root user. In addition, any process you start from this shell will inherit all of your permissions. Processes may give up rights, but they cannot gain rights.

NOTE There is an exception to the Linux inheritance principle. Programs configured with the SetUID bit do not inherit rights from their parent process, but rather start with the rights specified by the owner of the file itself. For example, the file containing the program `su` (**/bin/su**) is owned by root and has the SetUID bit set. If the user *yyang* runs the program `su`, that program doesn't inherit the rights of yyang but instead will start with the rights of the superuser (root). To learn more about SetUID, see Chapter 6.

How Apache Processes Ownership

To carry out initial network-related functions, the Apache HTTP server must start with root permissions. Specifically, it needs to bind itself to port 80 so that it can listen for requests and accept connections. Once it does this, Apache can give up its rights and run as a non-root user (unprivileged user), as specified in its configuration files. Different Linux distributions may have varying defaults for this user, but it is usually one of the following: *nobody*, *www, apache, wwwrun, www-data*, or *daemon*.

Remember that when running as an unprivileged user, Apache can read only the files that the user has permissions to read.

Security is especially important for sites that use executable scripts such as Common Gateway Interface (CGI), PHP, or Python scripts. By limiting the permissions of the web server, you decrease the likelihood that someone can send a malicious and executable request to the server. The server processes and corresponding scripts can damage only what they can access. As the user *nobody,* the scripts and processes don't have access to the same key files that the *root* user can access. (Remember that, by default, root can access everything, no matter what the permissions.)

Installing the Apache HTTP Server

Most Linux distributions have the Apache HTTP server software prepackaged as RPM, .deb, or other binaries, so installing the software is usually as simple as using the package management tool on the system. This section walks you through the process of obtaining and installing the program via RPM and Advanced Packaging Tool (APT). Mention is also made of installing the software from source code, if you choose to go that route. The actual configuration of the server covered in later sections applies to both classes of installation (from source or from a binary package).

On Fedora/RHEL/CentOS systems, the package that provides the Apache HTTP server is *semi-intuitively* named **httpd-*.rpm**. We'll use a package manager (**dnf** or **yum**) on our sample Fedora server to obtain and install the program.

To use **dnf** to install the program, type the following:

```
[master@fedora-server ~]$ sudo dnf -y install httpd
```

To confirm that the software is installed, type the following:

```
[master@fedora-server ~]$ sudo rpm -q httpd
```

For a Debian-based Linux distribution such as Ubuntu, the package that provides the Apache HTTP server (version 2) is more intuitively named "apache2." You can use APT to install Apache by running the following:

```
master@ubuntu-server:~$ sudo apt-get -y install apache2
```

The web server daemon is automatically started after you install using **apt-get** on Ubuntu systems.

And that's it! You now have Apache installed.

Apache Modules

Part of what makes Apache so powerful and flexible is that its design allows for it to be extended through modules. Apache comes with many modules by default and automatically includes them in the default installation.

If you can imagine "it," you can be almost certain that somebody has already written a module for it for the Apache web server. The Apache module application programming interface (API) is well documented, and if you are so inclined (and know how), you can probably write your own module for Apache to provide any functionality you want.

To give you some idea of what kinds of things people are doing with modules, visit http://modules.apache.org. There you will find information on how to extend Apache's capabilities using modules. Here are some common Apache modules:

- **mod_wsgi** Provides a Web Server Gateway Interface (WSGI)–compliant interface for hosting Python-based web applications
- **mod_authnz_ldap** Provides support for authenticating users of the Apache HTTP server against a Lightweight Directory Access Protocol (LDAP) database

- **mod_ssl** Provides strong cryptography for the Apache web server via the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols
- **mod_userdir** Allows user content to be served from user-specific directories on the web server via HTTP
- **mod_proxy** Implements an extensible proxy/gateway/load-balancing interface for Apache when used in conjunction with some other modules (such as **mod_proxy_ftp**, **mod_proxy_balancer**, **mod_proxy_http**, and so on)

If you know the name of a particular module you want (and if the module is popular enough), you might find that the module has already been packaged in an RPM format, so you can install it using the usual RPM methods. For example, if you want to include the SSL module (**mod_ssl**) in your web server setup, on a Fedora/RHEL/CentOS system, you can issue this **dnf** command to download and install the module for you automatically:

```
[master@fedora-server ~]$ sudo dnf install mod_ssl
```

Alternatively, you can go to the Apache modules project web site and search for, download, compile, and install the module that you want.

TIP Make sure the run-as user is there! If you build Apache from source, the sample configuration file (**httpd.conf**) expects that the web server will run as the user named *daemon*. Although that user exists on almost all Linux distributions, you may want to double-check the local user database (**/etc/passwd**) to make sure the user daemon does indeed exist.

Starting Up and Shutting Down Apache

Starting up and shutting down Apache on most Linux distributions is easy.

To start Apache on any distro that refers to Apache as **httpd** and also uses the **service** utility, use this command:

```
[master@server ~]$ sudo service httpd start
```

On modern Linux distributions running **systemd**, you can alternatively start the **httpd** daemon using the **systemctl** command like so:

```
[master@fedora-server ~]$ sudo systemctl start httpd
```

Debian-like systems like Ubuntu refer to the Apache binary as **apache2**, so you would instead start Apache on such distros by running this:

```
master@ubuntu-server:~$ sudo systemctl  start  apache2
```

To shut down Apache on an RPM-based distro like Fedora, enter this command:

```
[master@server ~]$ systemctl stop httpd.service
```

On Ubuntu or Debian, you should instead run the following command:

```
master@ubuntu-server:~$ sudo systemctl stop apache2
```

After making a configuration change to the web server that requires you to restart Apache on an RPM-based distro like Fedora, type this:

```
[master@server ~]$ sudo systemctl restart httpd
```

TIP On a system running openSUSE or SLE (SUSE Linux Enterprise), the commands to start and stop the web server, respectively, are

```
[opensuse-server ~]# rcapache2 start
```

and

```
[opensuse-server ~]# rcapache2 stop
```

Starting Apache at Boot Time

After installing the web server, it's reasonable to assume that you want the web service to be available at all times to your users; you will therefore need to configure the system to automatically start the service between system reboots. Use the **`chkconfig`** utility on legacy Linux distros to configure the automatic startup of the web server service (for example, via **chkconfig httpd on**). Change `httpd` to `apache2` on Debian-like systems in the following examples.

On **systemd**-enabled Linux distros, you can check if **httpd** (or apache2) is enabled for automatic startup by running this command:

```
[master@fedora-server ~]$ systemctl list-unit-files httpd
```

If the previous output shows that the unit file for the service is disabled, you can make the **httpd** (or apache2) daemon automatically start up with system reboots by issuing the **`systemctl`** command, like so:

```
[master@fedora-server ~]$ sudo systemctl enable httpd
```

In older/legacy Ubuntu distros, you can use either the **`sysv-rc-conf`** or **`update-rc.d`** utility to manage the runlevels in which Apache starts up, like so:

```
master@ubuntu-server:~$ sudo update-rc.d apache2  defaults
```

Testing Your Installation

You can perform a quick test of your Apache installation by trying to browse or visit the default basic web site or home page that's often bundled with Apache for testing purposes.

Use the status option with the **`systemctl`** command on **systemd**-aware systems to view a nice synopsis (cgroup information, child processes, and so on) of the Apache server status, like so:

```
[master@fedora-server ~]$ systemctl  status  httpd.service
```

On our sample Fedora system, Apache comes with a default page that gets served to visitors in the absence of a custom default home page (for example, **index.html** or **index.htm**). The file displayed to visitors when there is no default home page is **/usr/share/httpd/noindex/index.html** and is controlled by the **/etc/httpd/conf.d/welcome.conf** configuration file.

To find out if your Apache installation went smoothly, start a web browser and point it to the web site on your machine. To do this from the same system running the web server, simply type **http://localhost** (or the IPv6 equivalent, **http://[::1]/**) in the address bar of your web browser. You should see a simple Demo/Sample page showing the web server is working. If you don't see this, retrace your Apache installation steps and make sure you didn't encounter any errors in the process. Another thing to check if you can't see the default web page is to make sure you don't have any host-based firewall such as Netfilter (**iptables**, **nftables**, or **ufw**; see Chapter 14) blocking access to the web server.

To quickly open up the HTTP port on a Fedora server, type the following:

```
[master@fedora-server ~]$ sudo firewall-cmd  --permanent \
 --add-service=http
[master@fedora-server ~]$ sudo firewall-cmd --reload
```

Configuring Apache

Apache supports a rich set of configuration options that are sensible and easy to follow. This makes it a simple task to set up the web server in various configurations.

This section walks through a basic configuration. The default configuration is actually quite good and often works right out of the box, so if the default is acceptable to you, simply start creating your HTML documents! Apache allows several common customizations. After we step through creating a simple web page, you'll see how to make those common customizations in the Apache configuration files.

Creating a Simple Root-Level Page

If you like, you can start adding files to Apache right away in the **/var/www/html** directory for top-level pages. Just remember to make sure that any files or directories placed in that directory are world-readable.

As mentioned earlier, Apache's default web page is **index.html**. Let's create and change the default home page so that it reads "Welcome to webserver.example.org." Here are the commands:

```
$ cd /var/www/html/
$ echo  "Welcome to webserver.example.org"| sudo tee -a  index.html
$ sudo chmod  644 index.html
```

You could also use an editor such as **vi**, **pico**, or **emacs** to edit the **index.html** file and make it more interesting.

Apache Configuration Files

The configuration files for Apache are located in the **/etc/httpd/conf/** directory on RPM-based distros like Fedora. The main configuration file is named **httpd.conf** on such systems.

On Debian-like systems, the main configuration file for Apache is instead named **/etc/apache2/apache2.conf**.

One good method to familiarize yourself with and learn more about the configuration files is to read the **httpd.conf** file. The default configuration file is heavily commented, explaining each entry, its role, and the parameters you can set.

Common Configuration Options

The default configuration settings work just fine right out of the box, and for basic needs, they may require no further modification. Nevertheless, site administrators may need to customize their web server and/or web sites further.

This section discusses some of the common directives or options used in Apache's configuration file.

ServerRoot

This specifies the base configuration directory for the web server. On Fedora, RHEL, and CentOS distributions, this value, by default, is the **/etc/httpd/** directory. The default value for this directive in Ubuntu, openSUSE, and Debian Linux distributions is **/etc/apache2/**.

Syntax: `ServerRoot` *`directory-path`*

Listen

This is the port(s) on which the server listens for connection requests. It refers to the venerable port 80 (HTTP) for which everything good and bad on the web is so well known!

The **`Listen`** directive can also be used to specify the particular IP addresses over which the web server accepts connections. The default value for this directive is **80** for nonsecure web communications.

Syntax: `Listen [`*`IP-address:`*`] `*`portnumber`*

For example, to set Apache to listen on all its IPv4 and IPv6 interfaces on port 80, you would set the **`Listen`** directive to read as follows:

```
Listen 80
```

To set Apache to listen on a specific IPv6 interface (such as 2002:c0a8:1::) on port 8080, you would set the **`Listen`** directive to read as follows:

```
Listen  [2002:c0a8:1::]:8080
```

For Debian-like systems such as Ubuntu, set this directive outside of the main configuration file. The value is usually set in the **/etc/apache2/ports.conf** file.

ServerName

This directive defines the hostname and port that the server uses to identify itself. At many sites, a single (normally underutilized) server may fulfill multiple purposes and host several other services. For example, an intranet web server that isn't getting heavy usage may also double up as an FTP server to serve the same files. In such a situation, a computer name such as "www" (with a corresponding fully qualified domain name [FQDN] of www.example.org) wouldn't be a good choice because it suggests that the machine has only one purpose.

It's better to give a server a neutral name and then establish Domain Name System (DNS) Canonical Name (CNAME) entries or multiple hostname entries in the **/etc/hosts** file. In other words, from the users' perspective, you can define several names for accessing the server or services.

Consider a server whose real hostname is dioxin.eng.example.org. This server also doubles as a web server. You might be thinking of giving it the hostname alias www.sales.example.org. However, since dioxin will know itself only as dioxin, users who visit www.sales.example.org might be confused by seeing in their browsers that the server's real name is dioxin.

Apache provides a way to get around this through the use of the **`ServerName`** directive. This works by allowing you to specify what you want Apache to return as the hostname of the web server to web clients or visitors.

Syntax: `ServerName` *`fully-qualified-domain-name`*`[:` *`port`*`]`

ServerAdmin

This is the e-mail address that the server includes in error messages sent to the client.

It's often a good idea, for a couple of reasons, to use an e-mail alias for a web site's administrator(s). First, there might be more than one administrator. By using an alias, it's possible for the alias to expand out to a list of other e-mail addresses. Second, it'll be easier and quicker to perform mass updates/edits to a bunch of web pages that might have the site administrator e-mail address (incorrectly) hard-coded in them. Here is the syntax:

Syntax: `ServerAdmin` *`e-mail_address`*

DocumentRoot

This defines the primary directory on the web server from which HTML files will be served to requesting clients. On Linux distros such as CentOS, RHEL, Fedora, Ubuntu, and Debian, the default value for this directive is **/var/www/html/**. On openSUSE and SLE distributions, the default value for this directive is **/srv/www/htdocs**.

Syntax: `DocumentRoot` *`directory-path`*

TIP On a web server that is expected to host plenty of web content, the file system pointed to by the **`DocumentRoot`** directive should have a lot of free space to house any current and future [anticipated] web content.

MaxRequestWorkers

This sets a limit on the number of simultaneous requests that the web server will service.

Syntax: `MaxRequestWorkers` *`number`*

LoadModule

This is used for loading or adding other modules into Apache's running configuration. It adds the specified module to the list of active modules.

Syntax: `LoadModule` *`module filename`*

Enabling and Disabling Apache Modules

Debian-based distros like Ubuntu have a handy set of utilities that can be used easily to enable or disable Apache modules that are already installed. You can list the currently installed modules under the **/usr/lib/apache2/modules/** directory. For example, to enable the **userdir** module, simply type this:

```
master@ubuntu-server:~$ sudo a2enmod  userdir
```

To disable the **userdir** module, you would use the sister command named **`a2dismod`**:

```
master@ubuntu-server:~$ sudo a2dismod  userdir
```

Running the **`a2enmod`** command will "auto-magically" create any needed symbol links under the **/etc/apache2/mods-enabled/** directory. For example, for the **userdir** module, the first link will be to the file **/etc/apache2/mods-available/userdir.conf** (which contains the actual configuration details for the **userdir** module), and the second will be a link to **/etc/apache2/mods-available/userdir.load**, which contains the actual **LoadModule** directive.

Finally, don't forget to reload or restart Apache after enabling or disabling a module. This can be done quickly, like so:

```
master@ubuntu-server:~$ sudo systemctl restart httpd
```

User

This specifies the user ID with which the web server will answer requests. The server process will initially start off as the root user but will later downgrade its privileges to those of the user specified here. The user should have only just enough privileges to access files and directories that are intended to be visible to the outside world via the web server. Also, the user should not be able to execute code that is not HTTP or web related.

On a Fedora system, the value for this directive is automatically set to the user named **apache**. In openSUSE Linux, the value is set to the user **wwwrun**. In a Debian-like system such as Ubuntu, the value is set to the user **www-data** (set via the **`$APACHE_RUN_USER`** environment variable).

Syntax: `User` *`unix_userid`*

Group

This specifies the group name of the Apache HTTP server process. It is the group with which the server will respond to requests. The default value under the Fedora, CentOS, and RHEL flavors of Linux is **apache**. In openSUSE Linux, the value is set to the group **www**. In Ubuntu, the default value is **www-data** (set via the **`$APACHE_RUN_GROUP`** environment variable).

Syntax: `Group` *`unix_group`*

Include

This directive allows Apache to specify and include other configuration files at runtime. It is mostly useful for organization purposes; you can, for example, elect to store all the configuration directives for different virtual domains in appropriately named files, and Apache will automatically know to include them at runtime.

Syntax: `Include` *`file_name_to_include_OR_path_to_directory_to_include`*

Many of the mainstream Linux distros rely quite heavily on the use of the **`Include`** directive to organize site-specific configuration files and directives for the web server. Often, this file and directory organization is the sole distinguishing factor between Apache installation/setup among the different distros.

UserDir

This directive defines the subdirectory name within each user's home directory, where users can place personal content that they want to make accessible via the web server. This directory is usually named **public_html** and is often stored under each user's home directory. This option is, of course, dependent on the availability of the **mod_userdir** module in the web server setup.

Here's a sample usage of this option in the **httpd.conf** file:

```
UserDir public_html
```

Quick How-To: Serving HTTP Content from User Directories

First enable/configure the **`UserDir`** option. Now, assuming the user *yyang* wants to make some web content available from within their home directory via the web server, following these steps will make this happen:

1. While logged into the system as the user *yyang*, create the **public_html** folder:

   ```
   yyang@server:~$ mkdir ~/public_html
   ```

2. Set the proper permissions for the parent folder:

   ```
   yyang@server:~$ chmod a+x  .
   ```

(*continued*)

3. Set the proper permissions for the **public_html** folder:

```
yyang@server :~$ chmod a+x public_html
```

4. Create a sample page named **index.html** under the **public_html** folder:

```
yyang@server :~$ echo "Ying Yang's Home Page" >> \
 ~/public_html/index.html
```

As a result of these commands, files placed in the **public_html** directory for a particular user and set to world-readable will be on the Web via the web server.

To access the contents of that folder via HTTP, you would need to point a web browser to the URL

```
http://<YOUR_HOST_NAME>/~<USERNAME>
```

where <YOUR_HOST_NAME> is the web server's FQDN or IP address. And if you are sitting directly on the web server itself, you can simply replace that variable with **localhost**.

For the example shown here for the user yyang, the exact URL will be **http://localhost/~yyang**, and the IPv6 equivalent is **http://[::1]/~yyang**.

Note that on older versions of Red Hat–like distros with the SELinux subsystem enabled, you may have to do a little more to get the **`UserDir`** directive working. This is because of the default security contexts of the files stored under each user's home directory. By default, the context is **user_home_t**. For this functionality to work properly, you will have to change the context of all files under **~/username/public_html/** to **httpd_sys_content_t**. This allows Apache to read the files under the **public_html** directory. Here is the command to do this:

```
[yyang@fedora-server ~]$ chcon -Rt  httpd_sys_content_t public_html/
```

ErrorLog

This defines the location where errors from the web server will be logged.

Syntax: `ErrorLog` *`file_path`*`|` *`syslog`*`[:` *`facility`*`]`

Example: ErrorLog /var/log/httpd/error_log

LogLevel

This option sets the level of verbosity for the messages sent to the error logs. Acceptable log levels are **`emerg`**, **`alert`**, **`crit`**, **`error`**, **`warn`**, **`notice`**, **`info`**, and **`debug`**. The default log level is **`warn`**.

Syntax: `LogLevel` *`level`*

Alias

The **Alias** directive allows documents (web content) to be stored in any other location on the file system that is different from the location specified by the **DocumentRoot** directive. It also allows you to create abbreviations (or aliases) for path names that might otherwise be quite long.

Syntax: `Alias` *`URL_path actual_file_or_directory_path`*

ScriptAlias

The **ScriptAlias** option specifies a target directory or file as containing CGI scripts that are meant to be processed by the CGI module (**mod_cgi**).

Syntax: `ScriptAlias` *`URL-path actual_file_path_OR_directory_path`*

Example: ScriptAlias /cgi-bin/ "/var/www/cgi-bin/"

VirtualHost

One of the most-used features of Apache is its ability to support virtual hosts. This makes it possible for a single web server to host multiple web sites as if each site had its own dedicated hardware. It works by allowing the web server to provide different, autonomous content, based on the hostname, port number, or IP address that is being requested by the client. This is accomplished by the HTTP protocol, which specifies the desired site in the HTTP header rather than relying on the server to learn what site to fetch from its IP address.

This directive is actually made up of two tags: an opening **<VirtualHost>** tag and a closing **</VirtualHost>** tag. It is used to specify the options that pertain to a particular virtual host. Most of the directives that we discussed previously are valid here, too.

Syntax:
```
<VirtualHost ip_address_OR_hostname[:port] >
      Options
< /VirtualHost >
```

Suppose, for example, that we wanted to set up a virtual host configuration for a host named www.another-example.org. To do this, we can create a **VirtualHost** entry in the **httpd.conf** file (or use the **Include** directive to specify a separate file), like this one:

```
<VirtualHost www.another-example.org>
      ServerAdmin webmaster@another-example.org
      DocumentRoot /www/docs/another-example.org
      ServerName www.another-example.org
      ErrorLog logs/another-example.org-error_log
</VirtualHost>
```

On Debian-like distros, you can use another set of utilities (**a2ensite** and **a2dissite**) to enable or disable virtual hosts and web sites quickly under Apache.

For example, assuming while on an Ubuntu server we created the previous configuration file named **www.another-example.org** for the virtual web site and stored the file under

the **/etc/apache2/sites-available/** directory, we can enable the virtual web site using the following command:

```
master@ubuntu-server:~$ sudo a2ensite www.another-example.org
```

Similarly, to disable the virtual site, we can run this command:

```
master@ubuntu-server:~$ sudo a2dissite www.another-example.org
```

After running any of the previous commands (**a2ensite** or **a2dissite**), you should make Apache reload its configuration files by running the following:

```
master@ubuntu-server:~$ sudo systemctl  reload  apache2
```

Finally, don't forget that it is not enough to configure a virtual host using Apache's **VirtualHost** directive—the value of the **ServerName** option in the **VirtualHost** container must be a name that is resolvable via DNS (or any other means) to the web server machine.

NOTE Apache's options/directives are too numerous to all be covered in this section. However, the software comes with its own extensive online manual, which is written in HTML so that you can access it in a browser. If you installed the software via RPM, you might find that documentation for Apache has been packaged into a separate RPM binary, and as a result, you will need to install the proper package (for example, **httpd-manual**) to have access to it. If you downloaded and built the software from source code, you will find the documentation in the **manual** directory of your installation prefix (for example, **/usr/local/httpd/manual**). Depending on the Apache version, the documentation is available online at the project's web site at http://httpd.apache.org/docs/.

Troubleshooting Apache

The process of changing various configuration options (or even the initial installation) sometimes may not work as smoothly as you'd like. Thankfully—and with other things being equal—Apache does an excellent job at reporting in its error log file why it failed or what is failing.

The error log file is located in your **logs** directory. If you are running a stock Fedora or RHEL-type installation, this is in the **/var/log/httpd/** directory. If you are running Apache on a stock Debian- or Ubuntu-type distro, this is in the **/var/log/apache2/** directory.

The **access_log** file is simply that—a log of which files have been accessed by people visiting your web site(s). It contains information about whether the transfer completed successfully, where the request originated (IP address), how much data was transferred, and what time the transfer occurred. This is a powerful way of determining the usage of your site.

The **error_log** file contains all the errors that occur in Apache. Note that not all errors that occur are fatal—some are simply problems with a client connection from which Apache can automatically recover and continue operation. However, if you started Apache but still cannot visit your web site, take a look at this log file to see why Apache might not be responding. The easiest way to see the most recent error messages is by using the **tail** command, like so:

```
[master@fedora-server html]$ sudo tail -n 10 /var/log/httpd/error_log
```

If you need to see more log information than that, simply change the number 10 to the number of lines you need to see. If you would like to view the errors or logs in real time as they are being generated, you can use the **-f** option for the **tail** command. This provides a valuable debugging tool, because you can try things out with the server (such as requesting web pages or restarting Apache) and view the results of your experiments in a separate virtual terminal window. The **tail** command with the **-f** switch is shown here:

```
[master@fedora-server html]$ sudo tail -f /var/log/httpd/error_log
```

This command will constantly tail the logs until you terminate the program.

On a Fedora/CentOS/RHEL system running the **systemd-journald** service, you can alternatively use the **journalctl** utility to view the latest messages from the **httpd.service** unit by running the following:

```
[master@fedora-server ~]$ journalctl  -u httpd.service
```

TIP When SELinux is enabled, watch out for its unhelpful interference when troubleshooting your web server on Red Hat–like distros. You might find **httpd** not working as you'd expect and, even more alarming, with *no* corresponding logs to help you troubleshoot! You might need to temporarily disable SELinux in this case.

On Debian-based distros like Ubuntu running the **systemd-journald** service, you can use the **journalctl** utility to view the latest messages from the **apache2.service** unit, like so:

```
[master@ubuntu-server ~]$ journalctl  -u apache2.service
```

Summary

This chapter covered the process of setting up your own web server using Apache (aka **httpd**) from the ground up. This chapter by itself is enough to get you going with a top-level page and a basic configuration. At a minimum, the material covered here will help you get your web server on the interwebs, or internets—whichever one you prefer!

It is highly recommended that you take some time to page through the relevant and official Apache manual/documentation (http://httpd.apache.org/docs/). It is well written, concise, and flexible enough that you can set up just about any configuration imaginable.

This page intentionally left blank

CHAPTER

20 Simple Mail Transfer Protocol (SMTP)

The Simple Mail Transfer Protocol (SMTP) is the de facto standard for transporting mail around the Internet. Anyone who wants to have a mail server capable of sending and receiving mail must be able to support SMTP. As a standards-based protocol (see RFC 5321), SMTP is well understood, platform independent, and well supported across a variety of operating systems and devices.

In this chapter, we'll discuss the mechanics of SMTP as a protocol and its relationship to other mail-related protocols, such as Post Office Protocol (POP) and Internet Message Access Protocol (IMAP). Then we will go over the Postfix SMTP server, one of the easier and more secure SMTP servers out there.

Understanding SMTP

SMTP defines the method by which mail is sent from one host to another. That's it. It does not define how the mail should be stored. It does not define how the mail should be displayed to the recipient.

SMTP's strength is its simplicity, and that is due, in part, to the dynamic nature of networks during the early 1980s (circa when the protocol was invented). Back in those days, people were linking networks together with everything short of bubble gum and glue. SMTP was the first mail standard that was independent of the transport mechanism. This meant people using TCP/IP networks could use the same format to send a message as someone using two cans and a string—at least theoretically.

SMTP is also independent of operating systems, which means each system can use its own style of storing mail without worrying about how the sender of a message stores her mail. You can draw parallels to how the phone system works: Each phone service provider has its own independent accounting system. However, they all have agreed upon a standard way to link their networks together so that calls can go from one network to another transparently.

In the Free Open Source Software (FOSS) world, several software packages (such as Exim, Postfix, Sendmail, and opensmtpd) provide their own implementation of SMTP.

Rudimentary SMTP Details

Ever had a "friend" who sent you an e-mail on behalf of some government agency informing you that you owe taxes from the previous year, plus additional penalties? We're going to show you how they did it and, what's even more fun, how you can do it yourself. (Not that we would advocate such behavior, of course.)

The purpose of this example is to show how SMTP sends a message from one host to another. After all, more important than learning how to forge an e-mail is learning how to troubleshoot mail-related problems. So in this example you are acting as the sending host, and whichever machine you connect to is the receiving host.

SMTP requires only that a host be able to send straight ASCII text to another host. Typically, this is done by contacting the SMTP port (port 25) on a mail server. You can do this using the Telnet program. Here's an example:

```
[master@server ~]$ telnet mailserver 25
```

Here, the host ***mailserver*** is the recipient's fictitious mail server. The **25** that follows ***mailserver*** tells Telnet that you want to communicate with the server's port 25 (standard SMTP port) rather than the normal standard Telnet port 23.

The mail server will respond with a greeting message such as this:

```
220 mail ESMTP Postfix
```

You are now communicating directly with the SMTP server.

Although there are many SMTP commands, four are worth noting:

HELO, "MAIL FROM:", :, "RCPT TO:", :, and **"DATA"**

The **HELO** command is used when a client introduces itself to the server. The parameter to **HELO** is the hostname that is originating the connection. Of course, most mail servers take this information with a grain of salt and double-check it themselves. Here's an example:

```
HELO example.org
```

If you aren't coming from the example.org domain, many mail servers will respond by telling you that they know your real IP address, but they may or may not stop the connection from continuing.

The **MAIL FROM:** command requires the sender's e-mail address as its argument. This tells the mail server the e-mail's origin. Here's an example:

```
MAIL FROM: ass-kisser@example.org
```

This means the message is from ass-kisser@example.org.

The **RCPT TO:** command requires the receiver's e-mail address as an argument. Here's an example:

```
RCPT TO: manager@example.org
```

This means the message is destined to manager@example.org.

Now that the server knows who the sender and recipient are, it needs to know what message to send. This is done by using the **DATA** command. Once it's issued, the server will expect the entire message, with relevant header information, followed by one empty line, a period, and then another empty line. Continuing the example, ass-kisser@example.org might want to send the following message to manager@example.org:

```
DATA
354 End data with <CR><LF>.<CR><LF>
Just an FYI, boss. The project is not only on time, but it is within
budget, too! What are my next priorities?
Love Always -
The Devoted Employee
.
250 2.0.0 Ok: queued as B9E3B3C0D
```

And that's all there is to it. To close the connection, enter the **QUIT** command.

This is the basic technique used by applications that send mail—except, of course, that all the gory details are masked behind a nice GUI application. The underlying transaction between the client and the server remains mostly the same.

Security Implications

Developers of the Postfix mail server implementation wrote the server software from scratch with security in mind. Basically, the package ships in a tight security mode, and it's up to the individual user to loosen it up as much as is needed for a specific environment. This means the responsibility falls to us (as sysadmins) for making sure we keep the software properly configured (and thus not vulnerable to attacks).

When deploying any mail server, keep the following issues and questions in mind:

- When an e-mail is sent to the server, what programs will it trigger? And under what permissions do those programs run?
- Are those programs securely designed? And can you protect/secure the communication channels between the programs and the end clients via secure protocols?
- If the communication channels cannot be made secure, how can you limit the damage in case of an attack?

E-mail Components

Mail service has three distinct components:

- **Mail user agent (MUA)** The component of the e-mail system that the user sees and interacts with, such as the Thunderbird, Outlook, Evolution, or Mutt program. An MUA is responsible only for reading mail and allowing users to compose mail.
- **Mail transport agent (MTA)** Handles the process of getting the mail from one site to another. Postfix, Exim, and Sendmail are popular examples of an MTA.

- **Mail delivery agent (MDA)** Responsible for distributing and sorting any received messages on the local machine to the appropriate user mailbox. The Procmail program is a popular solution for handling the actual mail delivery (MDA) component of e-mail. This is because of its advanced filtering mechanism, as well as its secure design from the ground up.

NOTE Some mail systems integrate all three components. For example, Microsoft Exchange Server integrates the MTA and MDA functionalities into a single system. Postfix, on the other hand, works as an MTA only, passing the task of performing local mail delivery to another external program. This delineation of tasks allows the use of other tools or solutions for tasks such as determining mailbox storage mechanisms.

Installing the Postfix Server

We chose the Postfix mail server in this discussion for its ease of use, simple design, and secure track record. (The author of Postfix also argues that the simplicity has led to improved security.) Postfix provides most of the functionalities that Sendmail program does—in fact, the typical installation procedure for Postfix is to work as a drop-in replacement for Sendmail binaries completely.

Postfix is the default mail server program on most modern Linux distros. In the following sections, we show you how to install Postfix using the built-in package management (Red Hat's RPM or Debian's dpkg) mechanism of the distribution. This is the recommended method. We also show how to build and install the software from its source code.

Installing Postfix via DNF in Fedora, CentOS, or RHEL

To install Postfix via DNF on Fedora, CentOS, or RHEL distros, simply use the **dnf** package manager as follows:

```
[master@fedora-server ~]$ sudo dnf -y install postfix
```

Once the command runs to completion, you should have Postfix installed.

On older legacy Linux distros, you can use the **chkconfig** utility to make sure that your Postfix mail service starts automatically during system boot.

On modern systemd-enabled distros, use the **systemctl** command like so:

```
[master@fedora-server ~]$ sudo systemctl enable postfix.service
```

Finally, you can flip the switch and actually start the Postfix process. With a default configuration, it won't do much, but it will confirm whether the installation worked as expected.

On **systemd**-enabled distros, start the postfix service unit via the following:

```
[master@fedora-server ~]$ sudo systemctl start postfix.service
```

TIP You may find yourself inheriting or managing an existing Red Hat–based distro that has both of the popular MTAs (Sendmail and Postfix) installed, or you may decide to install a new MTA. If you want to switch the mail subsystem that is running on such a system—from, say, Sendmail to Postfix—you can use the **`alternatives`** facility to switch the default MTA provider. Run the command like so and follow the prompts:

```
# alternatives --config mta
```

Installing Postfix via APT in Ubuntu

Postfix can be installed in Ubuntu by using Advanced Packaging Tool (APT). Ubuntu does not ship with any MTA software preconfigured and running. You explicitly need to install and set one up. To install the Postfix MTA in Ubuntu, run this command:

```
master@ubuntu-server:~$ sudo apt-get -y install postfix
```

You will be prompted to select your Postfix mail server configuration type during the installation process. Here are the available types:

- **No configuration** This option will leave the current configuration unchanged.
- **Internet site** Mail is sent and received directly using SMTP.
- **Internet with smarthost** Mail is received directly using SMTP or by running a utility such as fetchmail. Outgoing mail is sent using a smarthost.
- **Satellite system** All mail is sent to another machine, called a smarthost, for delivery.
- **Local only** The only delivered mail is the mail for local users. The system does not need any sort of network connectivity for this option.

We will use the first option, **No configuration**, on our sample Ubuntu server. The install process will also create the necessary user and group accounts that Postfix needs.

Configuring the Postfix Server

By following the preceding tracks, you have installed the Postfix mail system using your distro's package manager. After installing the Postfix software, you will need to configure it. Most of its configuration files can be found under the **/etc/postfix/** directory.

You configure the server through the **/etc/postfix/main.cf** configuration file. It's obvious from its name that this configuration file is the main configuration file for Postfix! The other configuration file of note is the **master.cf** file. This is the process configuration file for Postfix, which allows you to change how Postfix processes are run. This can be useful, for example, for setting up Postfix on clients so that it doesn't accept e-mail and forwards to a central mail hub. (For more information on doing this, see the documentation at www.postfix.org.)

Now let's move on to the **main.cf** configuration file.

The main.cf File

The **main.cf** file is too large to list all of its options in this chapter, but we will cover the most important options that will get your mail server up and running. Thankfully, the configuration file is well documented and clearly explains each option and its function.

The sample options discussed next are enough to help you get a basic Postfix mail server up and running at a minimum.

myhostname

This parameter is used for specifying the hostname of the mail system. It sets the Internet hostname for which Postfix will be receiving e-mail. The default format for the hostname is to use the fully qualified domain name (FQDN) of the host. Typical examples of mail server hostnames are mail.example.com, smtp.example.org, mx1.example.org, and so on. Here's the syntax:

```
myhostname = server.example.org
```

mydomain

This parameter is the mail domain you will be servicing, such as example.com, spamrus.net, linuxserverexperts.com, or google.com. Here's the syntax:

```
mydomain = example.org
```

myorigin

All e-mail sent from this e-mail server will look as though it came from this parameter. You can set this to either **$myhostname** or **$mydomain**, like so:

```
myorigin = $mydomain
```

Notice that you can use the value of other parameters in the configuration file by placing a **$** sign in front of the variable name.

mydestination

This parameter lists the domains that the Postfix server will take as its final destination for incoming e-mail. Typically, this value is set to the hostname of the server and the domain name, but it can contain other names, as shown here:

```
mydestination = $myhostname, localhost.$mydomain, $mydomain, \
mail.$mydomain, www.$mydomain, ftp.$mydomain
```

If your server has more than one name (for example, server.example.org and serverA.another-example.org), you will want to make sure you list both names here.

mail_spool_directory

You can run the Postfix server in two modes of delivery: directly to a user's mailbox or to a central spool directory. The typical way is to store the mail in **/var/spool/mail**. The variable will look like this in the configuration file:

```
mail_spool_directory = /var/spool/mail
```

The result is that mail will be stored for each user under the **/var/spool/mail** directory, with each user's mailbox represented as a file. For example, e-mail sent to yyang@example.org will be stored in **/var/spool/mail/yyang**.

mynetworks

The **mynetworks** variable is an important configuration option. This lets you configure what servers can relay through your Postfix server. You will usually want to allow relaying from local client machines and nothing else. Otherwise, spammers can use your mail server to relay messages. Here's an example value of this variable:

```
mynetworks = 192.168.1.0/24, 127.0.0.0/8
```

If you define this parameter, it will override the **mynetworks_style** parameter. The **mynetworks_style** parameter allows you to specify any of the keywords **class**, **subnet**, and **host**. These settings tell the server to trust these networks to which the server belongs.

CAUTION If you do not set the **$mynetworks** variable correctly and spammers begin using your mail server as a relay, you might quickly find a surge of angry online mail administrators e-mailing you about it. Furthermore, it is a fast way to get your mail server blacklisted by one of the spam control techniques, such as a DNS Blacklist (DNSBL) or Realtime Blackhole List (RBL). Once your server is blacklisted, very few people will be able to receive mail from you, and you will need to jump through a lot of hoops to get unlisted. Even worse, no one will tell you that you have been blacklisted.

smtpd_banner

This variable allows you to return a custom response when a client connects to your mail server. It is a good idea to change the banner to something that doesn't give away what server you are using. This just adds one more slight hurdle for hackers trying to find faults in your specific software version.

```
smtpd_banner = $myhostname ESMTP
```

inet_protocols

This parameter specifies the network interface addresses that the mail system receives mail on. The default behavior is for the Postfix server software to make use of all active interfaces on the machine when accepting connections. Its default value is **all**. Setting this value to **ipv6** will make Postfix support IPv6. Here are some example values that this parameter accepts:

```
inet_protocols = ipv4
inet_protocols = ipv4, ipv6
inet_protocols = all (DEFAULT)
inet_protocols = ipv6
```

Tons of other parameters in the Postfix configuration file are not discussed here. You might see them commented out in the configuration file when you set the preceding options. These other options will allow you to set security levels and debugging levels, among other things, as required.

Now let's move on to running the Postfix mail system and maintaining *your* mail server.

Checking Your Configuration

Postfix includes a nice tool for checking a current configuration and helping you troubleshoot it. Simply run the following:

```
[master@server ~]$ sudo postfix check
```

This will list any errors that the Postfix system finds in the configuration files or with permissions of any directories that it needs. A quick run on our sample system shows this:

```
 master@server ~]$ sudo postfix check
postfix: fatal: /etc/postfix/main.cf, line 83-115:

 missing '=' after attribute
name: "mydomain example.org"
```

Looks like we made a typo in the configuration file!

When going back to fix any errors in the configuration file, you should be sure to read the error message carefully and use the line number as guidance, not as absolute. This is because a typo in the file could mean that Postfix detected the error well *after* the actual error took place.

In this example, an error of omission (forgetting the = symbol) that we made on line 83 in the configuration file was shown as occurring in lines 83 through lines 115 due to how the parsing engine works. However, by carefully reading the error message, we knew the problem was with the "mydomain" parameter, and so it took only a quick search before we found the real line culprit.

Let's run the check again:

```
[master@server ~]$ sudo postfix check
```

Groovy! No errors this time. We're ready to start using Postfix.

TIP You can use the nifty little **postconf** utility to quickly query or display the value of parameters in your Postfix configuration files. For example, to view the current value of the **`mydomain`** parameter, you can run the following:

```
$ postconf  mydomain
mydomain = localdomain
```

Running the Server

Controlling the Postfix mail server is easy and straightforward. On **systemd**-enabled distros, just pass the correct **`start/stop/reload`** option to the **`systemctl`** utility and specify the postfix service unit. To start Postfix, type the following:

```
[master@server ~]$ sudo systemctl start postfix
```

When you make any changes to the configuration files, you need to tell Postfix to reload itself to make the changes take effect. Do this by using the **`reload`** option:

```
[master@server ~]$ sudo systemctl reload postfix
```

Make sure that Postfix is configured to automatically start up between reboots by typing the following:

```
[master@server ~]$ sudo systemctl enable postfix
```

Checking the Mail Queue

Occasionally, the mail queues on your system will fill up. This can be caused by network failures or other various failures, such as other external mail servers. To check the mail queue on your mail server, simply type the following command:

```
[master@server ~]$  mailq
```

This command will display all of the messages that are in the Postfix mail queue. This is the first step in testing and verifying that the mail server is working correctly.

Flushing the Mail Queue

Sometimes after an outage, mail will be queued up, and it can take several hours for the messages to be sent. Use the **`postfix flush`** command to flush out any messages that are shown in the queue by the **`mailq`** command.

The newaliases Command

The **/etc/aliases** file contains a list of e-mail aliases. This is used to create site-wide e-mail lists and aliases for users. Whenever you make changes to the **/etc/aliases** file, you need to tell Postfix about it by running the **`newaliases`** command. This command will rebuild the Postfix databases and inform you of how many names have been added.

Making Sure Everything Works

Once the Postfix mail server is installed and configured, you should test and test again to make sure that everything is working correctly. The first step in doing this is to use a local mail user agent, such as **pine** (text-based, freeware), **mutt** (text-based, GNU GPL), or **mailx** (simple command-line MUA for UNIX systems), to send e-mail to yourself. If this works, great; you can move on to sending e-mail to a remote site, while keeping an eye on the output of the **`mailq`** command to see when the message gets sent. The final step is to make sure you can send e-mail to the server from outside networks (that is, from the Internet). If you can receive e-mail from the outside world, your work is done.

Mail Logs

On Fedora, RHEL, and CentOS systems, by default, mail logs go to **/var/log/maillog**, as defined by the **rsyslogd** configuration file. If you need to change this, you can modify the **rsyslogd** configuration file, **/etc/rsyslog.conf**, by editing the following line:

```
mail.*                  -/var/log/maillog
```

Most sites run their mail logs this way, so if you are having problems, you can search through the **/var/log/maillog** file for any relevant messages.

Debian-based systems, such as Ubuntu, store the mail-related logs in the **/var/log/mail.log** file.

openSUSE and SUSE Linux Enterprise (SLE) store their mail-related logs in the files **/var/log/mail**, **/var/log/mail.err**, **/var/log/mail.info**, and **/var/log/mail.warn**.

If Mail Still Won't Work

If mail still won't work, don't worry. SMTP isn't always easy to set up the first time. If you still have problems, walk logically through all of the steps and look for errors. The first step is to look at your log messages, which might show that other mail servers are not responding. If everything seems fine there, check your Domain Name System (DNS) settings. Can the mail server perform name lookups? Can it perform Mail Exchanger (MX) lookups? Can other people perform name lookups for your mail server? It is also possible that e-mails are actually being delivered but are being marked as junk or spam at the recipient end. If possible, ask the receiver to check the junk or spam mail folder at their end.

Proper troubleshooting techniques are indispensable for good system administration. A good resource for troubleshooting is to look at what others have done to fix similar problems. Check the Postfix web site at www.postfix.org, or search online, for the problems or symptoms of what you might be seeing.

Summary

In this chapter, you learned the basics of how SMTP works. You also installed Postfix and learned how to configure a basic Postfix mail server. With this information, you have enough knowledge to set up and run a minimal production mail server.

If you're looking for additional information on Postfix, start with the online documentation at www.postfix.org. The documentation is well written and easy to follow. It offers a wealth of information on how Postfix can be extended to perform a number of additional functions that are outside the scope of this chapter. Another excellent reference on the Postfix system is *The Book of Postfix: State-of-the-Art Message Transport*, by Ralf Hildebrandt and Patrick Koetter (No Starch Press, 2005). This book covers the Postfix system in excellent detail.

As with any other service, don't forget to keep up with the latest news on Postfix. Security updates do come out from time to time, and it is important that you update your mail server to reflect these changes.

CHAPTER

21 Post Office Protocol and Internet Mail Access Protocol (POP and IMAP)

In Chapter 20, we covered Simple Mail Transfer Protocol (SMTP), the underlying e-mail transport mechanism or protocol by which e-mail is sent from e-mail clients to the server and from one mail server to another mail server. We also mentioned how mail delivery agents (MDAs) facilitate the processing of incoming mail (such as sorting or filtering mail according to sender, subject line, length of message, keywords, and so on). All of this processing is usually done after the mail has been transported to the final destination mail server. For the MDA functionality, a program like Procmail can be used. Procmail can make copies of user e-mails available to users in the *mbox* format. The mbox format is a simple text format that can be read by a number of console mail user agents (MUAs) such as **pine**, **elm**, **mailx**, and **mutt**, as well as some GUI-based mail clients such as Thunderbird.

To make the mbox format usable, the e-mail client (MUA) needs to have direct access (at the file system level) to the mbox file itself. This works well enough in tightly administered environments where the administrator of the mail server is also the administrator of the client hosts; however, this system of mail folder administration might not scale well in certain scenarios. The following sample scenarios might prove to be a bit thorny:

- Users are unable to stay reasonably connected to a fast/secure network for file system access to their mbox file (for example, roaming laptops).
- Users need local copies of e-mail for offline viewing.
- Security requirements dictate that users not have direct access to the mail store; for example, Network File System (NFS) shared mail spool directories are considered unacceptable.
- MUA does not support the mbox format (typical of MS Windows-based e-mail clients).

To handle these thorny cases and others where Procmail and other traditional MDAs will simply not suffice, another class of protocols was created. We'll collectively describe this class of protocols as *mail access protocols*. This chapter covers two popular mail access protocols: Post Office Protocol (POP) and Internet Message Access Protocol (IMAP).

POP was created to allow for network-based access to mail stores. Many early Windows-based mail clients used POP for access to Internet e-mail, because it allowed users to access UNIX-based mail servers (the dominant type of mail server on the Internet until the rise of Microsoft Exchange in the late 1990s).

The idea behind POP is simple: A central mail server remains online at all times and can receive and store mail for all of its users. Mail that is received is queued on the server until a user connects via POP and downloads the queued mail. The mail on the server itself can be stored in any format (such as mbox) so long as it adheres to the POP protocol.

When a user wants to send an e-mail, the e-mail client relays it through the central mail server via SMTP. This allows the client the freedom to disconnect from the network after passing on its e-mail message to the server. The task/responsibility of forwarding the message, taking care of retransmissions, handling delays, and so on, is then left to the well-connected mail server. Figure 21-1 shows this relationship.

Certain aspects of the POP protocol are too limiting. Features such as being able to keep a master copy of a user's e-mail on the server with only a cached copy on the client were missing. This led to the development of IMAP.

The earliest RFC (request for comments) documenting the inner workings of IMAPv2 is RFC 1064, dated 1988. After IMAPv2 came IMAP version 4 (IMAPv4) in 1994. Most e-mail clients are compatible with IMAPv4. Some design deficiencies inherent in IMAPv4 led to

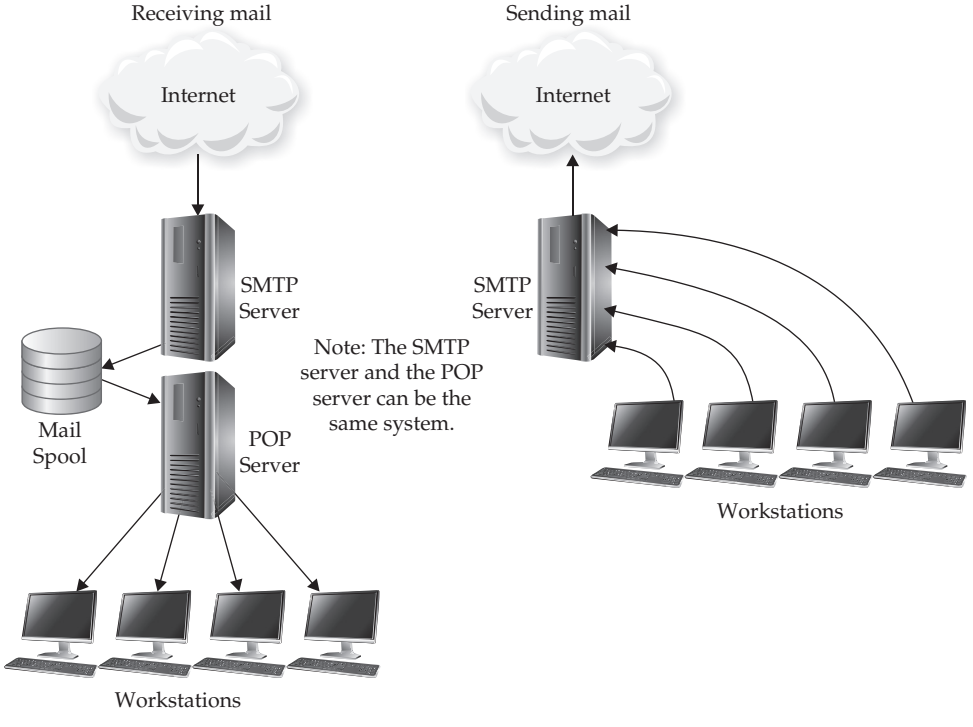


Figure 21-1 Sending and receiving mail with SMTP and POP

another update in the protocol specifications, and, thus, IMAPv4 is currently at its first revision—IMAP4rev1 (RFC 3501).

The evolution of IMAP can best be understood by thinking of mail access as working in one of three distinct modes: online, offline, and disconnected. The *online* mode is akin to having direct file system access to the mail store (for example, having read access to the **/var/mail** file system). The *offline* mode is how POP works, where the client is assumed to be disconnected from the network, except when explicitly pulling down its e-mail. In offline mode, the server normally does not retain a copy of the mail.

Disconnected mode works by allowing users to retain cached copies of their mail stores. When the client is connected, any incoming/outgoing e-mail is immediately recognized and synchronized; however, when the client is disconnected, changes made on the client are kept until reconnection, when synchronization occurs. Because the client retains only a cached copy, a user can move to a completely different client and re-synchronize his or her e-mail.

By using IMAP, your mail server will support all three modes of access. After all is said and done, deploying and supporting both POP and IMAP is usually a good idea. It allows users the freedom to choose whatever mail client, protocol, and workflow that best suits them.

There are several Free and Open Source Software (FOSS) mail servers that implement POP and IMAP. Some of them are Dovecot, University of Washington IMAP server (UW IMAP), Cyrus IMAP server, and Courier IMAP server. This chapter covers the installation and configuration of the popular Dovecot server software. This particular mail server has been available for many years. The installation process is also easy.

POP3 and IMAP Protocol Basics

Like the other services discussed so far, POP3 and IMAP each need a server process to handle requests. The POP3, POP3S, IMAP, and IMAPS server processes listen on ports 110, 995, 143, and 993, respectively.

Each request to and response from the server is in clear-text ASCII, which means it's easy for us to test the functionality of the server using Telnet. This is especially useful for quickly debugging mail server connectivity/availability issues. Like with an SMTP server, you can interact with a POP3 or IMAP server using a short list of commands. Although there are many POP commands, a couple worth mentioning are **`USER`** and **`PASS`**.

And a few noteworthy IMAP commands are

`LOGIN`, **`LIST`**, **`STATUS`**, **`EXAMINE`**/**`SELECT`**, **`CREATE`**/**`DELETE`**/**`RENAME`**, and **`LOGOUT`**.

Later in this chapter, we'll use some of these commands and walk through the connection and login process on a POP3 server and an IMAP server. This will allow us to verify that the server does in fact work.

Dovecot (IMAP and POP3 Server)

The Dovecot POP3 and IMAP server software is well regarded and is used in many production sites around the world. It is a well-tested implementation and readily available on most of the mainstream Linux distros.

Dovecot provides its IMAP and POP3 functions through the following main processes. Collectively, these processes provide various services that make up the Dovecot ecosystem.

- **Master process** As its name implies, the master process is the primary/overseer process. It is responsible for starting and keeping all the other processes running as needed. It reads the settings/options in the configuration files and exports the values to the other processes. The master process is responsible for collecting and managing all logging information that Dovecot generates. The master process runs under the **`dovecot`** executable.
- **Login processes** The login processes listen for connection requests for the POP3 and IMAP protocols and implement the minimum handshaking protocol requirement before a user logs in successfully. The login processes run under the **imap-login** and **pop3-login** executables.
- **Authentication (auth) process** Once the login processes complete the underlying POP3 or IMAP protocol handshaking and setup, control is passed on to the appropriate authentication process. The auth process is responsible for performing the actual user authentication (Simple Authentication and Security Layer [SASL] functions) to verify that the user is who she says she is. The authentication processes run under a similarly named **auth** executable.
- **Mail processes (IMAP, POP3)** After authentication is successfully completed, the desired mail process kicks in and provides the user access to her mailboxes. The mail process is the actual workhorse that implements the POP3 and IMAP protocol details. The IMAP process runs under an executable called **imap**, and the POP3 process runs under an executable named **pop3**.

Installing Dovecot

Most Linux distributions have prepackaged binaries for Dovecot in the distros' repositories. For example, Dovecot can be installed in Fedora/CentOS/RHEL by using **`dnf`** like so:

```
[master@fedora-server ~]$ sudo dnf -y install dovecot
```

On Debian-like systems, such as Ubuntu, Dovecot's IMAP and POP3 functionality is provided in two separate packages: dovecot-imapd and dovecot-pop3d, respectively. They can be installed by using Advanced Packaging Tool (APT) like so:

```
master@ubuntu-server:~$ sudo apt-get -y install \
 dovecot-imapd dovecot-pop3d
```

During the package installation on Debian-based distros, you will be prompted to create self-signed certificates for using IMAP and POP3 over SSL/TLS. Select **Yes** when prompted.

You will also be prompted for the hostname to use for the commonName field of the self-signed certificate. Input the correct hostname for your system and select Ok to continue.

Installing Dovecot from Source

Begin by downloading the latest Dovecot IMAP and POP3 server software to **/usr/local/src or $HOME/src**. The latest version can be found at http://dovecot.org/download.html. We'll download version 2.3.9.2 (dovecot-2.3.9.2.tar.gz) in our walkthrough. Once it is downloaded, unpack it as follows:

```
[master@server src]$ tar xvzf dovecot-2.*tar.gz
```

This will create a new directory under which all the source code will reside. For the version we are using, you will see a new directory called **dovecot-2.3.9.2** created. Change into the directory as follows:

```
[master@server src]$ cd dovecot-2.*/
```

The defaults that ship with the Dovecot server work well for most installations. But as with most other enterprise-grade software of its caliber, Dovecot offers you many configurable features that you can turn on or off during the configuration and build process. In addition to the default build options, we will enable a couple of simple but important configuration tweaks.

First, we want to make sure that the Dovecot server that we build from source is able to support secure communication using the OpenSSL libraries. For this, we need to make sure that proper libraries (libssl) are available on the system and that the appropriate option is specified during configuration. For Red Hat–like distros such as Fedora, CentOS, and RHEL, make sure that the **openssl-devel** package is installed. And for Debian-based distros like Ubuntu, make sure you have the **libssl-dev** package installed.

Second, we will specify a prefix option of **/usr/local/dovecot** to specify the install location for all Dovecot-related files. Let's begin.

Look through the **INSTALL** file in the software source tree.

Configure the build environment, with the desired custom options:

```
[master@server dovecot-*]$./configure --with-ssl=openssl --prefix=/usr/local/dovecot
```

Begin the software compilation by running the following command:

```
[master@server dovecot-*]$ make
```

The entire build/compile process might take only a few minutes.

Once **make** completes without any errors, you can install Dovecot under the prefix directory by running this:

```
[master@server dovecot-*]$ sudo make install
```

The preceding command will result in Dovecot binaries being installed under the following directories: **/usr/local/dovecot/bin**, **/usr/local/dovecot/sbin**, and **/usr/local/dovecot/libexec/dovecot**. The executables should be run only by root, so make sure to limit non-privileged access to them accordingly. And that's it!

Dovecot Configuration File	Description
/etc/dovecot/dovecot.conf	Dovecot's main configuration file
/etc/dovecot/conf.d/10-auth.conf	Configuration file for the different authentication modules
/etc/dovecot/conf.d/10-mail.conf	Stores mailbox locations and namespaces
/etc/dovecot/conf.d/10-master.conf	Stores options for various Dovecot services
/etc/dovecot/conf.d/90-plugin.conf	Stores plug-in-specific settings
/etc/dovecot/conf.d/10-ssl.conf	Stores SSL-related settings
/etc/dovecot/conf.d/20-imap.conf	Stores IMAP-specific configuration options
/etc/dovecot/conf.d/20-pop3.conf	Stores POP3-specific configuration options

Table 21-1 Dovecot Configuration Files

Dovecot Configuration Files and Options

Dovecot provides you, the system administrator, with a wide range of configuration options to help you meet the mailing needs of your end users. The software is helpfully modularized, and you can easily pick and choose which modules you want to customize.

The primary configuration file for Dovecot is **dovecot.conf**, which most mainstream Linux distros store under **/etc/dovecot/**. Using an **`include`** directive (**`!include conf.d/*.conf`**) in **dovecot.conf**, some Linux distros go further and organize (split) various files that control different aspects of Dovecot under the **/etc/dovecot/conf.d/** subdirectory.

Table 21-1 shows the location and descriptions of some Dovecot configuration files.

The configuration files accept a rich set of options that can be used to control and tune various aspects of Dovecot as well as turn features on and off. Table 21-2 describes only a small subset of the more oft-used configuration options.

Configuring Dovecot

After installation (from source or binary), the next step is to configure or customize your Dovecot instance to suit your environment. The software ships with many sane default settings that can be used *as is* out of the box with very little customization from you.

To get Dovecot up and running quickly, you may have to tweak some configuration parameters at a minimum. The parameters that we will be changing are shown in Table 21-3, along with the desired target values. Table 21-2 also describes the parameters.

Configure Protocols

Use the **`doveconf`** utility to make sure that your server supports LMTP, POP3, and IMAP:

```
[master@server ~]$ sudo doveconf  protocols
```

If any of the protocols are missing, open the main **/etc/dovecot/dovecot.conf** configuration file with any text editor, look for the **`protocols`** setting, and update it to look like the line here:

```
protocols = imap pop3 lmtp
```

Dovecot Config Option	Description
`listen`	A comma-separated list of IPs or host addresses on which the Dovecot server listens for connections. An example is **listen = *, ::** where **`*`** listens on all IPv4 interfaces and **`::`** listens on all IPv6 interfaces.
`protocols`	A space-separated list of protocols for the Dovecot server to serve. For example: **protocols = imap pop3 lmtp submission**
`service`	Services are used for implementing various Dovecot functions. Some common services are **aggregator**, **anvil**, **auth-worker**, **auth**, **config**, **dict**, **director**, **dns_client**, **doveadm**, **imap-login**, **imap-urlauth-login**, **imap-urlauth-worker**, **imap-urlauth**, **imap**, **indexer-worker**, **indexer**, **ipc**, **lmtp**, **log**, **pop3-login**, **pop3**, **replicator**, **ssl-params**, and **stats**.
`mail_location`	Location for users' mailboxes. This is unset by default and thus you need to supply a value for it. At least one of the locations should correspond to the location of the SMTP server's mail spool directory. The mbox, Maildir, and dbox mailbox formats are all supported. An example setting is: **mail_location = mbox:~/mail:INBOX=/var/mail/%u** where **`mbox`** is the mailbox format and **`%u`** is a special variable meaning the username.
`ssl`	Controls SSL/TLS support. Here are the possible values: **`yes`** SSL/TLS is used for plain-text authentication mechanisms. **`no`** SSL/TLS is not required. **`required`** SSL/TLS is required for both plain-text and non-plain-text authentication.
`ssl_cert`	Path pointing to the location of a PEM-encoded X.509 SSL/TLS certificate file.
`ssl_key`	Path pointing to the location of a PEM-encoded X.509 SSL/TLS private key file.
`disable_plaintext_auth`	Disables all plain-text authentication methods (including **`LOGIN`** command) *unless* SSL/TLS is used. Note that connecting locally from the same Dovecot server is considered secure and plain-text authentication is allowed. The default value is **`yes`**.
`auth_mechanisms`	A list of wanted authentication mechanisms. The authentication types can be broadly categorized into **PlainText** and **Non-plaintext** authentication mechanisms. Here are some possible values: **`plain`** Most widely supported authentication method, because the client simply sends the password unencrypted (plain text) to Dovecot. **`login`** Usually used by SMTP servers to let e-mail clients perform SMTP authentication. This is also a plain-text mechanism. **`cram-md5`** Protects the password in transit against eavesdroppers. Offers good support in clients. **`ntlm`** Suite of authentication and session security protocols created by Microsoft and used in various Microsoft products. **`anonymous`** Provides support for anonymous login. By default, the PLAIN mechanism is enabled.

Table 21-2 Dovecot Configuration Options

Description	Option/Section	Target Value
Protocols we want to support	`protocols`	`protocols = lmtp pop3 imap` (where `lmtp` stands for Local Mail Transfer Protocol)
IP address to listen on	`listen`	`listen = *, ::`
Password database to use for authenticating users	`passdb { driver = }`	`driver = pam`
User database to consult for authenticated users	`userdb { driver = }`	`driver = passwd`
Mail location and mailbox format	`mail_location`	`mail_location = mbox:~/mail:INBOX=/var/mail/%u`
Specify extra groups that should have necessary rights to the mail location	`mail_access_groups`	`mail_access_groups = mail`
SASL mechanisms	`auth_mechanisms`	`auth_mechanisms = plain`
SSL	`ssl`	`ssl = required`
Mail server certificate file	`ssl_cert`	`ssl_cert = </etc/pki/dovecot/certs/dovecot.pem`
Mail server key file	`ssl_key`	`ssl_key = </etc/pki/dovecot/private/dovecot.pem`

Table 21-3 Target Dovecot Configuration Settings

You can also use the **`sed`** utility to quickly find and replace the option you want in the configuration file, like so (all on one line!):

```
# sed -i 's/^\ *protocols\ *=\ *.*$/protocols = lmtp pop3  imap/g' \
/etc/dovecot/dovecot.conf
```

Configure the listen Parameter

Edit the **/etc/dovecot/dovecot.conf** file if necessary and make sure that the **`listen`** entry exists and looks like the one here:

```
listen = *, ::
```

You can use the **`sed`** utility to quickly edit the file in place by running the following:

```
# sed -i 's/^\ *#*listen\ *=\ *.*$/listen = *, ::/g' \
 /etc/dovecot/dovecot.conf
```

Configure System Users and Password Databases (passdb and userdb)

Use **doveconf** to check that the PAM database is one of the drivers configured for authenticating system users. In the same command, also check for the driver being used for the user database:

```
[master@server ~]$ sudo doveconf passdb userdb
```

The **driver** entries for the **passdb** and **userdb** sections of the output should look similar to the following:

```
driver = pam
driver = passwd
```

If the **driver** entries in your output are different, open the **/etc/dovecot/conf.d/auth-system.conf.ext** configuration file and ensure that the **driver** values are set to the corresponding values in Table 21-3.

Configure Mail Location

Use **doveconf** to check the current value of the **mail_location** parameter:

```
[master@server ~]$ sudo doveconf mail_location
```

By default, the **mail_location** parameter is unset. Use any text editor to edit and set the parameter so that the entry in **/etc/dovecot/conf.d/10-mail.conf** looks like the one here:

```
mail_location = mbox:~/mail:INBOX=/var/mail/%u
```

If for some reason you like pain and you derive pleasure from constructing and debugging regular expressions to make edits that are probably easier done by using a text editor, you can use the **sed** utility with some convoluted-looking options (all on one line!) to "quickly" edit the file in place and make the changes that you want by running the following:

```
$  sudo sed  -i  \
's/^\ *\#*mail_location\ *=\ *.*$/mail_location = \
mbox:~\/mail:INBOX=\/var\/mail\/%u/g'  \
/etc/dovecot/conf.d/10-mail.conf
```

Configure Mail Access Group

Use **doveconf** to check the current value of the **mail_access_groups** parameter:

```
[master@server ~]$ sudo doveconf mail_access_groups
```

By default, the **mail_access_groups** parameter is unset. Use any text editor to edit and set the parameter so that the entry in **/etc/dovecot/conf.d/10-mail.conf** looks like the one here:

```
mail_access_groups = mail
```

You can use the **sed** utility to quickly edit the file in place by running the following:

```
$ sudo sed -i \
's/^\ *#*mail_access_groups\ *=\ *.*$/mail_access_groups = mail/g' \
/etc/dovecot/conf.d/10-mail.conf
```

Configure Authentication Mechanisms

Use **doveconf** to check the current value of the **auth_mechanisms** parameter:

```
[master@server ~]$ sudo doveconf  auth_mechanisms
```

Edit the **/etc/dovecot/conf.d/10-auth.conf** file if necessary and make sure that the **auth_mechanisms** entry exists and looks like the one here:

```
auth_mechanisms = plain
```

Configure Dovecot SSL Parameters

Use **doveconf** to check the current values of the **ssl**, **ssl_cert**, and **ssl_key** parameters:

```
[master@server ~]$ sudo doveconf ssl  ssl_cert ssl_key
```

Edit the **/etc/dovecot/conf.d/10-ssl.conf** file if necessary and make sure that the following parameters are set to the values here:

```
ssl = yes
ssl_cert = </etc/pki/dovecot/certs/dovecot.pem
ssl_key = </etc/pki/dovecot/private/dovecot.pem
```

TIP Dovecot IMAP and POP3 server software installed via the package management system on most distros ships with generic SSL certificates and key files that are used in the **10-ssl.conf** configuration file.

Dovecot ships with a simple script named **mkcert.sh** that can be used to generate custom certificates and keys that can be used with your Dovecot instance. To use the script, first customize the **/etc/pki/dovecot/dovecot-openssl.cnf** file with your custom settings and then execute the script. To run the script on a Red Hat–like distro such as Fedora, type the following:

```
[master@fedora-server ~]$ sudo /usr/libexec/dovecot/mkcert.sh
```

CAUTION While trying to connect to your Dovecot server, users will receive a warning that the certificate is not properly signed if you create and use self-signed certificates. If you do not want this warning to appear, you can obtain a certificate (for free) from a certificate authority (CA) such as the Let's Encrypt Project (https://letsencrypt.org/) or purchase one from Comodo, Symantec/Thawte, Symantec/VeriSign, and so on. Depending on your specific environment, this might or might not be a requirement. However, if all you need is an encrypted tunnel through which passwords can be sent, a self-signed certificate works fine.

Running Dovecot

After configuration, the next step is to learn how to control the Dovecot services. This includes how to start, restart, stop, and enable Dovecot. The following instructions apply to a Dovecot instance installed via the distro's package management system. To control or manage the Dovecot instance compiled and installed from source, you will have to tweak the steps a bit and specify the correct paths to the commands/binaries.

On Linux distros such as modern versions of Fedora, CentOS, Ubuntu, and RHEL that use **systemd** as the service manager, check the status of the Dovecot service by running the following:

```
[master@server ~]$ sudo systemctl status  dovecot
```

To stop Dovecot (assuming it's currently running), type this:

```
[master@server ~]$ sudo  systemctl stop  dovecot
```

To start Dovecot, type this:

```
[master@server ~]$ sudo systemctl start  dovecot
```

To disable Dovecot from automatic startup and stopping it (if running), type this:

```
[master@server ~]$ sudo systemctl  --now disable dovecot
```

To configure the Dovecot IMAP and POP3 services to automatically start up during system boot and simultaneously start the services, type the following:

```
[master@server ~]$ sudo systemctl  --now enable dovecot
```

TIP Dovecot software suite comes with a built-in administration tool named **doveadm**. This tool is distribution agnostic, so you can expect it to work in the same way regardless of how your Dovecot instance was installed. **doveadm** is a powerful tool and can be used to control and manage many aspects of Dovecot, such as reloading, stopping, logging, testing, and so on.

Checking Basic POP3 Functionality

If everything has worked correctly, you should now have a running IMAP server and POP3 server. The next logical step is to test the services for actual functionality.

We begin by using Telnet to connect to the POP3 server (localhost in this example). From a command prompt, type the following:

```
[master@server ~]$ telnet localhost 110
Trying ::1...
Connected to localhost.
Escape character is '^]'.
+OK Dovecot ready.
```

The server is now waiting for you to give it a command. (Don't worry that you don't see a prompt.) Start by submitting your login name as follows:

```
USER yourlogin
```

Here, ***yourlogin*** is, of course, your login ID. The server might respond with something like this:

```
+OK [User name accepted, password please]
```

Now tell the server your password using the **PASS** command:

```
PASS yourpassword
```

Here, ***yourpassword*** is your password. The server might respond with this:

```
+OK Logged in.
```

You're now logged in and can issue commands (such as **LIST**, **STAT**, and **RETR**) to read and manage your mail. Since you are simply validating that the server is working, you can log out now. Simply type **QUIT**, and the server will close the connection:

```
QUIT
+OK Logging out.
Connection closed by foreign host.
```

That's it.

Checking Basic IMAP Functionality

We will use Telnet to connect to the IMAP server (localhost in this example) and test it for basic IMAP functionality. From the command prompt, type the following:

```
[master@server ~]$ telnet localhost 143
```

The IMAP server will respond with something similar to this:

```
* OK [CAPABILITY IMAP4rev1 LITERAL+ SASL-IR
LOGIN-REFERRALS ID ENABLE IDLE
STARTTLS AUTH=PLAIN] Dovecot ready.
```

The server is now ready for you to enter commands. Note that like the POP3 server, the IMAP server will not issue a prompt.

The format for IMAP commands is shown here:

```
<tag><command>  <parameters>
```

Here, ***<tag>*** represents any unique (user-generated) value used to identify (tag) the command. Example tags are **A001**, **b**, **box**, **c**, **box2**, **3**, and so on. Commands can be executed asynchronously, meaning that it is possible for you to enter one command and, while waiting for the response, enter another command. Because each command is tagged, the output will clearly reflect what output corresponds to what request.

To log into the IMAP server, simply enter the **login** command, like so:

```
A001 login <username>  <password>
```

Here, ***<username>*** is the username you want to test and ***<password>*** is the user's password. If the authentication is a success, the server will respond with something like this:

```
A001 OK [CAPABILITY IMAP4rev1 LITERAL+ SASL-IR LOGIN-REFERRALS
...<OUTPUT TRUNCATED>...]
Logged in
```

That is enough to tell you two things:

- The username and password are valid.
- The mail server was able to locate and access the user's mailbox.

With the server validated, you can choose from and issue a multitude of IMAP commands to manage your mailbox or logoff by simply typing the **logout** command:

```
A002 logout
```

The server will reply with something similar to this:

```
* BYE Logging out
A002 OK Logout completed.
Connection closed by foreign host.
```

The doveconf Utility: Viewing Dovecot Services and Modules

Earlier on, we hinted that **dovecot** processes provide various services that make up the Dovecot ecosystem. These are the actual **dovecot** workhorses. These services themselves may be implemented by stand-alone modules or via a built-in Dovecot process. Some of the services are listed in Table 21-2 under the service configuration options. You can view the details about all the services by using the **doveconf** utility, like so:

```
$ sudo  doveconf  service
```

If you want to see the setting of a particular service section, you can do so by dumping just that section. For example, to dump the service settings for the **imap-login** service section alone, run the following:

```
$ sudo doveconf service/imap-login
```

Services that are implemented via stand-alone modules often have their own module-specific parameters or options that can be tweaked. The module-specific configuration files are often stored under the **/etc/dovecot/conf.d/** directory. For example, the **pop3** service is backed by the pop3 module. To view the module-specific settings for **pop3**, type this:

```
$ sudo doveconf   -m pop3
```

To drill down and view only the namespace section for the pop3 module, type this:

```
$ sudo doveconf   -m pop3  namespace
```

Other Issues with Mail Services

Thus far, we've covered enough material to get you started with a working mail server, but there is still a lot of room for improvements. In this section, we walk through some of the issues you might encounter and some common techniques to address them.

SSL/TLS Security

Best security practices should be a big objective in any mail server (such as POP3 and IMAP) implementation. Some mail server implementations do not ship with secure options enabled out of the box (possibly to help make initial configuration easier). Some implementations may offer varying levels of support for encryption, password-hashing schemes, user/password databases, and so on. There is also the issue of ensuring that the majority of e-mail clients that will use the mail server are properly supported. Regardless of the e-mail server software stack you settle on, you should ensure that, at a minimum, you enable encryption of the entire protocol stream whenever possible.

Fortunately for us, the Dovecot IMAP and POP3 server implementation that we installed earlier was written from the ground up with security in mind, and it also ships with sane and secure default configuration options. This is one of the reasons why we did not need to do too much in the way of configuring SSL support for our **Dovecot** instance, thus keeping things simple! We made sure that the SSL support is enabled and we accepted the default *certificates* and *keys*. Besides keeping things simple, our approach hopefully made for a nice confidence booster for you to be able to get something working quickly—before we start tinkering too much and adding other layers of complexity.

The Telnet protocol that we used for testing POP3 and IMAP functionality earlier is not a secure protocol by default. Rather, by default everything done over Telnet is transmitted in plain text. So we were connecting and testing the POP3 and IMAP server over an unencrypted channel. You may be wondering, then, why we said that Dovecot is secure out of the box. You may also be wondering what's the point of *enabling* SSL when the system allows us to successfully connect insecurely.

Well, we are glad you caught us and called us out before we continue venturing down this possible path of lies and vignette of deceit. Kindly allow us to explain:

- We assumed that all our testing with Telnet was being done from and to the same computer (localhost) running the Dovecot server software. The Telnet testing would have failed if we had tried to do it from a different system.
- By default in Dovecot, plain-text (nonsecure) authentication is always permitted for connections originating from the localhost. This means you can connect to Dovecot without using SSL, or even configuring it, *whenever* you connect from localhost.

Now that we have (hopefully) earned your trust again, we'll go ahead and make sure our POP3 and IMAP server is truly secure by testing it from a different computer. We will use the Swiss Army program of all things related to SSL—the OpenSSL program suite—to do our testing in the following section.

TIP The operating system's logging subsystem is an indispensable tool when troubleshooting mail server issues, so make sure you keep an eye on the log files when you are troubleshooting issues such as the **dovecot** service failing to start or restart properly (for example, via "**`journalctl -xe -u dovecot`**").

Testing POP3 and IMAP Connectivity over SSL/TLS

While still logged onto the Dovecot server, first make sure that the TCP ports for POP3 and IMAP are open on the firewall for external connections.

On a Red Hat–like distro such as our Fedora server, we can use the **`firewall-cmd`** command to do this. We'll permanently add a rule to the current firewall ruleset to achieve this by running the following command:

```
[master@fedora-server ~]$ sudo firewall-cmd --permanent \
 --add-port=110/tcp  --add-port=143/tcp
```

Next, reload the firewall rules so that the new rule is immediately active:

```
[master@fedora-server ~]$ sudo firewall-cmd --reload
```

Now let's hop over to a different box and remotely test our Dovecot POP3 and IMAP services. Let's assume that our remote Dovecot server's IP address is 192.168.56.101.

1. Make sure our previous Telnet test will fail if we try to do it from a remote system:

   ```
   ubuntu-client:~$ telnet  192.168.56.101  110
   Trying 192.168.56.101...
   Connected to 192.168.56.101.
   Escape character is '^]'.
   +OK Dovecot ready.
   ```

2. The initial connection to the POP3 port, 110, succeeded. Let's issue the first POP3 command to begin the SASL process. At the Dovecot prompt, start by submitting your login name as follows:

   ```
   USER remoteloginname
   ```

   Here, ***remoteloginname*** is the username of a user on the remote Dovecot server with a POP3 mailbox.

   We are immediately *stopped* from going any further by the response we get from the remote Dovecot server:

   ```
   -ERR [AUTH] Plaintext authentication disallowed
   on non-secure (SSL/TLS) connections.
   ```

   So, we've further supported our claim that nonsecure connections are not supported by default on a Dovecot server. Issue the **`QUIT`** command to end the current POP3 session.

3. Now let's try to connect to the remote server using **`openssl`**. Use **`openssl`** to connect to the same remote POP3 server using **`STARTTLS`**:

   ```
   ubuntu-client:~$ openssl s_client -connect \
   192.168.56.101:110  -starttls pop3
   ```

If the server is listening securely on port 110, you should be greeted with a bunch of SSL transaction-related messages and a prompt similar to this one:

```
depth=0 OU = IMAP server, CN = imap.example.com,
 emailAddress = postmaster@example.com
Server certificate
-----BEGIN CERTIFICATE-----
MIICQzCCAaygAwIBA
...<OUTPUT TRUNCATED>...
+OK Dovecot ready.
```

4. As before, the initial connection to the POP3 port 110 succeeded. Let's issue the first POP3 command to begin the SASL process. At the Dovecot prompt, submit a login name as follows:

```
USER remoteloginname
+OK
```

The first sign of progress is that the POP3 server allowed us to issue a **USER** command and submit the username over the **openssl**-protected connection.

Issue the POP3 **PASS** command to submit the password:

```
PASS  password_for_remoteloginname
```

Here, ***password_for_remoteloginname*** is the password associated with the ***remoteloginname*** username.

```
+OK Logged in.
```

The "+OK Logged in" output from the server shows that we logged in successfully! You can now continue to issue other POP3-related commands to interact with the remote mailbox. Issue **QUIT** to exit.

5. You can similarly test the IMAP service on the remote Dovecot server using **openssl**. Use **openssl** to connect to the remote IMAP server listening on port 143 using **STARTTLS**:

```
ubuntu-client:~$ openssl s_client -connect 192.168.56.101:143 \
 -starttls imap
...<OUTPUT TRUNCATED>...
Verify return code: 18 (self signed certificate)
---
. OK Pre-login capabilities listed, post-login capabilities have more.
```

6. You should be able to continue issuing supported IMAP protocol commands (such as **LOGIN**, **LOGOUT**, and so on) to authenticate yourself and interact with the remote server.

TIP Remember that there isn't too much point in implementing security if nobody is using it, so make sure that your mail clients use SSL when connecting to the IMAP or POP3 server. In most of the popular e-mail client programs, such as Thunderbird, Evolution, Outlook, and so on, the option to enable SSL may be as simple as a check box in the Email Account configuration options.

Availability

In managing a mail server, you will quickly find that e-mail qualifies as one of the most visible resources on your network. When the mail server goes down, everyone will know, they will know quickly, and, worst of all, they may even alert you, the administrator, before you even realize that something is amiss! Thus, it is important that you carefully consider how you will be able to provide 24/7 availability for e-mail services.

A simple issue that can threaten mail servers is "fat fingering" a configuration—in other words, making an error when performing basic administration tasks. There is no solution to this problem other than *being careful!* When you're dealing with any kind of production server, it is prudent to perform each step carefully and make sure you type what you meant to type. When at all possible, work as a normal user rather than root and use **sudo** for specific commands that need root permissions.

The second big issue with managing mail servers is hardware availability. Unfortunately, this is best addressed with money. The more the better! Make an investment up front in a good server chassis. Adequate cooling and as much redundancy as you can afford is a good way to make sure the server doesn't take a fall over something silly like a CPU fan going out. Employing dual power supplies is another way to help keep mechanical things from failing on you. Uninterruptible power supplies (UPSs) for your servers are *almost always* a must. Make sure that the server disks are configured in some kind of RAID fashion. This is all to help mitigate the risk of hardware failure.

Finally, consider expansion and growth early in your design. Your users will inevitably consume all of your available disk space. The last thing you will want is to start bouncing mail because the mail server has run out of disk space! To address this issue, consider using disk volumes that can be expanded on the fly and RAID systems that allow new disks to be added quickly. This will allow you to add disks to the volume with minimal downtime and without having to move to a completely new server.

Log Files

Although we've mentioned this earlier in the chapter, watching the **/var/log/messages**, **/var/log/syslog**, **/var/log/maillog**, and **/var/log/mail.log** files is a prudent way to manage and track the activities on your mail server. The Dovecot software provides a rich array of logging options and log messages to help you understand what is happening with your server and troubleshoot any peculiar behavior. In short, when in doubt, take a moment to look through the log files. You'll probably find a solution or pointer to your problem there.

Summary

This chapter covered some theory behind the IMAP and POP3 protocols, ran through the complete installation for the Dovecot software (from source and from prepackaged binaries), and discussed how to test connectivity to each service manually. With this chapter, you have enough information to set up and run a simple POP3 and IMAP server instance.

The chapter also covered enabling secure access to your mail server assets via SSL/TLS. This is an easy way to prevent clear-text passwords (embedded in IMAP or POP3 traffic) from making their way into hands that should not have them. We ended by touching on some basic human- and hardware-related concerns, necessities, and precautions in regard to ensuring that your mail server is available 24/7.

If you find yourself needing to build out a larger mail system, take the time to read/learn more about the mail server software of your choice (such as Dovecot, Cyrus, UW IMAP, or Courier). If you find that your environment requires more groupware functionality (such as provided with Microsoft Exchange Server), you might want to check out other software, such as Scalix, Open-Xchange, Zimbra, Horde Groupware, and EGroupware. They all provide significant extended capabilities at the expense of additional complexity in setup and configuration.

As with any server software that is visible to the outside world, you will want to keep up to date with the latest releases. Thankfully, the Dovecot package has shown sufficient stability and security so as to minimize the need for frequent updates, but a watchful eye is still nice. Finally, consider perusing the latest IMAP and POP RFCs to understand more about the protocols. The more familiar you are with the underlying protocols, the easier you'll find troubleshooting to be.

CHAPTER

22 Voice over Internet Protocol (VoIP)

Since its introduction in 1995, Voice over IP, or simply VoIP, like Linux, has evolved rapidly, and it continues to revolutionize communication, replacing traditional telephone networks all around the world. Every day, more and more people communicate over VoIP. International and continental long-distance calls, local calls, and even calls via smartphone apps (applications on smartphones) utilize VoIP networks in ways most people never even think about.

In the following sections, we will review the fundamentals of VoIP technology and Internet telephony, describe the basic elements of VoIP, and show an implementation of VoIP using the open source framework created by the popular Asterisk project (www.asterisk.org). By the end of the chapter, you will have set up a fully functional VoIP system with which you can make and receive phone calls.

VoIP Overview

In its simplest form, VoIP is the transmission of digitized speech in "data packets" from one IP address to another over one or more IP networks, such as the Internet or a LAN. A VoIP system takes in uncompressed speech as an input, converts it into compressed digital form, packetizes it, puts it on the wire, and sends it to the next station, where the process is reversed, as shown in Figure 22-1. The accuracy and clarity of the reconstructed signal is dependent on various technical factors, such as the compression algorithms used, the encoding and decoding algorithms used, the sample rate, and so on. This is obviously an oversimplified version of how VoIP works.

As with most complex systems, the VoIP system can be broken down into several individual components. A subset of the components we will be exploring in the following sections includes VoIP servers, analog telephone adapters, IP phones, and VoIP protocols.

VoIP Server

The VoIP server plays the role of the communications switch. It is central to all VoIP communications and is responsible for directing and controlling the VoIP communications

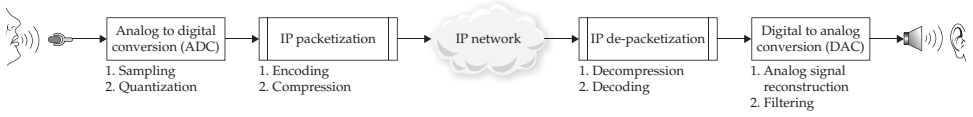


Figure 22-1 Basic principles of VoIP

among the various components. The VoIP server provides ways for end users to make and receive VoIP calls or interact with VoIP systems and other VoIP components.

Besides the seemingly all-important role of the VoIP server, it isn't terribly useful by itself. Its needs and relies on the other components to realize its full potential.

Analog Telephone Adapter (ATA)

An ATA is a simple device that lets you connect any standard (analog) telephone to your network so it can use VoIP communication through your Internet connection. The ATA converts the analog signal from your telephone into digital data that can be transmitted over IP networks, such as your LAN or the Internet, and vice versa. VoIP providers, such as Vonage, Lingo, and magicJack, typically bundle an ATA device with their VoIP telephony service to make it easier for new customers to use their existing traditional analog phones with the VoIP service.

IP Phones

IP phones use VoIP technologies to transmit and receive telephone calls over the IP network. The IP phone converts your speech directly into digital signals and transmits it over the IP network, and vice versa. There are two common forms of IP phones:

- **Hardphone** A digital phone set that connects directly to an IP network without any other device.
- **Softphone** A software application that runs on a device such as your PC, smartphone, or tablet and relies on the underlying hardware to do the heavy lifting.

Figure 22-2 shows a simplified network layout of a VoIP setup using an ATA and IP phones.

VoIP Protocols

A typical VoIP connection generally involves a series of handshakes, also known as *signaling,* between the VoIP endpoints (and the components in between) to exchange information that culminates in two persistent media streams (one for each direction) that carry the actual conversation. Several protocols have been created to handle this exchange. We'll discuss some of those that are important to VoIP in general and to Asterisk specifically.

For clarity, we categorize the VoIP protocols into three groups: signaling protocols, media protocols, and codecs.

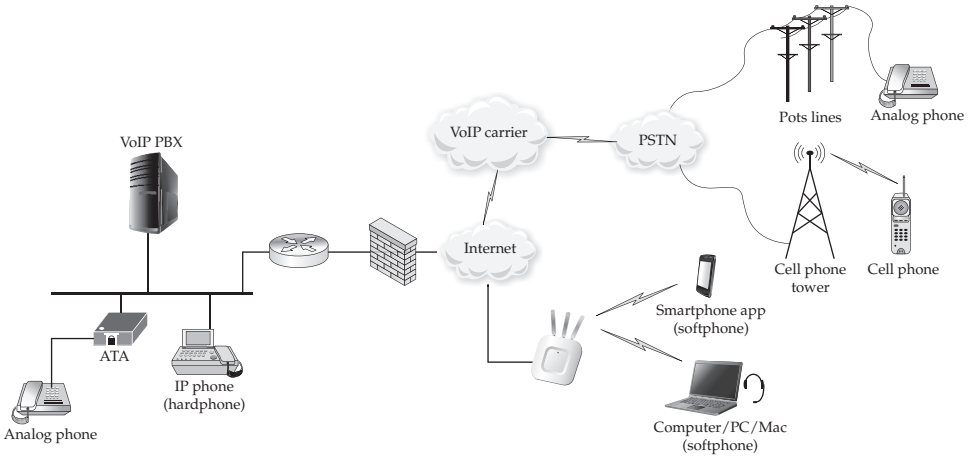


Figure 22-2 VoIP network layout using an ATA and IP phones

Signaling Protocols

Generally, VoIP signaling controls the setting up, modifying, and tearing down of a conversation. The signaling protocols are not concerned with the nitty-gritty details of the data streams carrying the actual media content but rather focus on verifying the identities of the calling and called parties (authentication), ensuring that the network is not misused (authorization), and communicating the capabilities, such as call features and security, supported by the endpoints.

In rough chronological order of introduction, the most widely used signaling protocols for VoIP systems are detailed in Table 22-1.

Signaling Protocol	Description
H.323	An ITU-T standard that provides multimedia video conferencing, voice, and data capability for use over packet-switched networks.
Media Gateway Control Protocol (MGCP)	ITU/IETF RFC 3435. An alternative to H.323 that is used for controlling media gateways from external call control elements called media gateway controllers or call agents.
Session Initiation Protocol (SIP)	IETF RFC 3261. A plain-text signaling protocol that uses HTTP-like messages to initiate, modify, and terminate VoIP connections.
Skinny Client Control Protocol (SCCP, aka Skinny)	Cisco proprietary protocol that defines a simple messaging set that allows skinny clients to communicate with the Cisco Unified Communications Manager.
Extensible Messaging and Presence Protocol (XMPP)	ITU/IETF RFC 6120. Based on Extensible Markup Language (XML), XMPP is used for exchanging near-real-time communications, such as instant messages, presence, and session control (signaling) for VoIP communications.

Table 22-1 Signaling Protocols

Media Protocol	Description
Real-time Transport Protocol (RTP)	Defines how real-time multimedia data is delivered over IP networks. RTP data streams are carried in connectionless UDP packets and are designed to be independent of the underlying Transport and Network layers.
Real-time Transport Control Protocol (RTCP)	RTCP carries statistical and control data for RTP sessions such as delay, jitter, and packet loss calculations.

Table 22-2 Media Protocols

Media Protocols

Media protocols are responsible for transporting multimedia data (such as audio and video) over IP networks. RTP and RTCP (RFC 3550) are closely linked media protocols that are often used for this purpose. Both are described in Table 22-2.

Codecs

A codec (short for *co*der-*dec*oder) is a set of instructions used to convert analog signals to digital form and then back again. The codec used is often referred to as the encoding method or the payload type for the RTP packet. Codecs differ in a number of ways, including sound quality, bandwidth, and computational requirements. Common codecs are described next:

- **G.711** Does not use compression and delivers precise speech transmission (toll quality). Mostly used over LAN or on high-bandwidth networks. Consists of two variants: μ-law, used in North America and Japan, and A-law, used in Europe and elsewhere.
- **G.722** A wideband codec that provides high sound quality and adapts well to varying compressions.
- **G.729A** Delivers good audio quality and provides excellent bandwidth utilization. It typically requires a license to use.
- **GSM** Similar to encoding used in GSM mobile phones. It provides high compression ratio and is free and available on many VoIP platforms.
- **Opus** An extension to the SILK codec developed by Skype. Supports constant and variable bitrate encoding and has good packet loss concealment (PLC).
- **iLBC** Internet Low Bitrate Codec. It is very resilient to packet loss and works well in low-bandwidth networks.

VoIP Implementations

A good number of VoIP systems are now implemented as services, and users typically only interact with the systems using endpoints such as a phone set, software application, or

smartphone app to communicate with other users. The internal workings of these systems are complex and usually hidden from users.

VoIP systems can also be implemented to replace components of the traditional telephony infrastructure. Sample popular open source VoIP projects that provide Private Branch Exchange (PBX) functionality are FreeSWITCH, OpenSIPs, Asterisk, and FreePBX. We will focus on Asterisk because of its maturity and strong community base, which continually develops new features. Asterisk is also well documented with numerous online resources and individuals available to assist.

Asterisk

So what is Asterisk? Sponsored by Digium, Inc., Asterisk is their community software for building VoIP and PBX replacements. It is *very* flexible, runs on a variety of platforms, and supports numerous gateways, effectively turning an ordinary computer into an advanced communications server.

Asterisk software is used in IP PBX systems, VoIP gateways, conference servers, and other embedded applications. It includes high-end features such as interactive voice response (IVR), voicemail, conference calling, automatic call distribution (ACD), and so on. New functionalities can be created by writing scripts in Asterisk's native language, Perl, or other languages, and by writing modules in C. Asterisk is capable of interfacing with many traditional telecom protocols, VoIP protocols, and a majority of the standard codecs. To sum it all up, Asterisk can be used to create powerful, programmable VoIP systems at relatively low cost compared to proprietary PBXs with similar features.

We will go over installing and configuring an Asterisk system and also discuss the structure of the main Asterisk configuration files.

How Asterisk Works

Asterisk consists of several software components and modules. This modularity provides the flexibility to build a customized Asterisk-based solution for your project. For example, the PJSIP module will allow your Asterisk system to communicate with SIP phone endpoints, while the Call Details Record (CDR) module will add call-reporting capabilities.

On a high level, Asterisk has a *core* that can interact with many modules. The modules, also called channel drivers, provide the *channels* that are driven by the Asterisk *dialplan* to execute programmed behavior and facilitate communication between devices or programs outside Asterisk. Channels often use *bridging* infrastructure to interact with other channels.

We will dig deeper into these concepts in later sections.

Asterisk Installation

Asterisk is complex. There are many steps to cover to install a fully functional system. The checklist in Table 22-3 provides the high-level steps in rough order required to install Asterisk.

No.	Step	Component
1	Asterisk installation	Linux server
2	Starting and stopping Asterisk	Asterisk configuration files
3	Configuring the local firewall for Asterisk	Linux server and/or external firewall
4	Configuring the PBX	Asterisk configuration files (**pjsip.conf**, **extensions.conf**, and so on)
5	Testing	Phone system and Asterisk

Table 22-3 High-Level Steps to Install Asterisk

To keep things simple on our sample Fedora server, we'll install the prepackaged Asterisk version 16 as well as the **asterisk-pjsip** package, which provides the PJSIP module. Without further ado, let's get on with it!

1. Start off by disabling SELinux:

```
[master@server ~]$  setenforce 0
[master@server ~]$  sed -i 's/\(^SELINUX=\).*/\SELINUX=disabled/' \
/etc/selinux/config
```

2. Verify that the Asterisk package exists in your distros' online repository:

```
[master@fedora-server ~]$ dnf info asterisk
```

3. Once this is verified, install the Asterisk packages that we'll be using:

```
[master@fedora-server ~]$ sudo dnf -y install asterisk asterisk-pjsip
```

And that's it—you now have Asterisk installed on your Fedora server!

Starting and Stopping Asterisk

In the following steps, we go over how to manage your Asterisk server after it has been successfully installed:

1. On **systemd**-enabled distros, you can use **`systemctl`** to stop the Asterisk service:

```
[master@server ~]$ sudo systemctl stop  asterisk
```

2. Use the **`systemctl`** command to enable the Asterisk service for automatic startup and simultaneously start it by running the following:

```
[master@server ~]$ sudo  systemctl --now enable asterisk
```

3. Once you've verified that the service is running, you can connect to the Asterisk console:

```
[master@server ~]$ sudo asterisk -rv
Asterisk 16.*, Copyright (C) 1999 - 20xx, Digium, Inc. and others.
...<OUTPUT TRUNCATED>...
server*CLI>
```

Understanding Asterisk Configuration Files and Structure

The configuration files are the core of an Asterisk system. They give it character, purpose, and personality and differentiate one Asterisk implementation from another.

Table 22-4 lists the default installation paths for Asterisk component files and libraries. This is not an exhaustive list; only the core components relevant to this chapter are listed.

To get started, we will focus on the specific files, and the options within those files, that are required to configure your first PBX.

TIP Asterisk ships with sample configuration files that serve as very good reference to learn about other advanced options available in Asterisk configuration files.

The following are the configuration files you would interact with most of the time when configuring your Asterisk system:

- **pjsip.conf** Used to configure the PJSIP channels
- **extensions.conf** Used to configure the dialplan
- **modules.conf** Used to enable or disable the resources available to the Asterisk system

PJSIP Channel Config: pjsip.conf

The PJSIP channel config file, **pjsip.conf**, is a flat text file where we configure everything related to the SIP protocol; this includes adding new SIP users and defining SIP providers for both inbound and outbound calls. **chan_pjsip** is the new channel driver that replaces the older **chan_sip** driver, and thus, **pjsip.conf** replaces **sip.conf**. Unlike **chan_sip**, where everything is a channel, **pjsip** has a number of different conceptual objects.

pjsip.conf is composed of sections like most configuration files used with Asterisk. Each section defines the configuration for a configuration object within **res_pjsip** or an associated module. All section names are encased in square brackets, and anything that follows a section name is applied to that section. Each section will have a type option that defines what kind of section is being configured. You'll see that in the following sample configuration sections.

Section Names In general, you can name a section whatever makes sense to you. However, in some cases, specifically for the endpoint and/or types, the section name has a relationship

Path	Description
/etc/asterisk/	Contains lots of configuration files used to describe the functionalities of the Asterisk system and to run it.
/usr/lib64/asterisk/modules/	Contains all of the modules loadable in the Asterisk system, such as the various applications, codecs, and channels.
/usr/sbin	Contains the executable binary files.
/var/log/asterisk	Contains the Asterisk messages, errors, and CDR information.

Table 22-4 Asterisk File Structure

to its function and must match the user portion of the SIP URI in the "From" header for inbound SIP requests.

Section Types We will describe a few of the common sections you will use in your first Asterisk PBX. There are many config options for some of the sections, but the following examples are deliberately minimal for the sake of simplicity. For more details on the available options, visit https://wiki.asterisk.org/wiki/display/AST/Asterisk+16+Configuration_res_pjsip.

- **Transport** Configures the Transport layer options, like **`tcp`**, **`udp`**, and **`websockets`**, and the encryption methods, like TLS/SSL, the endpoint devices will use.

```
[trans-udp]
type=transport
protocol=udp
bind=0.0.0.0
```

  Here are the explanations for these settings:

type	Must be of type **`transport`**.
protocol	Protocol to use for SIP traffic. The supported options include udp, tcp, tls, ws, and wss. The default is udp.
bind	IP address and optional port to bind to for this transport. The default port is 5060.

- **AoR** Address of Record (AoR) specifies where an endpoint can be contacted. The **`contact`** object in the AoR can be configured dynamically, when a SIP endpoint is configured to register:

```
[1001]
type=aor
max_contacts=1
```

  Or manually, when the endpoint is not configured for registration:

```
[sip-phone]
type=aor
contact=sip:1001@192.168.1.101:5060
```

  Here are the explanations for these settings:

`type=aor`	Must be of type **`aor`**.
`max_contacts`	Maximum number of contacts (SIP endpoints) that can bind to an AoR. The default is 0.
`contact`	Permanent location of the endpoint assigned to the AoR.

- **Auth** This section holds inbound or outbound authentication options and credentials for use by endpoints, trunks, and registrations:

```
[1001-auth]
type=auth
auth_type=userpass
password=1001
username=1001
```

Here are the explanations for these settings:

type=auth	Must be of type **auth**.
auth_type	Specifies the authentication method and credentials to use to authenticate an endpoint inbound request. Supported values include **userpass** and **md5** hash.
username	Username for the account.
password	Plain-text password used for authentication, if **auth_type=userpass**.
md5_cred	MD5 hash used for authentication, if **auth_type=md5**.

- **Identify** Maps a host directly to an endpoint. If we do not define an **Identify** section, user identity is based on the packet's SIP "From" header.

```
[1001]
type=identify
endpoint=1001
match=192.168.1.101
```

Here are the explanations for these settings:

type=identify	Must be of type **identify**.
endpoint	Name of endpoint section to be identified if there is a match.
match	Comma-separated list of source IP addresses, networks, or hostnames to match against. SRV records are supported (unless the option **srv_lookups=no** is set.)

- **Endpoint** Specifies core SIP functionalities and their relationships to other sections, such as **Auth**, **AoR**, and **Transport**. This essentially describes the configuration of a SIP endpoint or a remote service, such as a phone or SIP Trunk service.

```
[1001]
type=endpoint
context=internal
disallow=all
allow=ulaw,alaw
transport=trans-udp
auth=1001-auth
aors=1001
```

Here are the explanations for these settings:

type=endpoint	Definition of the section type. Must be of type **endpoint**.
context=internal	Arbitrary name that specifies where sessions from the devices associated with this section will enter the dialplan.
disallow=all	Disables the use of all codecs.
allow=ulaw,alaw	Comma-separated list of one or more codecs to allow.
transport=trans-udp	Specifies the transport configuration to use.
auth=1001-auth	Specifies the authentication object associated with the endpoint.
aors=1001	Comma-separated list of AoRs to associate with endpoint.

TIP If your endpoints are behind a NAT router or firewall, relative to the Asterisk server, then you can add the following NAT configurations to your endpoint configurations:
`direct_media=no` Species whether media may flow directly between endpoints. Set to **`no`** to ensure Asterisk is in the media path.
`rtp_symmetric=yes` Set to **`yes`** to enforce that RTP must be symmetric.
`force_rport=yes` Configuration to force use of return port. Default is **`yes`**.

- **Registration** Configures outbound connection with another system, such as an Internet telephony service provider (ITSP). This connection is often referred to as SIP *trunking*.

```
[my-sip-trunk]
type=registration
transport=trans-udp
outbound_auth=mytrunk_auth
server_uri=sip:sip.example.com:5060
client_uri=sip:yyang@example.com
retry_interval=60
```

Here are the explanations for these settings:

`type`	Must be of type **`registration`**.
`transport`	Transport used for outbound registration.
`outbound_auth`	Authentication object to be used for outbound registration. This will authenticate your Asterisk system to the ITSP.
`server_uri`	SIP URI of the location of the server to register against (for example, sip:sip.example.com).
`client_uri`	Client username used when attempting outbound registration. This is the AoR to which you are binding a contact URI.
`Retry_interval`	Interval in seconds between retries if outbound registration is unsuccessful. The default is 60.

You can review the configurations of the **pjsip** objects by connecting to your Asterisk console and typing the following:

```
Server*CLI> pjsip show <object type>
```

For example, here we review the configuration of the **AoR** object:

```
Server*CLI> pjsip show aors
      Aor:  <Aor..............>  <MaxContact>
    Contact:  <Aor/ContactUri......> <Hash....> <Status> <RTT(ms)..>
==========================================================================
Aor:  1001                                          1
Contact: 1001/sip:1001@192.168.1.101:1024;transport=U 0114bd  Unknown  nan
Objects found: 1
```

Now that the SIP entities are defined, the next section introduces how to configure the actions that Asterisk will perform when interacting with these entities and channels.

The Dialplan: extensions.conf

At the heart of every PBX is its *dialplan:* the logic that determines which connections are made based on the number and pattern of the digits dialed for all incoming and outgoing calls. The dialplan is a scripting language specific to Asterisk and is one of the primary ways of instructing Asterisk to route and manipulate calls in a programmatic way.

The dialplan is configured in **/etc/asterisk/extensions.conf** and is organized in sections. The sections are used for static settings and definitions, as well as for executable dialplan components, and are referred to as *contexts*. The contexts are used to manage all assigned telephone numbers within the Asterisk system, and almost all functionalities of the Asterisk system are defined in the contexts, such as IVR menus, call forwarding, and hunt groups. Access and capabilities that are available to the clients and users are also defined within contexts, such as defining a group of phones that might be able to make outgoing calls and restricting another group to only internal calls. In general, a dialplan looks like this:

```
[general]
--> some settings go here

[globals]
--> definition of some global variables go here

[context_name_1]
--> extension 1, priority 1, application
--> extension 1, priority 2, application

[context_name_2]
include => context_name_1

--> extension 999, priority 1, application
--> extension 999, priority 1, application
```

Here are the explanations for these settings:

[general]	The first section in the **extensions.conf** file. The primary configuration options can be set here, including how Asterisk rewrites the **extension.conf** file when an extension is added or delete, whether you can rewrite the dialplan from the CLI, or how Asterisk behaves after completing a call.
[globals]	Optional section that can be used to specify your own variables in this context. Note that a global variable name is not case-sensitive, so **${MYVAR}** and **${mYvaR}** are the same. General format: ***variable_name => variable_value***
[context_name]	Context title. This is the entry point for calls from the devices and channels into the dialplan. For every incoming connection, Asterisk looks up the context defined for that device or channel and executes the sets of instructions, or extensions, defined under the section title.
include => context	You can include extensions from one context in another context. Asterisk will always try to find a matching extension in the current context first, and only follow the **include** statement to a new context if there isn't anything that matches in the current context.

Modules: modules.conf

Asterisk is built on modules. A module is a loadable component that provides a specific functionality, such as a channel driver, resource, codecs, or applications. The **modules.conf** file is the Asterisk Module Loader configuration file, where all of the modules to load or not to load from the modules directory (**/usr/lib64/asterisk/modules/**) at startup of Asterisk are configured.

As an analogy, all of these modules make up the organs and limbs of the Asterisk system. Though the **modules.conf** file is not strictly required for an Asterisk setup, the Asterisk system cannot provide any functionality without the modules.

Type the command **`module show`** to review the modules that are loaded in your Asterisk system. Note that your console output will probably be different from the following list.

```
server*CLI> module show
Module         Description                    Use Count  Status  Support Level
chan_sip.so  Session Initiation Protocol (SIP)  0  Running  core
codec_ulaw.so   mu-Law Coder/Decoder            0  Running  core
...<OUTPUT TRUNCATED>...
format_g729.so     Raw G.729 data               0   Running  core
res_rtp_asterisk.so  Asterisk RTP Stack         0  Running  core
res_xmpp.so       Asterisk XMPP Interface       0   Running  core
211 modules loaded
```

For our first PBX installation, we will accept the default configuration directives in the **modules.conf** file. Let's review our sample **module.conf** file to understand the inner workings of the module loading subsystem.

To view the **modules.conf** file, enter the following:

```
[master@server ~]$ sudo less /etc/asterisk/modules.conf
```

You will see the **[modules]** section is the only section in the **modules.conf** file. Table 22-5 summarizes the options available.

Option	Value/Example	Notes
`autoload`	`yes`	Directs Asterisk to load all modules that it finds in the **modules** directory, with the exception of those explicitly excluded from being loaded using the `noload` directive.
`preload`	`res_odbc.so`	Indicates that a module should be loaded at the beginning of the module load order.
`load`	`chan_pjsip.so`	Defines a module that should be loaded. This directive is only relevant if `autoload` is set to `no`.
`noload`	`chan_alsa.so`	Defines a module that should not be loaded. This directive is only relevant if `autoload` is set to `yes`.
`require`	`chan_pjsip.so`	Does the same thing as `load`; additionally, Asterisk will fail if this module fails to load for some reason.
`preload-require`	`res_odbc.so`	Does the same thing as `preload`; additionally, Asterisk will exit if this module fails to load.

Table 22-5 Modules.conf [modules] Section

NOTE With the exception of **autoload**, all the options in Table 22-5 may be specified more than once.

TIP Asterisk can be fussy about its modules. To avoid the wrath of this fussiness, it's best for newcomers to Asterisk to err on the side of being overly generous by setting **autoload=yes**. As you get more familiar with Asterisk, you can use the **noload** directive to explicitly instruct Asterisk not to load unneeded modules.

Asterisk Network, Port, and Firewall Requirements

VoIP relies on your network being optimized for connectivity and good sound quality. Most of the sound quality issues that you will experience with the Asterisk system will be network related. If it is at all possible, all VoIP-related traffic on the network should be segmented away from other day-to-day network traffic. This can be done through the use of a virtual local area network (VLAN) and/or through the use of dedicated networking equipment. All relevant network infrastructure equipment should be optimized to prioritize VoIP traffic.

This section assumes that your Linux server has some default firewall rules in place to protect the system. To proceed with the Asterisk setup and to make sure the relevant asterisk components are reachable on the network, we need to ensure that the firewall subsystem on the server allows access to the necessary ports required by Asterisk.

Table 22-6 summarizes some default ports required by Asterisk. If needed, the ports can be changed in the various configuration files specific to the protocols and applications.

Configuring the Local Firewall for Asterisk

For the exercise, we will configure the firewall to allow traffic on port 5060 for SIP signaling and ports 10000 through 20000 for the RTP (media) traffic. You can also narrow the range of RTP ports in the **/etc/asterisk/rtp.conf** file.

On Red Hat–like distros such as Fedora, Centos, or RHEL, use the **firewall-cmd** command to open the required ports (5060 and 10000–20000) by running the following:

```
[master@fedora-server ~]$sudo firewall-cmd --permanent --add-port=5060/udp
[master@fedora-server ~]$sudo firewall-cmd --permanent --add-port=10000-20000/udp
[master@fedora-server ~]$sudo firewall-cmd --reload
```

Protocol	Port Number	Transport
SIP	5060	TCP/UDP
RTP	10000 to 20000	UDP
MGCP	2727	UDP
SCCP	2000	TCP
XMPP	5222	TCP

Table 22-6 List of Common VoIP Protocol Port Numbers

For Debian-based systems such as Ubuntu, you can use UFW (Uncomplicated Firewall). Type the following commands:

```
[master@ubuntu-server ~]$ sudo ufw allow 5060/udp
[master@ubuntu-server ~]$ sudo ufw allow 10000:20000/udp
```

Be aware that the preceding commands will allow all UDP traffic from *any* (possibly untrusted) source address to access port 5060 and the entire port range 10000 through 20000 on your server! On a production server, you will want to be stricter and more deliberate when crafting your firewall rules.

Configuring the PBX

Now that you have a fully installed Asterisk server and verified that it is running, you will proceed to configure your first PBX. There are a number of technologies supported by Asterisk to configure both the local extensions and the connection to the outside world, such as SIP, XMPP, IAX2, and so on.

In this chapter we will focus on connectivity using the SIP protocol. Configuring a SIP phone to work with Asterisk is not complex. However, because there are so many options possible in both Asterisk and the configuration of the particular telephone set or softphone, things can get confusing. What we are going to do, therefore, is give you the bare-bones basics. At the end of this chapter, you will be able to configure the VoIP phone selected for the exercises (and be equipped with enough background knowledge to configure most SIP phones available today). We are not claiming that this is the best way, or only way, but we hope it will form a foundation from which to take a basic configuration and tweak things until you get the solution you need.

NOTE We did not use any GUI implementation in this chapter because we want to expose you to the nuts and bolts of the Asterisk project. You may decide to pick any GUI of your choice afterward. However, in production environments, using a GUI is discouraged on Asterisk servers because it is a waste of resources and can hamper performance of your VoIP platform.

We'll build up our PBX to a very simple configuration consisting of only two SIP phones and a connection to the public switched telephone network (PSTN) to make phone calls to regular mobile phones and landlines, as shown in Figure 22-3. This minimal configuration should provide a good foundation to demonstrate how incoming and outgoing calls are routed through the dialplan.

The configuration in Figure 22-3 consists of the following items:

- One Asterisk server
- Two SIP extensions: Alice, at extension 1001, and Bob, at extension 1002
- One outside connection through an ITSP, Twilio, using its Elastic SIP Trunk
- One local phone number from the ITSP to receive calls from the PSTN (for example, with caller ID 1-408-555-1212)

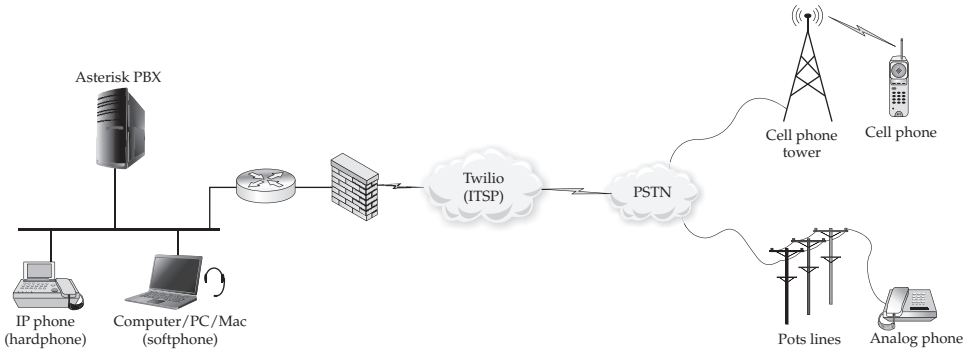


Figure 22-3 Your first VoIP PBX configuration

TIP We've bundled working copies of the configuration and/or script files used in this chapter at the following URL to save you some typing:
http://linuxserverexperts.com/8e/chapter-22.tar.xz

The files are named descriptively to match the scenario. If you choose to follow along using this bundle, just download the archive, extract its contents (decompress it), and then rename and copy the file(s) to the proper directories on your local system.

Local Extensions

The term *local extensions* refers to telephone sets, or endpoints, attached to the Asterisk server. Typical configurations enable the local extensions to communicate with each other at a reduced cost (usually free) and also allow calls out to the PSTN, if the Asterisk server has external connection to the PSTN.

We will set up the local extensions in the following sections. The entire configuration and setup process involves configuring SIP extensions on the server in the **pjsip.conf** file, configuring the dialplan on the server in the **extensions.conf** file, downloading and installing a softphone (SIP/VoIP client), and configuring the softphone for the local extensions. Finally, we'll run through a test scenario where we dial from one extension to another.

PJSIP Channel Configuration (pjsip.conf)

Let's back up the original **pjsip.conf** config file to **pjsip.conf.original**:

```
[master@server ~]$ sudo mv /etc/asterisk/pjsip.conf{,.original}
```

Use your favorite text editor to create a new **/etc/asterisk/pjsip.conf** with the following configuration. We use the **nano** editor in this example:

```
[master@server ~]$ sudo nano /etc/asterisk/pjsip.conf

[global]
type=global
```

```
[transport-udp]
type = transport
protocol = udp
bind = 0.0.0.0:5060

;===============EXTENSION 1001
[1001]
type=endpoint
context=internal
disallow=all
allow=ulaw
transport=transport-udp
auth=1001-auth
aors=1001
callerid=Alice <1001>

[1001-auth]
type=auth
auth_type=userpass
password=1001
username=1001

[1001]
type=aor
max_contacts=1
remove_existing = yes

;===============EXTENSION 1002
[1002]
type=endpoint
context=internal
disallow=all
allow=ulaw
transport=transport-udp
auth=1002-auth
aors=1002
callerid=Bob <1002>

[1002-auth]
type=auth
auth_type=userpass
password=1002
username=1002

[1002]
type=aor
max_contacts=1
remove_existing = yes
```

Save your changes to the file and quit the editor when you are done.

This configuration will enable extensions 1001 and 1002 and make them available for the SIP devices. **`auth=`** is used for the endpoint as opposed to **`outbound_auth=`** since we want to allow inbound registration for this endpoint. **`max_contacts=`** is set to something nonzero, as we want to allow contacts to be created through registration.

We used **callerid=** to set the caller ID information for the endpoint. Caller ID is in the format **Name <Number>**, or only **<Number>**.

The contact option (IP address and port) in the **aor** section will be configured dynamically from the registration information of the SIP endpoint.

Connect to the Asterisk console and reload the PJSIP configuration:

```
server*CLI> pjsip reload
```

TIP You can execute Asterisk commands without always first having to connect to the native Asterisk console. To do this, use the **-rx** option followed by the sub-command that you want to execute. For example, here is the full command to run the **"pjsip reload"** sub-command:

```
# asterisk -rx "pjsip reload"
```

And similarly, to show the current dialplan, you would run the following:

```
# asterisk -rx "dialplan show"
```

Now list the available SIP endpoints and you will see the two SIP phones we just configured in the **pjsip.conf** file in the console output. You will notice that **State** is **Unavailable** to indicate that the device is offline, and no endpoint is registered.

```
server*CLI> pjsip list endpoints
 Endpoint:  <Endpoint/CID.........>  <State.....>  <Channels.>
============================================================
 Endpoint:  1001                    Unavailable   0 of inf
 Endpoint:  1002                    Unavailable   0 of inf
Objects found: 2
```

Dialplan Configuration (extensions.conf)

The sample dialplan that ships with the Asterisk package contains many interesting parameters/settings in its default and unmodified state, but since these are not needed for this exercise, we'll rename/back up the original file and create a new one from scratch:

```
[master@server ~]$ sudo mv /etc/asterisk/extensions.conf{,.original}
```

Then use a text editor to create a new **extensions.conf** file with the following contents:

```
[master@server ~]$ sudo nano /etc/asterisk/extensions.conf
[general]
;
[internal]
;Dialing between internal phones (extensions) - Alice and Bob
;the extensions are four digits and must start with "10" to match this context.
exten => _10XX,1,Dial(PJSIP/${EXTEN},60,t)
exten => _10XX,n,Hangup()
```

Save your changes and exit the text editor.

Connect to the Asterisk console and reload the configuration:

```
server*CLI> dialplan reload
```

Congratulations! You have set up and configured an Asterisk system that is capable of allowing phone conversations between two SIP devices. This simple setup is not dependent on any other external services or infrastructure beyond your private network or LAN.

However, there is one more component missing—the telephones!

Configure the Phone Sets (Softphone) for the Local Extensions

The Asterisk system works with many different types of phone endpoints, which can be either IP phones or analog phones attached to an ATA. All of these phone types have the same basic SIP configuration, and once you understand the fundamental concept of SIP configurations, you will be able to configure almost any phone set.

A principle to follow is to start your configurations in the simplest manner possible, as we have done in previous sections, and leave the Asterisk configuration static. Then move on to configure the phone sets and carry out tests. If you have followed the steps up to this point in the chapter, you should have a working system configured and now can focus on getting your phone sets to work so that you can make and receive calls. You will be in a far better position to begin experimenting with different settings. We have seen too much suffering (including ours!) that stems from overly complicated configurations, and we want it to end here and now!

You will need at least two phone endpoints to configure the extensions and to make and receive calls. For simplicity, we will use the free X-Lite softphone from CounterPath, Inc., for each of the extensions and install it on two different computers, smartphones, or tablets.

TIP In the configuration menus of the phone itself, you will want to look for fields that are labeled user name, auth name, authentication name, and so on. The thing to remember is that since you know that the Asterisk end of the equation is configured simply and correctly, you can experiment with the phone settings until you find a combination that works.

Download and Install the X-Lite Softphone

Follow these steps to obtain and configure the X-Lite softphones on each of the client systems or devices from which you want to run the SIP clients:

1. Launch your favorite browser and go to www.counterpath.com/x-lite-download/.
2. Click the Download button for the operating system of the computer on which you want to install the softphone.
3. Follow the instructions to install the software and then launch the X-Lite software.
4. Repeat Steps 1–4 on the second system that will be running the X-Lite software and acting as the second extension.

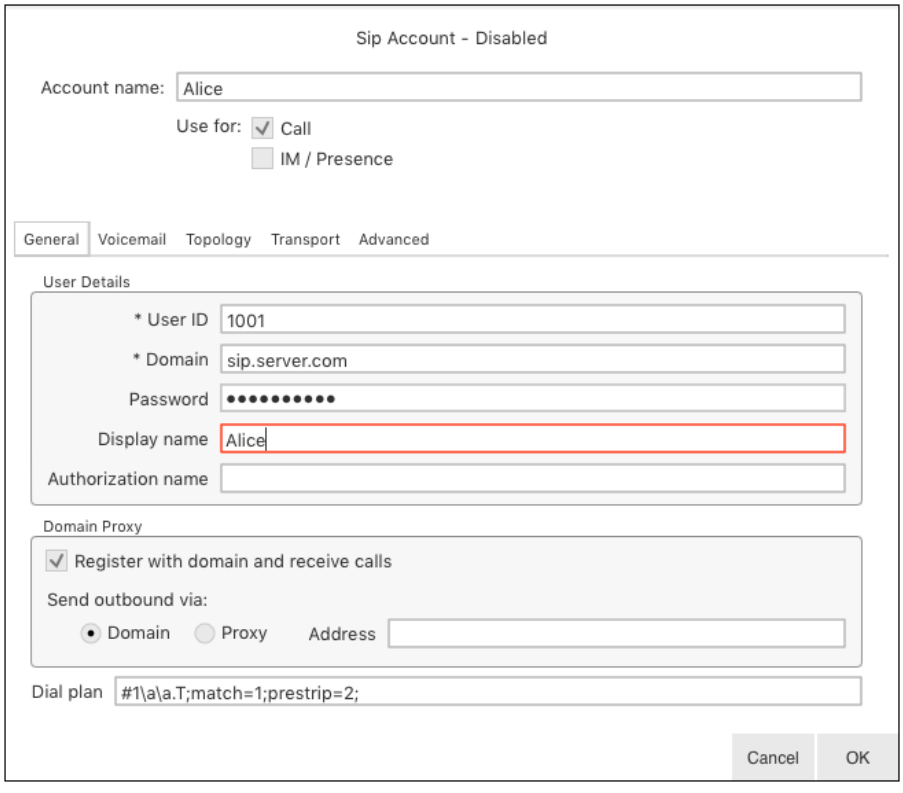
Configure the SIP Extensions

After installing the SIP clients as described, you'll need to configure the SIP extensions as follows:

1. Set up your SIP accounts. Go to Preferences → Account Settings.

2. Fill out the screen, shown next, with the following values:

   User ID: **1001**
   Domain: **<sip.server.com>**
   Password: **herSecret**
   Display name: **Alice**



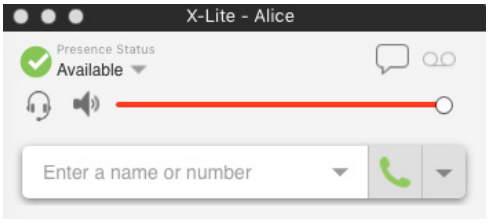
3. Click OK to save the configuration and attempt to connect to the Asterisk server.
4. Repeat Steps 1–3 to configure the second X-Lite SIP client (Bob). Substitute all the information for Alice (User ID:1001, Password: herSecret. Display name: Alice) with Bob's correct information wherever applicable (User ID: 1002, Password: hisSecret, Display name: Bob).

Before we continue to the next section, you can run the following commands on the Asterisk console to verify the configuration of the softphone and ensure that they both are registered correctly:

```
server*CLI> pjsip show endpoints
 Endpoint:  <Endpoint/CID.............>  <State...>  <Channels.>
...<OUTPUT TRUNCATED>...
==========================================================
 Endpoint:  1001                       Not in use    0 of inf
```

```
InAuth:  1001-auth/1001
Aor:  1001                                1
Contact:  1001/sip:1001@192.168.1.101:51266;rinstance 974 Unknown   nan
Transport:  transport-udp          udp      0       0     0.0.0.0:5060
...<OUTPUT TRUNCATED>...
server*CLI>
```

The X-Lite client will show the presence status as Available, as shown here:



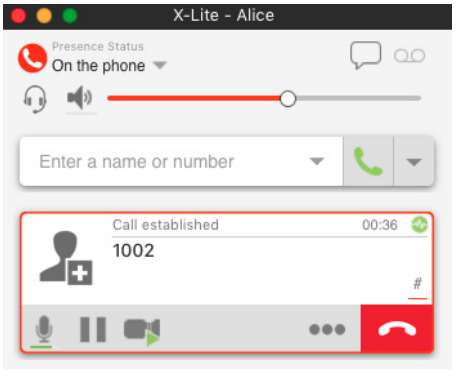
NOTE The status information may look different on your version of softphones (and hardphones).

Test Scenario: Dial from Alice (Extension 1001) to Bob (Extension 1002), First Phone Call

The reward for your hard work is to hear the first phone ring. So, let the fun begin!

For this scenario, we'll call Bob's extension from Alice's extension:

1. Launch both softphones, ensure that they are both registered to the Asterisk server, and check that the status is set to Available.
2. On Alice's softphone (1001), enter **1002** on the keypad.
3. You should hear and observe Bob's phone ringing and showing the caller ID of Alice as the caller, as shown next:



4. Ensure you are able to speak into one of the phones and get good audio from the other phone, and vice versa.
5. Hang up. Then make a few more test calls.

Outside Connection (VoIP Trunking)

Outside connection is the ability to be able to communicate with devices outside of your private network or LAN. This capability, generally known as VoIP trunking, extends the practical uses of the server, because it not only allows users to make calls to others outside of the private local network, but it also allows your internal users to be reachable from outside. Asterisk supports a number of different technologies to interconnect to outside networks, and these fall into two main categories (described next) depending on whether the interconnectivity is digital or analog:

- **Connectivity using a VoIP gateway** If you own a PSTN line (regular telephone line provided by your local telecom provider, also known as plain old telephone service, or POTS), you can connect it to your Asterisk server to make and receive calls. Similar to the ATA, you will need a device to connect your analog PSTN line to your Asterisk system. This device is called a VoIP gateway or PSTN gateway; and works as a bridge between the VoIP network and the PSTN. Depending on where the voice traffic originates from, the VoIP gateway will convert the voice traffic into the proper form for receipt by the destination network (IP or PSTN). The VoIP gateway may be a stand-alone device connected to your LAN, such as the Cisco SPA122, or a PCI card installed in your Linux server from a manufacturer such as Digium, Sangoma, AudioCodes, or Obihai.
- **Connectivity through an Internet telephony service provider (ITSP)** It is still possible to connect to outside networks even if you do not own a traditional PSTN line. This is achieved by interconnecting to one of an increasing number of ITSPs. Asterisk supports various protocols for interconnection, including SIP and XMPP. ITSPs provide various services, including, but not limited to, outbound calling, inbound calling, or both. Your choice of ITSP will ultimately depend on the services you require, your budget, and the call quality offered by the ITSP. Most ITSPs will provide sample Asterisk configuration to help get you connected to their service as quickly as possible.

In the next section, we will walk you through a common setup that will help you get started with SIP technology. The configuration may differ slightly depending on which ITSP you use and its specific implementation.

Regardless of the method you use to achieve outside connectivity, Asterisk treats these methods as channels, much like the PJSIP channels we set up earlier, and controls them through the **dialplan**.

Trunking Using Twilio Elastic SIP Trunks

Twilio Elastic SIP Trunks provide flexible, enterprise-ready, global connectivity for your VoIP infrastructure. Through the Twilio console, you can sign up and purchase phone numbers, referred to as Direct Inward Dialing numbers, or simply DIDs, from different area codes within the U.S., as well as DIDs from most other countries, and configure them to route calls to and from your Asterisk system.

Configuring Twilio Elastic SIP Trunks

Let's walk through the steps needed to configure SIP Trunks using Twilio:

1. To get started, you will need a new or existing Twilio account. See the Twilio SIP Trunking site for more details (https://www.twilio.com/sip-trunking).
2. Once you have signed up, log into the Twilio Console (https://www.twilio.com/login).
3. Click the "All Products and Services" icon (…) and then click the Elastic SIP Trunking icon.
4. From the Elastic SIP Trunking Dashboard screen, click the Get Started button or select the Getting Started menu/link.
5. Click the Create a SIP Trunk button to create a new SIP Trunk.

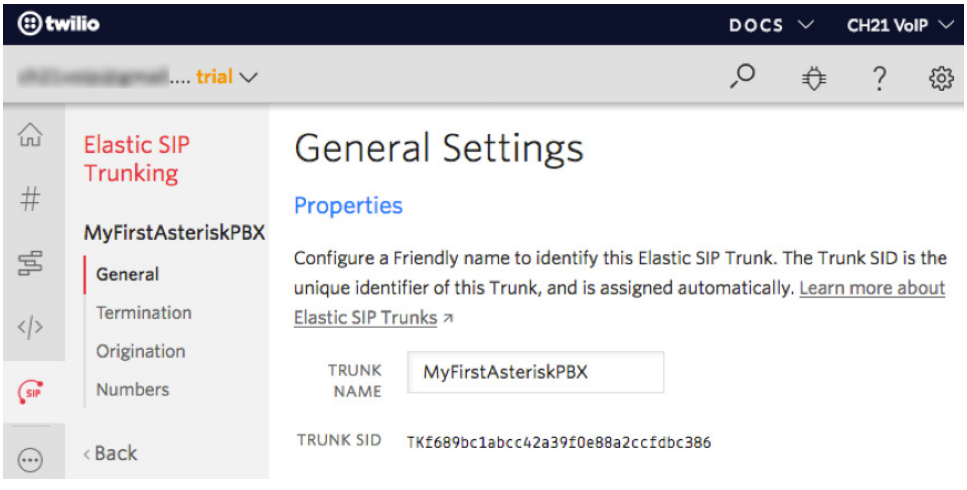
NOTE You can also create the SIP Trunk by selecting the Trunks menu item and clicking the **Create New SIP Trunk** button.

6. Add a friendly name to identify this trunk and then click the Create button.

At this point, we've created the shell of the trunk. Let's now configure the trunk.

Trunk Configuration

The Twilio SIP Trunk supports incoming and outgoing calls between your Asterisk server and the PSTN. Settings for outgoing calls will be configured under Termination, and incoming calls via your Twilio provisioned numbers will be configured under Origination. You'll also need to associate Twilio numbers with your trunk under the "Numbers" page. We'll see these next.



Termination

1. From the menu on the left, under the Elastic SIP Trunking section, select Termination.
2. In the Termination SIP URI field, enter a unique URI. The interface would verify and notify you if your selected URI is available. For our exercise, we entered **ch22voip** so that our Termination SIP URI becomes **ch22voip.pstn.twilio.com**.
3. Select and configure at least one authentication scheme to use to authenticate your Asterisk server to Twilio—either IP ACL or Credentials List, or both. The IP ACL restricts termination of calls to the IP addresses specified. Credentials List provides a username and password to authenticate your requests to the Termination URI.

   In IP access Control Lists, click the + button to create a new access control list.

   Properties FRIENDLY NAME: <example: MyPBXACL>
   IP ADDRESS: <your_server_NAT_public_IP_Address>
   IP Address FRIENDLY NAME: <example: MyFedoraAsteriskServer>
4. Click the Create ACL button.

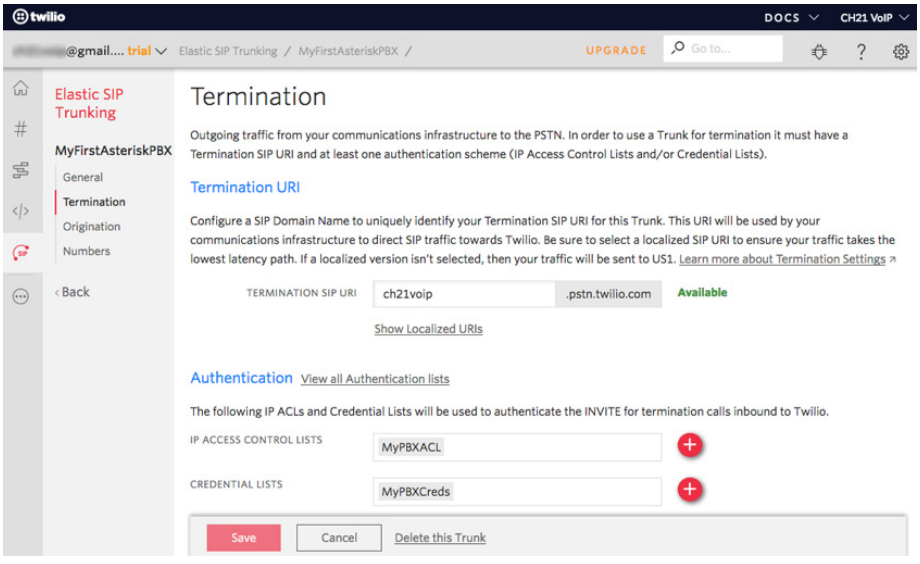
NOTE In the future, you may have other Asterisk PBXs you want to allow to communicate with your SIP termination URI. Simply add their public or NAT IP addresses and a corresponding friendly name to the IP ACL.

5. Click the + button to create Credential Lists. Enter a friendly name. Select a unique username and strong password (minimum length of 12 characters, at least one number, uppercase character, and lowercase character). Configure the following fields:

   FRIENDLY NAME: <your_friendly_name>
   USERNAME: <your_unique_username>
   PASSWORD: <your_credential_password>
6. Click the Create button when finished.

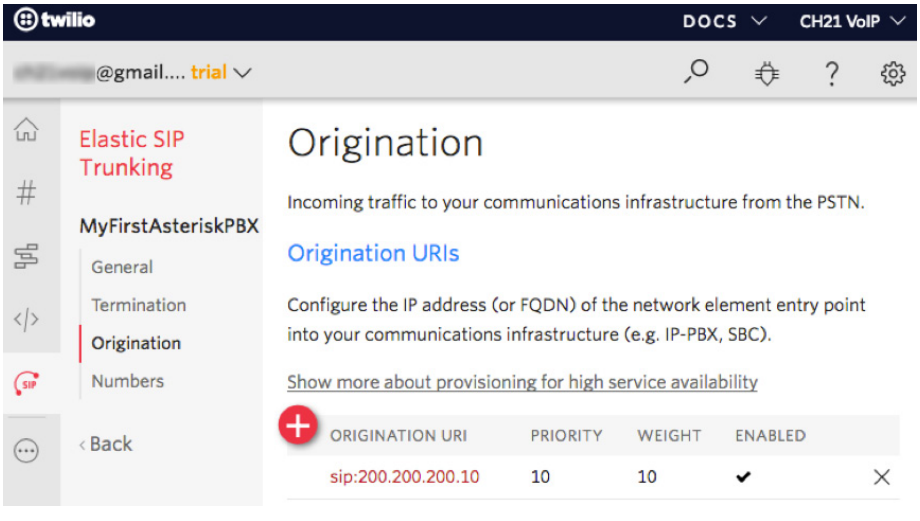
NOTE You can create multiple credentials to authenticate with your Termination SIP URI. The **username** and **password** credentials will be used in the **auth** section in the **pjsip.conf** configuration file of the Asterisk server.

7. Scroll down to the bottom and click Save. You should receive confirmation you have successfully updated your SIP Trunk.



Origination

1. From the menu on the left, under the Elastic SIP Trunking section, select Origination.
2. Click Add New Origination URI to create a new Origination URI that defines the entry point URI (or public or NAT IP address; for example, 200.200.200.10) to your Asterisk PBX. Also, set the Priority and Weight options to 10.
3. Check the ENABLED box and then click the Add button.



Assign Telephone Number to the Elastic SIP Trunk

At this stage, we have completed the internal Twilio configuration; the next step is to configure and assign the telephone number that will be called to reach your Asterisk PBX.

You have the option to purchase a new number from Twilio, to assign an existing Twilio number, or to "port" a number you own on another network to Twilio and assign that number to your trunk, or any combination of these three options. For this exercise, we will buy a new number.

CAUTION Purchasing a number from Twilio is *not* free. At time of writing, each number costs $1 per month. If you purchase a number, you may cancel it at any time to stop payment by releasing the number in the Number menu option.

1. From the menu on the left, click Numbers.
2. Click on the Buy a Number button.
3. Search for a number that meets your criteria. You can search based on a specific number or parts of a number, by location, or by capabilities.
4. Click the Buy button next to the number you select.
5. Then click Buy This Number to confirm your purchase of the number.

Your purchased number should now be associated with your SIP Trunk.

IP Address Whitelist

In the next steps, we will configure your network infrastructure to ensure that Twilio has connectivity to your Asterisk PBX and vice versa.

1. Click <Back to go back to the Elastic SIP Trunking menu, and then select Networking Info.
2. Scroll down to the IP Address Whitelist section and note the Twilio IP addresses and port ranges in the region(s) that you have phone numbers.
3. Whitelist these in your environment at your firewalls and/or ACLs. These IP addresses are the gateways that calls will originate from for those numbers.
4. Since we elected to authenticate by IP address, in addition to the credentials, we must also configure the signaling IPs in the Twilio **identify** section in the **pjsip.conf** file.

This completes the Twilio-specific configuration. We will integrate your Twilio number service to your Asterisk server in the next sections.

SIP Trunk Configuration (pjsip.conf) The Twilio Elastic SIP Trunk, as the name indicates, utilizes the SIP protocol for signaling. We will add the trunk configurations to the **pjsip.conf** file. Similar to the configurations of the internal extensions, we will configure the **auth**, **aor**, **endpoint**, and **identify** sections for the Twilio trunks.

1. We'll start by taking a backup of the working internal extensions configuration so we can roll back to our last known working configuration if we run into issues:

```
[master@server ~]$ sudo  cp /etc/asterisk/pjsip.conf{,.working_config}
```

2. Edit the **pjsip.conf** file and add the following contents:

```
[master@server ~]$ sudo nano /etc/asterisk/pjsip.conf
```

3. Configure the Twilio credentials in the **auth** section by adding the following:

```
;===============TRUNK
;
[twilio-auth]
type = auth
auth_type=userpass
username = your_twilio_trunk_credentials_username
password = your_twilio_trunk_credentials_password
```

The username and password are the credentials created in the Credentials Lists in the Twilio console.

4. Configure the Termination SIP URI you configured in the Twilio console in the **contact** field of the Address of Records (**aor**) section:

```
[twilio-aor]
type = aor
qualify_frequency = 60
contact=sip:<your_termination_SIP_URI>:5060   ; our termination SIP
URI
```

The **contact** field is the Twilio target that outgoing calls (SIP INVITEs) will be sent to. In our example, we configured **mysip.pstn.twilio.com**. The **qualify_frequency** is the interval at which SIP NOTIFY messages will periodically be sent to the Twilio server to determine both its availability and the latency of its replies.

5. Specify the signaling IPs in Step 4 in the IP Address Whitelist section to allow Asterisk to recognize the endpoint when a call comes in:

```
[twilio-na-us]
type = identify
endpoint = twilio
match=54.172.60.0,54.172.60.1,54.172.60.2,54.172.60.3
match=54.244.51.0,54.244.51.1,54.244.51.2,54.244.51.3
```

TIP For North American gateways, you can also configure the **match** fields as follows:
match = *twilio-asteriskpbx.pstn.us1.twilio.com*
match = *twilio-asteriskpbx.pstn.us2.twilio.com*

6. Configure the characteristics of the Twilio endpoint in the **endpoint** section:

```
[twilio]
type = endpoint
transport= transport-udp
context = pstn-in
outbound_auth = twilio-auth
aors = twilio-aor
disallow = all
allow = ulaw
```

7. Enable the trunk configuration by reloading the **pjsip** module in the Asterisk console:

```
server*CLI> pjsip reload
```

8. While still at the Asterisk console, review the configuration:

```
server*CLI> pjsip list endpoints
 Endpoint:  <Endpoint/CID........>  <State.....>  <Channels.>
============================================================
 Endpoint:  1001/1001                Not in use    0 of inf
 Endpoint:  1002/1002                Not in use    0 of inf
 Endpoint:  twilio                   Not in use    0 of inf
Objects found: 3
```

Dialplan Configuration: Incoming Calls (extensions.conf) Now that you have configured your Twilio account, obtained an inbound U.S. number from Twilio (say, 408-555-1212), and have successfully configured your connection to the Twilio server, the Twilio Elastic SIP Trunks service will forward any calls to your Twilio number to your Asterisk system using the configured Termination SIP URI. The next step is to describe how your Asterisk system will handle the call when it is received in the dialplan.

Let's configure the dialplan to receive an incoming call from Twilio and route it to Alice, at the SIP phone you created with extension 1001.

Edit your dialplan and insert the following at the end of the file:

```
[master@server ~]$ sudo nano /etc/asterisk/extensions.conf

[pstn-in]
exten => _+1NXXXXXXXXX,1,NoOp()
 same => n,Dial(PJSIP/1001,60,t)
 same => n(end),Hangup()
```

The **exten => _+1NXXNXXXXXX** matches a full E.164-formatted U.S. number. Once the call is received, Asterisk dials extension 1001 and then bridges the call when answered by Alice.

TIP If you have numbers from other regions, outside North America, adjust this match criteria accordingly. For regions that have numbers with variable lengths, you can also create a match pattern using the "**.**" wildcard, as follows:

exten => _+44X.

This pattern will match any number that starts with +44[0-9]….

E.164: The International Public Telecommunication Numbering Plan

E.164 is an international standard that ensures each device on the PSTN has globally unique number. E.164 numbers are formatted [+] [**country code**] [**subscriber number, including area code**] and can have a maximum of 15 digits.

Dialplan Configuration: Outgoing Calls to PSTN (extensions.conf) The Twilio network requires dialed numbers to be in E.164 format.

Edit your dialplan and insert the following at the end of the **/etc/asterisk/extensions.conf** file under the **[outgoing]** section:

```
[master@server ~]$ sudo  nano /etc/asterisk/extensions.conf
[outgoing]
;E.164-formatted dialed number
exten => _1NXXNXXXXXX,1,NoOp()
 same => n,Set(CALLERID(all)="YYang" <+14085551212>)
 same => n,Dial(PJSIP/+${EXTEN}@twilio)
 same => n(end),Hangup()
```

If you want to support the North American Numbering Plan (NANP), which allows 7- and 10-digit dialing, in addition to the 11-digit dialing, then add the following to the **[outgoing]** context to match 7- and 10-digit numbers and append **+1** to the **Dial()** application:

```
;10-digit formatted dialed number
exten => _NXXNXXXXXX,1,NoOp()
 same => n,Set(CALLERID(all)="YYang" <+14085551212>)
 same => n,Dial(PJSIP/+1${EXTEN}@twilio)
 same => n(end),Hangup()
```

And in the same vein, add the following configuration to also support 7-digit dialed numbers. Append +1<your_area_code> to the **Dial**() application. In this example, the area code is 408:

```
;7-digit formatted dialed number
exten => _NXXXXXX,1,NoOp()
 same => n,Set(CALLERID(all)="YYang" <+14085551212>)
 same => n,Dial(PJSIP/+1408${EXTEN}@twilio)
 same => n(end),Hangup()
```

Finally, we need to allow the internal phones to use the twilio trunk to make outgoing calls. Enable this by adding the following to the **[internal]** context:

```
[internal]
; dialing from internal phones to pstn via Twilio
include => outgoing
```

Save all the changes to the files you edited and reload all the configurations, like so:

```
[master@server ~]$ sudo asterisk -rx "dialplan reload"
```

Test Scenario: Dial from a Phone Extension to the PSTN

Now make a call from either Alice's or Bob's softphone to the PSTN:

1. Launch Alice's (or Bob's) softphone.
2. Dial a PSTN routable number—you can dial your mobile or home phone number.

3. You will hear a ringback tone coming from the softphone and the dialed number will start ringing. If the ITSP supports custom caller ID, the caller ID displayed on the dialed number will be the caller ID you set with the **`CALLERID(all)=`** option. Otherwise, the caller ID will be the DID you configured into the system.
4. Ensure you are able to speak into one of the phones and get good audio from the other phone, and vice versa.
5. Hang up. Then make a few more test calls.

Test Scenario: Inbound Call from the PSTN

The final scenario is to call the DID from a PSTN number to ring to the configured extension:

1. Launch Alice's (or Bob's) softphone.
2. Dial the assigned DID from a mobile phone or landline:
3. You will hear a ringback tone coming from your mobile phone, and Alice's softphone will start ringing and display the caller ID of the caller.
4. Answer Alice's extension and ensure you are able to speak into one of the phones and get good audio from the other phone, and vice versa.
5. Hang up. Then make a few more test calls.

Asterisk Maintenance and Troubleshooting

Every effort has been made to ensure you do not experience any issues with the preceding examples, but, *hey*, stuff happens, interfaces change, and modules are deprecated every day. In real-world scenarios, you may run into issues originating from the network, host server, Asterisk configurations, or even issues and service outages from the VoIP provider! This section introduces some commands and tips to help you troubleshoot common problems.

Asterisk CLI Commands

Asterisk CLI commands follow this general syntax:

```
<module name>  <action type>  <parameters>
```

For example:

- **`pjsip list endpoints`** Returns a list PJSIP endpoints
- **`core set verbose xx`** Sets level of verbose chattiness

Helpful CLI Commands

To list the CLI commands available in the Asterisk console, simply type the following:

```
server*CLI> core show help
```

or

```
server*CLI> help
```

Both will list valid CLI commands available on your server.

The following shows help on a specific command:

```
server*CLI> help [command]
```

TIP The Asterisk CLI supports command-line completion on all commands, including many arguments. To use it, simply press the TAB key at any time while entering the beginning of any command. If Asterisk can complete the command unambiguously, it will do so; otherwise, it will complete as much of the command as possible. Additionally, Asterisk will print a list of all possible matches.

Next are some often-used commands for managing and troubleshooting Asterisk:

- **`core restart now`** Immediately restarts Asterisk. You will exit the CLI and be returned to the Linux prompt.
- **`core stop now`** Immediately stops Asterisk and drops back to the Linux prompt.
- **`core show calls [uptime]`** Displays information on calls.
- **`core show channels [concise|verbose|count]`** Displays calls information on channels.
- **`core show channel`** Displays calls information on a specific channel.
- **`dialplan show`** Displays all of the active (in-memory) dialplan. This includes, but is not limited to, the configuration contained in **/etc/asterisk/extensions.conf**.
- **`channel request hangup`** Requests a hangup on a given channel.
- **`channel request hangup all`** Requests a hangup on all active channels.
- **`core reload`** Globally reloads the configuration files.
- **`dialplan reload`** Reloads extensions and *only* extensions.
- **`pjsip reload`** Reloads SIP configuration.
- **`pjsip show channel`** Shows detailed PJSIP channel info.
- **`pjsip show endpoints`** Shows PJSIP endpoints.
- **`pjsip show contacts`** Shows PJSIP contacts.

Common Issues with VoIP

We've equipped you with the basics of setting up and configuring a simple VoIP server in this chapter. However, many things can still go wrong or prevent you and your users from getting the best VoIP experience possible. Even more annoying is that some of the factors causing this are usually out of your control and hard to pinpoint.

After all of your (and our) best efforts, you may find that the issues are simply because the planets are not properly aligned. Needless to say, this chapter can't help in cases where Neptune is not properly aligned with Saturn, but you can at least equip yourself with the knowledge about some of the technical issues (Quality of Service, latency, jitter, and so on) discussed next.

Quality of Service

It should be noted that no matter how well tuned you think your network is, and no matter how good the statistics say your calls are, it is the subjective perception of call quality that is important to the end user. Quality of Service, or QoS, refers to the perceived quality of a call and the overall performance of a VoIP network.

To quantitatively measure QoS, several related aspects of the network service are often considered, such as packet error rates, bandwidth, throughput, transmission delay, availability, jitter, and so forth.

Latency

Latency is the delay in the delivery of data packets across a network. As a rule of thumb, a normal flow of conversation is possible if you can deliver the sound produced by the speaker to the listener's ear within 150 milliseconds. Packet delays exceeding 250 milliseconds make normal conversation increasingly awkward and frustrating, and usually causes the communicating parties to start to talk on top of each other. Beyond 500 milliseconds, the communication becomes unintelligible.

Packet Loss

In addition to getting digitized voice packets to the destination on time, it is essential to ensure that the transmitted information arrives intact. If *packet loss* occurs in a network, the receiving endpoint will not be able to completely reproduce the sampled audio, which results in gaps in the data and is heard as static or, in severe cases, results in entire missed words or sentences. Packet loss of as low as 1 percent can severely impede VoIP calls using a G.711 codec. Ideally, there should not be any packet loss on the network.

Jitter

Jitter is defined as a variation in the delay of received packets. It is used to indicate when data packets arrive either ahead of or behind a standard clock cycle due to network congestion, improper queuing, or configuration errors. If jitter is too large in a data stream, some packets are discarded and the receiving endpoint may hear dropouts in the audio.

Some devices are QoS aware and can compensate for the jitter that is encountered by introducing a *jitter buffer*. The jitter buffer stores these packets and then plays them out in a steady stream to the digital signal processors (DSPs), which can then convert them back to an analog audio stream.

Out-of-Order Delivery

Data packets routed through a network may take different routes, each resulting in a different delay. As a result, the packets may arrive in a different order from the order in which they were sent. The out-of-order packets will result in garbled audio. To correct this condition, there are special network protocols responsible for rearranging out-of-order packets before delivery to the receiving device. The extra processing done by this rearranging can introduce latency into the network.

SELinux

Granted, SELinux is not specifically a VoIP issue, nor is it even a bad thing—when it is configured properly. However, when starting out with Asterisk on distros that implement SELinux and whenever things are not working as expected, we recommend disabling SELinux.

Summary

This chapter discussed the fundamentals of Voice over Internet Protocol (VoIP) and described some of the most common types of VoIP implementations, which include the use of analog telephone adapters (ATAs) and IP phones to directly connect to a VoIP service.

We discussed a few different types of VoIP implementations and used the open source Asterisk software to describe an implementation of VoIP. The exercises in the chapter were designed to walk you through basic configurations of the Asterisk system, showing you the steps necessary to set up a fully functional VoIP system that enables you to make calls between two endpoints and communicate with phone numbers in the public network (PSTN).

There is a lot more that can be discussed about the subject of VoIP and Asterisk than we can fit in one chapter; however, we hope we've armed you with the basic knowledge and skills required to be creative and expand upon to build more advanced features and capabilities.

You are now fully positioned to set up your VoIP server to compete with major providers such as AT&T, Bell Canada, Vodafone, MTN, Orange, Deutsche Telekom, Telstra, Verizon, Telefónica, Vonage, Lingo, magicJack, and so on!

CHAPTER

23 Secure Shell (SSH)

In Chapter 16, we discussed techniques (and considerations) for securing your Linux system. Our discussions emphasized the importance of limiting network-based exposure to your system to the bare essentials. But what if you need to perform system administrative duties remotely? How can you reap the benefits of a truly multiuser system if you can't easily and securely log into it?

Secure Shell (SSH) was developed to tackle the issue of secure remote logins. SSH is a suite of network communication tools that are collectively based on an open protocol/standard that is guided by the Internet Engineering Task Force (IETF). It allows users to connect to a remote server just as they would using Telnet, rlogin, FTP, and so on, except that the session is 100 percent encrypted. Someone using a packet sniffer merely sees encrypted traffic going by. Should they capture the encrypted traffic, decrypting it could theoretically take a long time!

In this chapter, we take a brief and general look at cryptography concepts. Then we take a grand tour of SSH, how to get it, how to install it, and how to configure it.

Understanding Public Key Cryptography

A quick disclaimer is probably necessary before proceeding: This chapter is by no means an authority on the subject of cryptography and, as such, is not the definitive source for cryptography matters. What you will find here is a general discussion as it relates to systems administration.

Secure Shell relies on a technology called *public-key cryptography.* It works similarly to a safe deposit box at the bank: You need two keys to open the box or at least multiple layers of security/checks have to be crossed. In the case of public-key cryptography, you need two mathematically related keys: a public one and a private one. Your public key can be published on a public web page, printed on a T-shirt, or posted on a billboard in the busiest part of town. Anyone who asks for it can have a copy. Any data encrypted with the public key can be decrypted with the private key. On the other hand, your private key must be protected to the best of your ability. It is this piece of information that makes the data you want to encrypt

truly secure. Any data signed (encrypted) with the private key can be verified (decrypted) with the public key. Every public key/private key combination is unique.

The actual process of encrypting data and sending it from one person to the next requires several steps. We'll use the popular "Alice and Bob" analogy and go through the process one step at a time, as they both try to communicate in a secure manner with one another. Figures 23-1 through 23-4[1] illustrate an oversimplified version of the actual process.

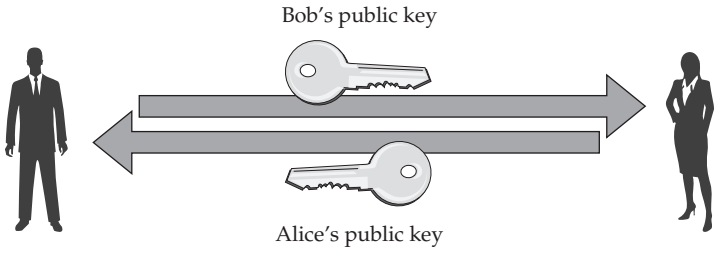


Figure 23-1 Alice and Bob exchange public keys via billboard, T-shirt, or over the network.

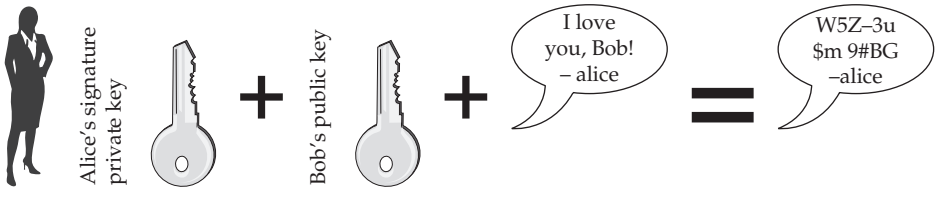


Figure 23-2 Alice uses Bob's public key, along with her private key, to encrypt and sign the data, respectively.

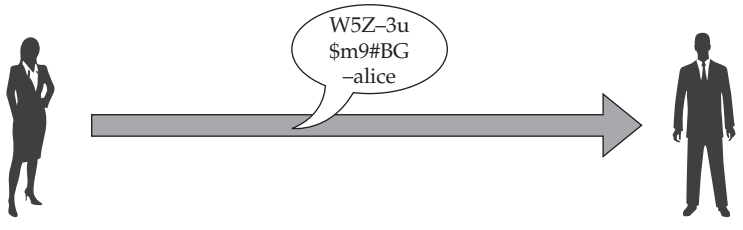


Figure 23-3 Alice sends the encrypted and signed data to Bob.

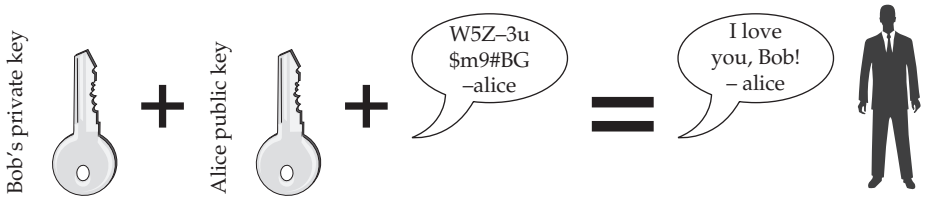


Figure 23-4 Bob uses Alice's public key, along with his private key, to verify and decrypt the data, respectively.

[1] Figures 23-1 through 23-4 are based on illustrations by Omolara Soyinka.

Looking at these steps, you'll notice that at no point was the secret (private) key sent over the network. Also notice that once the data was encrypted with Bob's public key and signed with Alice's private key, the only pair of keys that could decrypt and verify it were Bob's private key and Alice's public (signing) key. Thus, if someone intercepted the data in the middle of the transmission, he or she wouldn't be able to decrypt the data without the proper private keys.

To make things even more interesting, SSH regularly changes its session key. The session key is a randomly generated, symmetric key for encrypting the communication between the SSH client and server. It is shared by the two parties in a secure manner during SSH connection setup. In this way, the data stream gets encrypted differently every few minutes. Thus, even if someone happened to figure out the key for a transmission, that miracle would be valid for only a few minutes until the keys changed again.

NOTE SSH supports a variety of encryption algorithms. Public-key encryption just happens to be one of the more interesting methods of performing end-to-end encryption, and it's arguably the most secure!

Key Characteristics

So what exactly *is* a key? Essentially, a key is a large number that has special mathematical properties. Whether someone can break an encryption scheme depends on his or her ability to find out what the key is. Thus, the larger the key is, the harder it will be to discover it.

Low-grade encryption has 56 bits. This means there are 2^{56} possible keys. To give you a sense of scale, 2^{32} is equal to 4 billion, 2^{48} is equal to 256 trillion, and 2^{56} is equal to 65,536 trillion. Although this seems like a significant number of possibilities, it has been demonstrated that a loose network of PCs dedicated to iterating through every possibility could conceivably break a low-grade encryption code in less than a month.

For a key to be sufficiently difficult to break, experts usually recommended minimum key lengths. Keep in mind that every extra bit effectively doubles the number of possibilities. For example, if you really want to make the encryption solid, a key size of 2048 bits or higher is recommended for *RSA* type keys. Depending on the internal limitations of the key type (RSA, DSA, ECDSA, and so on), SSH can use various key lengths to encrypt your data.

The trade-off to using higher bit encryption is that it requires more math-processing power for the computer to churn through and validate a key. This takes time and, therefore, makes the authentication process a touch slower—but most people think this trade-off is worthwhile.

SSH Backstory (Versions)

The first version of SSH that was made available by DataFellows (now F-Secure) restricted free use of SSH to noncommercial activities; commercial activities required that licenses be purchased. The early closed source versions of SSH suffered from some serious security deficiencies. Some of these security issues might have been avoidable if the vendors of the software had made the source code open. This open access is especially important to cryptographic software, because it allows peers to examine the source code and make sure

there are no holes that might allow hackers to easily break the security. In other words, serious cryptographers do not rely on security through obscurity. Since the U.S. government has relaxed some of its encryption laws, work on the OpenSSH project has increased and it has become a popular alternative to some of the commercial versions of the SSH protocol.

Because the SSH protocol has become an IETF standard, other developers are also actively working on SSH implementations for other operating systems. There are many Linux/UNIX clients, Microsoft Windows implementations, macOS and iOS clients, Android clients, and even a Palm client (for people who like devices from the 1820s). You can find the version of OpenSSH discussed in this chapter at www.openssh.org.

OpenSSH and OpenBSD

The OpenSSH project was spearheaded by the OpenBSD project. OpenBSD is a version of the Berkeley Software Distribution (BSD) operating system (another UNIX variant) that strives for the best security of any operating system available. A quick trip to its web site (www.openbsd.org) shows that the organization has gone over two decades with only two remote exploits in its default installation. Unfortunately, this level of fanaticism over security comes at the expense of sometimes not having the most whiz-bang-feature-rich tools available, since anything added to their distribution must get audited for security first. The nature and focus of OpenBSD has also made it a popular foundation for firewalls.

The core of the OpenSSH package is considered part of the OpenBSD project and is thus simple and specific to the OpenBSD operating system. To make OpenSSH available to other operating systems, a separate group exists to make OpenSSH portable with each new release issued. Typically, this happens quickly after the original release.

NOTE Since this book focuses on Linux-based operating systems, you will frequently see versions of OpenSSH for this platform that are suffixed with the letter *p,* indicating that they have been *ported.*

Alternative Vendors for SSH Clients

The SSH client is the client component of the SSH protocol suite. It allows users to interact with the service(s) provided by an SSH server daemon.

These days, people work within heterogeneous environments, and it's impossible to ignore all the Windows 20**/7/8/10 and macOS systems out there. To allow these folks to work with a *real* operating system (Linux, of course!), there must be a mechanism in place for logging into such systems remotely. Virtually all Linux systems come with their own built-in SSH clients, and as such, there isn't any need to worry about them; however, the non-UNIX operating systems are a different story.

Here is a quick rundown of some SSH clients and other useful SSH resources:

- **PuTTY (www.chiark.greenend.org.uk/~sgtatham/putty)** This is probably one of the oldest and most popular SSH implementations for the Win32 (Microsoft Windows) platforms. It is extremely lightweight and can either be used as a stand-alone, self-contained executable or be installed like other Windows programs. The web site also hosts other tools such as **`pscp`**, which is a Windows command-line version of Secure Copy (SCP).

- **OpenSSH for Apple systems** macOS is actually a UNIX-based and UNIX-compliant operating system. One of its main core components—the kernel—is based on the BSD kernel. So you shouldn't be too surprised that OpenSSH is available on macOS systems. When you open the **terminal** application, you can simply issue the **`ssh`** command. macOS systems also ship with an OpenSSH SSH server.
- **MindTerm, multiplatform (www.cryptzone.com)** This program supports versions 1 and 2 of the SSH protocol. Written in 100 percent Java, it works on many UNIX platforms (including Linux), as well as Windows and macOS. See the web page for a complete list of tested operating systems.
- **Cygwin (www.cygwin.com)** This might be a bit of overkill, but it is well worth the initial effort involved with getting it set up. It is a collection of tools that provides a POSIX-compatible environment for Windows. It can be used to run numerous GNU/Linux programs without extensive changes to their source code. Under Cygwin, you can run all your favorite GNU/Linux programs, such as **`bash`**, **`grep`**, **`find`**, **`nmap`**, **`gcc`**, **`awk`**, **`vim`**, **`emacs`**, **`rsync`**, OpenSSH client (**`ssh`**), OpenSSH server (**`sshd`**), and so on, as though you were at a traditional GNU/Linux shell.
- **FileZilla (https://filezilla-project.org/)** The FileZilla client is a cross-platform FTP, FTPS, and SFTP client.
- **PowerShell (https://github.com/PowerShell/openssh-portable)** A native port of OpenSSH to Microsoft Windows platforms via the PowerShell environment.

The Weakest Link

You've probably heard the saying, "Security is only as strong as your weakest link." This saying is particularly relevant when it comes to OpenSSH and securing your network: OpenSSH is only as secure as the weakest connection between the user and the server. This means that, for example, if a user uses Telnet to connect from host A to host B and then uses **`ssh`** to connect to host C, the entire connection can be monitored from the link between host A and host B. The fact that the link between host B and host C is encrypted becomes irrelevant. Be sure to explain these subtle points to your users.

NOTE As you make connections and use services across the Internet, you are crossing several network boundaries. Each of those providers has full rights and capabilities to sniff traffic and gather any information they want. For example, someone can easily see your e-mail as you read it. With SSH and other things being equal, you can rest assured that your connection is secure.

Installing OpenSSH on RPM-Based Systems

The easiest and quickest way to get an SSH server up and running on any RPM-based Linux system (like Fedora, CentOS, or RHEL) is to use the default package manager available on the system. It is almost guaranteed that you will already have the SSH package installed and running on most modern Linux distributions.

But, again, just in case you are running a Linux distribution that was developed on the planet Neptune but which at least has Red Hat Package Manager (RPM) installed, you can always download and install the precompiled RPM package for OpenSSH.

On our sample Fedora system, we'll type the following to query the RPM database to make sure that OpenSSH is indeed installed:

```
[master@fedora-server ~]$ rpm  -qa | grep -i openssh
openssh-clients-*
openssh-*
openssh-server-*
```

And, if by some freak occurrence, you don't have it already installed (or you accidentally uninstalled it), you can install an OpenSSH server using **dnf** (or Yum) by issuing this command:

```
[master@fedora-server ~]$ sudo dnf  -y install openssh-server
```

Installing OpenSSH via APT in Ubuntu

The Ubuntu Linux distribution usually comes with the client component of OpenSSH preinstalled, but you may sometimes have to (re)install the server component explicitly. Installing the OpenSSH server using Advanced Packaging Tool (APT) in Ubuntu is as simple as running this:

```
master@ubuntu-server:~$ sudo apt-get -y install openssh-server
```

The install process will also automatically start the SSH daemon for you after installation. You can confirm that the software is installed by running the following:

```
master@ubuntu-server:~$ dpkg -l openssh-server
```

Downloading, Compiling, and Installing OpenSSH from Source

As mentioned, virtually all Linux distros ship with OpenSSH; however, you may have a need to roll your own version from source for whatever reason. This section will cover downloading the OpenSSH software and the two components it needs: OpenSSL and zlib. Once these are in place, you can then compile and install the software. If you want to stick with the precompiled version of OpenSSH that ships with your distribution, you can skip this section and move straight to the section "Server Startup and Shutdown."

We will use OpenSSH version 8.4p1 in this section, but you can still follow the steps using any current version of OpenSSH available to you (just change the version number). You can download it from www.openssh.com/portable.html. Select a download site that is closest to you, and download **openssh-8.4p1.tar.gz** (or similar) to a directory with enough free space. (Both **/usr/local/src** and **$HOME/src** are good choices. We use **$HOME/src** in this example.)

Change to the download directory and unpack the OpenSSH source like so:

```
[master@server src]# tar xvzf openssh-8.4p1.tar.gz -C $HOME/src
```

(*continued*)

This will create a directory called **openssh-8.4p1** under **$HOME/src**.

Along with OpenSSH, you will need the proper OpenSSL version. We'll use OpenSSL version 1.1.1* (**openssl-1.1.1*.tar.gz**) in this example. Download the latest version from www.openssl.org. After downloading OpenSSL, unpack it with the **tar** command, like so:

```
[master@server src]$ tar xvzf openssl-1.1.1*.tar.gz
```

Finally, the last package you need is the zlib library, which is used to provide compression and decompression facilities. Most modern Linux distributions have this already, but if you want the latest version, you need to download it from www.zlib.net. We use zlib version 1.2.11 in our example. To unpack the package in **$HOME/src** after downloading, type this:

```
[master@server src]$ tar xvzf zlib-1.2.*.tar.gz
```

The following steps will walk through the process of compiling and installing the various components of OpenSSH and its dependencies:

1. Begin by going into the directory where zlib was unpacked, like so:

   ```
   [master@server src]$ cd $HOME/src/zlib-1.2.11/
   ```

2. Then run **configure** and **make**:

   ```
   [master@server zlib-*]$ ./configure --prefix=/usr/local/
   [master@server zlib-*]$ make
   ```

   This will result in the zlib library being built.

3. Install the zlib library:

   ```
   [master@server zlib-*]$ sudo make install
   ```

   The resulting library will be placed in the **/usr/local/lib** directory.

4. Now you need to compile OpenSSL. Begin by changing to the directory where the downloaded OpenSSL was unpacked:

   ```
   [master@server ~]$ cd $HOME/src/openssl-1.1.1?
   ```

5. Once you're in the OpenSSL directory, all you need to do is run **config** and **make**. OpenSSL will take care of figuring out the type of system it is on and configure itself to work in an optimal fashion. Here are the exact commands:

   ```
   [master@server openssl-1*]$ ./config -fPIC
   [master@server openssl-1*]$ make
   ```

   Note that this step may take a few minutes to complete.

6. If all went well, the compilation step should run without problems by spewing a bunch of stuff on the terminal. If there are any problems, the OpenSSL build process will report them to you. If you do get an error, you should remove this copy of OpenSSL and try the download/unpack/compile procedure again.

(*continued*)

7. Once you have finished compiling, you can install OpenSSL via the following command:

```
[master@server openssl-1*]$ sudo make install
```

   This step will install OpenSSL into the **/usr/local/ssl** directory.

8. You are now ready to begin the actual compile and install of the OpenSSH package. Change into the OpenSSH package directory, like so:

```
[master@server ~]$ cd $HOME/src/openssh-8.4??
```

9. As with the other two packages, you need to begin by running the **`configure`** program. For this package, however, you need to specify some additional parameters. Namely, you need to tell it where the other two packages got installed. You can always run **`./configure`** with the **`--help`** option to see all of the parameters, but you'll find that the following **`./configure`** statement will probably work fine:

```
[master@server openssh-8*]$ ./configure --prefix=/usr/local/ssh  \
--with-ssl-dir=/usr/local/ssl  --with-zlib=/usr/local/lib
```

10. Once OpenSSH is configured, simply run **`make`** and **`make install`** to put all of the files into the appropriate **/usr/local** directories:

```
[master@server openssh-8*]# make
[master@server openssh-8*]# sudo make install
```

That's it—you are done. This set of commands will install the various OpenSSH binaries and libraries under the **/usr/local/ssh/** directory. The SSH server (**sshd**), for example, will be placed under the **/usr/local/ssh/sbin** directory, and the various client components will be placed under the **/usr/local/ssh/bin/** directory.

Note that even though we just walked through how to compile and install OpenSSH from source, the rest of this chapter will assume that we are dealing with OpenSSH as it is installed via RPM or APT (as discussed in previous sections).

Server Startup and Shutdown

If you want users to be able to log into your system via SSH, you will need to make sure that the service is running and also make sure that the service gets started automatically between system reboots.

On modern **systemd**-enabled RPM-based Linux distros, use the **`systemctl`** utility to manage the **sshd** service unit. First check the status of the **sshd** daemon by running the following:

```
[master@server ~]$ systemctl  status  sshd
```

The sample output shows the service is up and running. On the other hand, if the service is stopped, issue this command to start it:

```
[master@server ~]$ sudo systemctl start sshd
```

If you are connected to the SSH server remotely, you should be *very* careful before stopping the service, because you run the risk of kicking yourself off the server once SSHD is stopped. But, if for some reason, you *do* need to stop the SSH server, type the following:

```
[master@server ~]$ sudo systemctl  stop  sshd
```

If you make configuration changes that you want to go into effect, you can restart the daemon at any time by simply running this:

```
[master@server ~]$ sudo systemctl reload sshd
```

On a **systemd**-enabled Debian-based Linux distro such as Ubuntu, you can also use **`systemctl`** to manage the OpenSSH daemon. Note, however, that the daemon is referred to as "**ssh"** in this world and not "**sshd**" (as in the RPM world).

For example, to view the status of the OpenSSH daemon on an Ubuntu distro, type this:

```
master@ubuntu-server:~$ systemctl  status  ssh
```

To start the OpenSSH server, you would run the following:

```
master@ubuntu-server:~$ sudo systemctl  start  ssh
```

To reload the daemon after making any configuration changes, type this:

```
master@ubuntu-server:~$  sudo systemctl  reload  ssh
```

TIP On an openSUSE distro, the command to check the status of **sshd** is

```
opensuse-server:~ # rcsshd status
```

And to start it, the command is

```
opensuse-server:~ # rcsshd start
```

SSHD Configuration File

Out of the box, most Linux systems already have the OpenSSH server configured and running with sane default settings.

On most Linux distributions, the main configuration file for **sshd** usually resides under the **/etc/ssh/** directory and is called **sshd_config**. For the OpenSSH version that we installed from source earlier, the configuration file is located under the **/usr/local/ssh/etc/** directory.

Next we'll discuss some of the configuration options found in the **sshd_config** file:

- **`AuthorizedKeysFile`** Specifies the path to the file that contains the public keys that can be used for user authentication. The default is **`/<User_Home_Directory>/.ssh/authorized_keys`**.

- **`Ciphers`** This is a comma-separated list of ciphers allowed for the SSH protocol version 2. Examples of supported ciphers are 3des-cbc, aes256-cbc, aes256-ctr, arcfour, and blowfish-cbc.
- **`HostKey`** Defines the file containing a private host key used by SSH. The default is either **/etc/ssh/ssh_host_rsa_key**, **/etc/ssh/ssh_host_dsa_key**, **/etc/ssh/ssh_host_ecdsa_key**, or **/etc/ssh/ssh_host_ed25519** for protocol version 2.
- **`Port`** Specifies the port number on which **sshd** listens. The default value is **`22`**.
- **`AllowTcpForwarding`** Specifies whether Transmission Control Protocol (TCP) forwarding is permitted. The default is **`yes`**.
- **`X11Forwarding`** Specifies whether X11 (or Xorg) forwarding is permitted. The argument must be **`yes`** or **`no`**. The default is **`no`**.
- **`ListenAddress`** Specifies the local address on which the SSH daemon listens. By default, OpenSSH will listen on both Internet Protocol version 4 (IPv4) and Internet Protocol version 6 (IPv6) sockets. But if you need to specify a particular interface address, you can tweak this directive.

NOTE **sshd_config** is a rather odd configuration file. Unlike other Linux configuration files, out-of-the-box comments (**#**) in the **sshd_config** file denote the default values of the options that are enabled. In other words, the commented-out parameters represent defaults that are already compiled in.

Using OpenSSH

OpenSSH comes with several useful programs that are covered in this section: the **`ssh`** client program, the Secure Copy (**`scp`**) program, and the Secure FTP (**`sftp`**) program. The most common application you will probably use is the **`ssh`** client program.

Secure Shell (ssh) Client Program

The **`ssh`** client program can be used to securely log into a machine running an **`sshd`** server daemon from any remote location.

By default, the **`ssh`** client program assumes that you want to log into the remote system (destination) as the same user with which you are logged into the local system (source). However, if you need to use a different login (for instance, if you are logged in as *root* on one host and want to **`ssh`** to another and log in as the user *yyang*), all you need to do is provide the **`-l`** option along with the desired login. For example, if you want to log into the host server-B as the user yyang from server-A, you would type this:

```
[master@server-A ~]$ ssh -l yyang server-B
```

Or you could use the ***`username@host`*** command format, like so:

```
[master@server-A ~]$ ssh yyang@server-B
```

You would be prompted with a password prompt from server-B for the user yyang.

But if you just want to log into the remote host without needing to change your login at the remote end, simply run **ssh**, like so:

```
[master@server-A ~]$ ssh server-B
```

With this command and the proper credentials, you'll be logged in as the master user at server-B.

Of course, you can always replace the hostname with a valid IP address, like this:

```
[master@server-A ~]$ ssh yyang@192.168.1.50
```

To connect to a remote SSH server that is also listening on an IPv6 address (for example, 2001:DB8::2), you could try the following:

```
[master@server-A ~]$ ssh -6  yyang@2001:DB8::2
```

TIP If you don't have a remote server to test your **ssh**/**scp**/**sftp** connections, you can easily switch all references to **server-A** or **server-B** to **localhost**. Similarly, you can also switch all references to remote IP addresses from 192.168.1.50 to the loopback IP address of 127.0.0.1. Note that both **localhost** and 127.0.0.1 refer to your local system.

Creating a Secure Tunnel

This section covers what is commonly called the "poor man's virtual private network" (VPN). Essentially, you can use SSH to create a tunnel from your local system to a remote system. This is a handy feature when you need to access an intranet or another system that is not exposed to the outside world on your intranet. For example, you can **ssh** to a file server that will set up the port forwarding to the remote web server.

Let's imagine a scenario like the one described next with the following components:

- **Inside** The inside component consists of the entire local area network, or LAN (the 192.168.1.0 network). It houses various servers and workstations that are accessible *only* by other hosts on the inside. Let's assume that one of the internal servers on the LAN hosts a web-based accounting application. The internal web server's hostname is "accounts," with an IP address of 192.168.1.100.
- **Middle** In the middle, we have our main component—a system with two network interfaces. The system's hostname is serverA. One of the interfaces is connected directly to the Internet. The other interface is connected to the company LAN.

  On serverA, assume the first interface (the wide area network, or WAN, interface) has a public/routable-type IP address of 1.1.1.1 and the second interface has a private-type IP address of 192.168.1.1. The second interface of serverA is connected to the LAN (the 192.168.1.0 network), which is completely cut off from the Internet.

  The only service that is allowed and running on the WAN interface of serverA is the **sshd** daemon. ServerA is said to be "dual-homed" because it is connected to two different networks: the LAN and the WAN.

- **Outside** Our remote user, yyang, needs to access the web-based accounting application running on the internal server (accounts) from home. User yyang's home workstation hostname is hostA. Yyang's home system is considered to be connecting via a hostile public Internet. HostA has an SSH client program installed.

We already said the entire internal company network (LAN, accounts server, other internal hosts, and so on) is cut off from the Internet and the home system (hostA) is part of the public Internet, so what gives? The setup is illustrated in Figure 23-5.

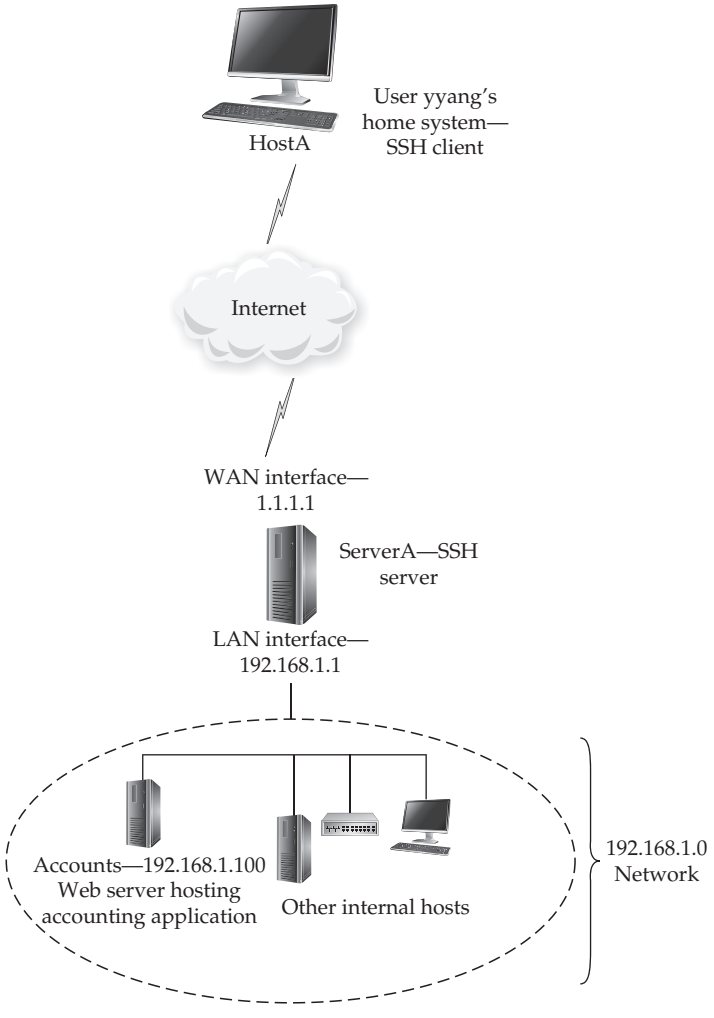


Figure 23-5 Port forwarding with SSH

Enter the poor man's VPN (aka SSH tunneling). The user yyang will set up an SSH tunnel to the web server running on "accounts" by following these steps:

1. While sitting in front of her home system (hostA), the user **yyang** will log into the home system as herself.
2. Once logged in locally, she will create a tunnel from port 9000 on the local system to port 80 on the system (named accounts) running the web-based accounting software.
3. To do this, yyang will connect via SSH to serverA's WAN interface (1.1.1.1) by issuing this command from her system at home (hostA):

```
[yyang@hostA ~]# ssh  -L  9000:192.168.1.100:80   1.1.1.1
```

NOTE The complete syntax and meaning of the port-forwarding command is **`ssh -L local_port:destination_host:destination_port ssh_server`**, where ***`local_port`*** is the local port you will connect to after the tunnel is set up, ***`destination_host:destination_port`*** is the host:port pair where the tunnel will be directed, and ***`ssh_server`*** is the host that will perform the forwarding to the end host.

4. After yyang successfully authenticates herself to serverA and has logged into her account on serverA, she can then launch a web browser installed on her workstation (hostA).
5. User yyang can use a web browser to access the forwarded port (9000) on the local system. For this example, she needs to type the Uniform Resource Locator (URL) **http://localhost:9000** into the address field of the browser.
6. If all goes well, the web content being hosted on the accounting server should show up on yyang's web browser—just as if she were accessing the site from within the local office LAN (that is, the 192.168.1.0 network).
7. To close down the tunnel, she simply closes all windows that are accessing the tunnel and then ends the SSH connection to serverA by typing **exit** at the prompt.

The secure tunnel affords you secure access to other systems or resources within an intranet or a remote location. It is a great and inexpensive way to create a virtual private network between your host and another host. It is not a full-featured VPN solution, since you can't easily access every host on the remote network, but it gets the job done.

In this demo, we port-forwarded HTTP traffic. You can tunnel almost any protocol, such as Virtual Network Computing (VNC) or Remote Desktop Protocol (RDP). Note that this is a way for people inside a firewall or proxy to bypass the firewall mechanisms and get to computers on the outside world. The ProxyJump and SOCKS Proxy features of OpenSSH are alternative methods of transparently traversing firewalls or other barriers to connect to other hosts/resources.

OpenSSH Shell Tricks

It is also possible to create a secure tunnel after you have already logged into the remote SSH server. That is, you don't have to set up the tunnel when you are setting up the initial SSH connection. This is especially useful if you have a shell on a remote host and you need to hop around onto other systems that would otherwise be inaccessible.

SSH has its own nifty little shell that can be used to accomplish this and other neat tricks.

To gain access to the built-in SSH shell after you log into an SSH server, press these three keys: **SHIFT ~ C** (that's a tilde in the middle). That opens a prompt similar to this one:

```
ssh>
```

To set up a tunnel similar to the one we set up earlier, type this command at the **ssh** prompt/shell:

```
ssh> -L 9000:192.168.1.100:80
```

To leave or quit the SSH shell, press ENTER on your keyboard, and you'll be back to your normal login shell on the system.

While logged in remotely to a system via SSH, simultaneously typing the tilde character (~) and the question mark (?) will display a listing of all the other things you can do at the **ssh** prompt. Note that escapes are recognized only immediately after newlines.

```
[master@server ~]$ ~?
```

Here are some of the supported escape sequences:

~.	Terminate the connection.
~C	Open a command line.
~#	List forwarded connections.
~&	Background SSH (when waiting for connections to terminate).
~?	List the supported escape sequences and their meanings.
~~	Send the escape character by typing it twice.

Secure Copy (scp) Program

Secure Copy (**scp**) is used for securely copying data from one host to another remote host. The format and usage of **scp** is very simple—you only need to know the *source* and the *destination*.

Suppose user yyang, for example, is logged into her home workstation (client-A) and wants to copy a file named **.bashrc** located in the local home directory to her home directory on server-A. Here's the command:

```
[yyang@client-A ~]$ scp .bashrc      server-A:/home/yyang
```

If she wants to copy the other way—that is, from the remote system server-A to her local system client-A—the arguments need to be reversed, like so:

```
[yyang@client-A ~]$ scp   server-A:/home/yyang/.bashrc   .bashrc
```

Secure FTP (sftp) Program

Secure FTP is a subsystem of the **sshd** daemon. You access the Secure FTP server by using the **`sftp`** command-line tool. To **`sftp`** from a system named client-A to an SFTP server running on server-A as the user yyang, type this:

```
[master@client-A ~]$ sftp yyang@server-A
```

You will then be prompted for your password (similar to a regular **ssh** session). Once you have been authenticated, you will see a prompt like the following:

```
sftp>
```

You can issue various **`sftp`** commands while at the **sftp** shell. For example, to list all the files and directories under the **/tmp** folder on the **sftp** server, you can use the **`ls`** command:

```
sftp> ls -l /tmp
```

For a listing of all the commands, just type a question mark (**?**):

```
sftp> ?
Available commands:
cd path                 Change remote directory to 'path'
chmod mode path         Change permissions of file 'path' to 'mode'
...<OUTPUT TRUNCATED>...
```

Notice that some of the commands look strikingly similar to the FTP commands discussed in Chapter 18. Among other things, **`sftp`** is handy if you forget the full name of a file you are looking for, because you can browse the remote file system using familiar FTP commands.

Files Used by the OpenSSH Client

The configuration files for the SSH client and SSH server typically reside in the directory **/etc/ssh/** on most Linux distributions. (If you installed SSH from source into **/usr/local/ssh/**, the full path will be **/usr/local/ssh/etc/**.) If you want to make any system-wide changes to defaults for the SSH client, you need to modify the **/etc/ssh/ssh_config** file or its equivalent.

CAUTION Remember that the **sshd_config** file is for the server daemon, while the **ssh_config** file is for the SSH client! Note the letter **d** for daemon in the server configuration filename.

Within a user's home directory, SSH-related data is stored in the **~username/.ssh/** directory. The file **known_hosts** stores host key information and is used to guard against man-in-the-middle attacks. SSH will alert you when remote host keys change. If the keys have changed for a valid reason—for instance, if the server was reinstalled—you will need to edit the **known_hosts** file and delete the line referencing the (now incorrect) identity of the changed server.

Summary

The Secure Shell is the de-facto protocol for enabling secure remote logins for performing system administration tasks and everyday use on Linux and UNIX-like systems. Even Microsoft Windows systems have fully jumped on board and now have their own native SSH implementations! When properly implemented and used, SSH can help to provide confidentiality and integrity of data when used for communications or data transfers on untrusted networks like the Internet.

In closing, remember that using OpenSSH alone doesn't make your system magically and automatically secure. There is no replacement for a set of good security practices. Following the lessons from Chapter 16, you should disable all unnecessary services on any system exposed to untrusted networks.

PART

V

Intranet Services

CHAPTER 24
Network File System (NFS)

CHAPTER 25
Samba

CHAPTER 26
Distributed File Systems (DFS)

CHAPTER 27
Lightweight Directory Access Protocol (LDAP)

CHAPTER 28
Printing

CHAPTER 29
Dynamic Host Configuration Protocol (DHCP)

CHAPTER 30
Virtualization

CHAPTER 31
Backups

This page intentionally left blank

CHAPTER

24 Network File System (NFS)

Network File System (NFS) is one of the native ways of sharing files and applications across the network in the Linux/UNIX world. NFS is somewhat similar to Microsoft Windows File Sharing, in that it allows you to attach to a remote file system (or disk) and work with it as if it were a local drive—a handy tool for sharing files and large storage space among users.

NFS and Windows File Sharing are solutions to the same problem; however, the solutions are very different beasts. NFS requires different configurations, management strategies, tools, and underlying protocols. We will explore NFS and show how to deploy it in this chapter.

The Mechanics of NFS

As with most network-based services, NFS follows the usual client and server paradigms—that is, it has its client-side components and its server-side components.

Chapter 8 covered the concept of mounting and unmounting file systems. The same concepts apply to NFS, except you also need to specify the server hosting the share in addition to the other items (mount options) you would normally define. Of course, you also need to make sure the server is actually configured to permit access to the share!

Let's look at an example. Assume there exists an NFS server named *serverA* that wants to share its local **/home** file system over the network. In NFS parlance, we say that the NFS server is "exporting its **/home** file system." Assume there also exists a client system on the network named *clientA* that needs access to the contents of the **/home** file system being exported by the NFS server. Finally, assume all other requirements are met (permissions, security, compatibility, and so on).

For clientA to access the **/home** share being exported by serverA, clientA needs to make an NFS mount request for **/home** so that it can mount it locally, such that the remote share appears locally as the **/home** directory. Here is a simple command to trigger this:

```
[master@clientA ~]$ sudo mount serverA:/home /home
```

After executing the previous command on clientA, all the users on clientA would be able to view the contents of **/home** as if it were just another directory or local file system. Linux would take care of making all of the network requests to the server.

Remote procedure calls (RPCs) are responsible for handling the requests between the client and the server. RPC technology provides a standard mechanism for any RPC client to contact the server and find out to which service the calls should be directed. Thus, whenever a service wants to make itself available on a server, it needs to register itself with the RPC service manager, **portmap**, which tells the client where the actual service is located on the server.

Versions of NFS

The protocol behind NFS has evolved and changed a lot over the years. Standards committees have helped NFS evolve to take advantage of new technologies, as well as changes in usage patterns. At the time of this writing, three well-known versions of the protocol exist: NFS version 2 (NFSv2), NFS version 3 (NFSv3), and NFS version 4 (NFSv4).

NFSv2 is the oldest of the three. NFSv3 is the standard with perhaps the widest use. NFSv4 has been in development for a while and is the newest standard. NFSv2 should probably be avoided if possible and should be considered only for legacy reasons. NFSv3 should be considered if stability and widest range of client support are desired. NFSv4 should be considered if its bleeding-edge features are needed and probably for very new deployments where backward compatibility is not an issue. Perhaps the most important factor in deciding which version of NFS to consider would be the version that your NFS clients will support.

Here are some of the features of each NFS version:

- **NFSv2** Mount requests are granted on a per-host basis and not on a per-user basis. This version uses Transmission Control Protocol (TCP) or User Datagram Protocol (UDP) as its transport protocol. Version 2 clients have a file size limitation of less than 2GB that they can access.
- **NFSv3** This version includes a lot of fixes for the bugs in NFSv2. It has more features than version 2, has performance gains over version 2, and can use either TCP or UDP as its transport protocol. Depending on the local file system limits of the NFS server itself, clients can access files larger than 2GB in size. Mount requests are also granted on a per-host basis and not on a per-user basis.
- **NFSv4** This version of the protocol uses a stateful protocol such as TCP or Stream Control Transmission Protocol (SCTP) as its transport. It has improved security features thanks to its support for Kerberos; for example, client authentication can be conducted on a per-user basis or on a principal basis. It was designed with the Internet in mind, and as a result, this version of the protocol is firewall-friendly and listens on the well-known port 2049. The services of the RPC binding protocols (such as **rpc.mountd**, **rpc.lockd**, and **rpc.statd**) are no longer required in this version of NFS because their functionality has been built into the server; in other words, NFSv4 combines these previously disparate NFS protocols into a single protocol specification. (The **portmap** service is no longer necessary.) It includes support for file access

control list (ACL) attributes and can support both version 2 and version 3 clients. NFSv4 introduces the concept of the pseudo-file system, which allows NFSv4 clients to see and access the file systems exported on the NFSv4 server as a single file system. NFSv4 is currently at minor revision 2 (NFSv4.2).

The version of NFS used can be specified at mount time by the client via the use of mount options. For a Linux client to use a specific NFS version, the **`nfsvers`** mount option has to be specified with the desired version (for example, **`nfsvers=3`**). Otherwise, the client will negotiate a suitable version with the server.

The rest of this chapter concentrates mostly on NFSv3 and NFSv4, because they are considered quite stable, they are well known, and have the widest cross-platform support.

Security Considerations for NFS

In its default state, NFS is not a secure method for sharing disks. The same commonsense rules that you would apply to securing other network services also apply to securing NFS. You should be able to trust the users on the client systems accessing your server, but if you can't guarantee that this trust exists, you should have measures in place to mitigate the obvious security issues. So, for example, if you're the *root* user on both the client and the server, there is a little less to worry about. The important thing in this case is to make sure non-root users don't become root—which is something you should be doing anyway! You should also strongly consider using NFS mount flags, such as the **`root_squash`** flag discussed later on.

If you cannot fully trust the person with whom you need to share a resource, it will be worth your time and effort to seek alternative methods of sharing resources (such as read-only sharing of the resources).

As always, stay up to date on the latest security bulletins from the Computer Emergency Response Team (www.cert.org), and keep up with all the patches from your distribution vendor.

Mount and Access a Partition

Several steps are involved in a client's making a request to mount a server's exported file system or resource (these steps pertain mostly to NFSv2 and NFSv3):

1. The client contacts the server's portmapper to find out which network port is assigned as the NFS mount service.
2. The client contacts the mount service and requests to mount a file system. The mount service checks to see if the client has permission to mount the requested partition. (Permission for a client to mount a resource is based on directives or options in the **/etc/exports** file.) If all is well, the mount service returns an affirmative.
3. The client contacts the portmapper again—this time to determine on which port the NFS server is located. (Typically, this is port 2049.)
4. Whenever the client wants to make a request to the NFS server (for example, to read a directory), an RPC is sent to the NFS server.
5. When the client is done, it updates its own mount tables but doesn't inform the server.

Notification to the server is unnecessary, because the server doesn't keep track of all clients that have mounted its file systems. Because the server doesn't maintain state information about clients and the clients don't maintain state information about the server, clients and servers can't tell the difference between a crashed system and a really slow system. Thus, if an NFS server is rebooted, ideally all clients should automatically resume their operations with the server as soon as the server is back online.

Enabling NFS in Fedora, RHEL, and CentOS

Almost all the major Linux distributions ship with support for NFS in one form or another. The only task left for the administrator is to configure it and enable it. On our sample Fedora system, enabling NFS is easy.

Because NFS version 3 (and lower) and its ancillary programs are RPC based, you first need to make sure that the system **rpcbind** service is installed and running.

To make sure that the **rpcbind** package is installed on the system, on a RPM-based distro (such as Fedora), type the following:

```
[master@fedora-server ~]$ rpm  -q rpcbind
```

If the output is empty, you can use **dnf** to install it by running this:

```
[master@fedora-server ~]$ sudo dnf -y install rpcbind
```

To check the status of the **rpcbind** service on **systemd**-enabled Linux distros, type this:

```
[master@fedora-server ~]$ systemctl status  rpcbind.service
```

If the **rpcbind** service is stopped, start it like so:

```
[master@fedora-server ~]$ sudo systemctl start rpcbind.service
```

Before going any further, use the **rpcinfo** command to view the status of any RPC-based services that might have registered with **portmap**:

```
[master@fedora-server ~]$ rpcinfo  -p
   program vers proto   port  service
    100000    4   tcp    111  portmapper
    100000    4   udp    111  portmapper
....<OUTPUT TRUNCATED>....
```

Because we don't yet have an NFS server running on the sample system, the output may not show too many RPC services.

To start the NFS service, you can use the **systemctl** command:

```
[master@fedora-server ~]$ sudo systemctl start nfs-server
```

NOTE **systemd** will automatically start **rpcbind** (as a dependency) whenever the nfs server is started, and so you don't need to explicitly start **rpcbind** separately.

Running the **rpcinfo** command again to view the status of RPC programs registered with the portmapper shows this output:

```
[master@fedora-server ~]$ rpcinfo  -p
....<OUTPUT TRUNCATED>....
100003    4   tcp   2049  nfs
100005    1   udp  32892  mountd
```

This output shows various RPC programs (**mountd**, **nfs**, **nlockmgr**, and so on) are now running.

To stop the NFS service, enter this command:

```
[master@fedora-server ~]$ sudo systemctl stop nfs-server
```

To have the NFS service automatically start up with the system with the next reboot, use the **systemctl** command:

```
[master@fedora-server ~]$ sudo systemctl enable nfs-server
```

Enabling NFS in Ubuntu and Debian

Installing and enabling an NFS server in Ubuntu or Debian is as easy as installing the following components: **nfs-common**, **nfs-kernel-server**, and **rpcbind**.

To install these using Advanced Packaging Tool (APT), run the following command:

```
master@ubuntu-server:~$ sudo apt-get -y  install nfs-common  \
nfs-kernel-server  rpcbind
```

The install process will also automatically start up the NFS server, as well as all its attendant services, for you. You can check this by running the following:

```
master@ubuntu-server:~$ sudo rpcinfo  -p
```

To stop the NFS server in Ubuntu, type this:

```
master@ubuntu-server:~$ systemctl stop  nfs-server
```

The Components of NFS

Versions 2 and 3 of the NFS protocol rely heavily on RPCs to handle communications between clients and servers. RPC services in Linux are managed by the **portmap** service. As mentioned, this ancillary service is no longer needed in NFSv4 and higher.

The following list shows the various RPC processes that facilitate the NFS service under Linux. The RPC processes are mostly relevant only in NFS versions 2 and 3, but mention is made wherever NFSv4 applies.

- **rpc.statd** This process is responsible for sending notifications to NFS clients whenever the NFS server is restarted without being gracefully shut down. It provides status information about the server to **rpc.lockd** when queried. This is done via the Network Status Monitor (NSM) RPC protocol. It is an optional service that is started automatically by the **nfslock** service. It is not required in NFSv4.
- **rpc.rquotad** As its name suggests, **rpc.rquotad** supplies the interface between NFS and the quota manager. NFS users/clients will be held to the same quota restrictions that would apply to them if they were working on the local file system instead of via NFS. It is not required in NFSv4.
- **rpc.mountd** When a request to mount a partition is made, the **rpc.mountd** daemon takes care of verifying that the client has the appropriate permission to make the request. This permission is stored in the **/etc/exports** file. (The upcoming section "The /etc/exports Configuration File" tells you more about the **/etc/exports** file.) It is automatically started by the NFS server **init** scripts. It is not required in NFSv4.
- **rpc.nfsd** The main component to the NFS system, this is the NFS server/daemon. It works in conjunction with the Linux kernel either to load or unload the kernel module as necessary. It is, of course, still relevant in NFSv4.
- **rpc.lockd** The **rpc.statd** daemon uses this daemon to handle lock recovery on crashed systems. It also allows NFS clients to lock files on the server. The **nfslock** service is no longer used in NFSv4.
- **rpc.idmapd** This is the NFSv4 ID name-mapping daemon. It provides this functionality to the NFSv4 kernel client and server by translating user and group IDs to names, and vice versa.
- **rpc.svcgssd** This is the server-side **rpcsec_gss** daemon. The **rpcsec_gss** protocol allows the use of the **gss-api** generic security API to provide advanced security in NFSv4.
- **rpc.gssd** This provides the client-side transport mechanism for the authentication mechanism in NFSv4 and higher.

NOTE You should understand that NFS itself is an RPC-based service, regardless of the version of the protocol. Therefore, even NFSv4 is inherently RPC based. The fine point here lies in the fact that most of the previously used ancillary and stand-alone RPC-based services (such as **mountd** and **statd**) are no longer necessary, because their individual functions (or functionality) have now been folded into the NFSv4 daemon.

Kernel Support for NFS

NFS is implemented in two forms among the various Linux distributions. Most distributions ship with NFS support enabled in the kernel. A few Linux distributions also ship with support for NFS in the form of a stand-alone daemon that can be installed via a package.

Although not mandatory, kernel-based NFS server support is considered the de facto standard. However, if you choose to run NFS as a stand-alone daemon, you can rest assured that the **nfsd** program that handles NFS server services is completely self-contained and provides everything necessary to serve NFS.

NOTE On the other hand, clients must have support for NFS in the kernel. This support in the kernel has been around for a long time and is thus stable. Almost all present-day Linux distributions ship with kernel support for NFS enabled.

Configuring an NFS Server

Setting up an NFS server is a two-step process. The first step is to create the **/etc/exports** file, which defines which parts of your server's file system or disk are shared with the rest of your network and the rules by which they get shared (for example, is a client allowed read-only access to the file system, or also allowed to write to the file system?). After defining the **/etc/exports** file, the second step is to start the NFS server processes that read the **/etc/exports** file.

The /etc/exports Configuration File

This primary configuration file for the NFS server lists the file systems that are sharable, the hosts with which they can be shared, and with what permissions (as well as other parameters). The file specifies remote mount points for the NFS mount protocol.

The format for the file is simple. Each line in the file specifies the mount point(s) and export flags within one local server file system for one or more hosts.

Here is the format of each entry/line in the **/etc/exports** file:

```
/directory/to/export client|ip_network(permission) client|ip_network(permission)
```

The different fields are explained here:

- **/directory/to/export** This is the directory you want to share with other users—for example, **/home**.
- **client** This refers to the hostname(s) of the NFS client(s).
- **ip_network** This allows the matching of hosts by IP addresses (for example, 172.16.1.1) or network addresses with a netmask combination (for example, 172.16.0.0/16). Wildcard characters (such as * and ?) are also supported in this field.
- **permission** These are the corresponding permissions for each client. Table 24-1 describes the valid permissions for each client.

Following is an example of a complete NFS **/etc/exports** file. (Note that line numbers have been added to the listing to aid readability.)

```
1. # /etc/exports file for serverA
2. #
3. /home hostA(rw) hostB(rw) clientA(rw,no_root_squash)
4. /usr/local 172.16.0.0/16(ro)
```

Permission Option	Meaning
`secure`	The port number from which the client requests a mount must be lower than 1024. This permission is on by default. To turn it off, specify `insecure` instead.
`ro`	Allows read-only access to the partition. This is the default permission whenever nothing is specified explicitly.
`rw`	Allows normal read/write access.
`noaccess`	The client will be denied access to all directories below **/dir/to/mount**. This allows you to export the directory **/dir** to the client and then to specify **/dir/to** as inaccessible without taking away access to something like **/dir/from**.
`root_squash`	This permission prevents remote root users from having superuser (root) privileges on remote NFS-mounted volumes. Here, `squash` literarily means to squash the power of the remote root user.
`no_root_squash`	This allows the root user on the NFS client host to access the NFS-mounted directory with the same rights and privileges that the superuser would normally have.
`all_squash`	Maps all user IDs (UIDs) and group IDs (GIDs) to the anonymous user. The opposite option is `no_all_squash`, which is the default setting.

Table 24-1 NFS Permissions

Lines 1 and 2 are comments and are ignored when the file is read.

Line 3 exports the **/home** file system to the machines named hostA and hostB, and gives them read/write (`rw`) permissions. And to the machine named clientA, it gives read/write (`rw`) access and allows the remote root user to have root privileges on the exported file system (**/home**)—this last bit is indicated by the `no_root_squash` option.

Line 4 exports the **/usr/local/** directory to all hosts on the 172.16.0.0/16 network. Hosts in the network range are allowed read-only access.

Telling the NFS Server Process About /etc/exports

Once you have an **/etc/exports** file setup, use the `exportfs` command to tell the NFS server processes to reread the configuration. The parameters for `exportfs` are as follows:

exportfs Command Option	Description
`-a`	Exports all entries in the **/etc/exports** file. It can also be used to unexport the exported file systems when used along with the `u` option—for example, `exportfs -ua`.
`-r`	Re-exports all entries in the **/etc/exports** file. This synchronizes **/var/lib/nfs/xtab** with the contents of the **/etc/exports** file. For example, it deletes entries from **/var/lib/nfs/xtab** that are no longer in **/etc/exports** and removes stale entries from the kernel export table.
`-u clientA:`*`/dir/to/mount`*	Unexports the directory ***/dir/to/mount*** to the host `clientA`.

exportfs Command Option	Description
`-o options`	The ***options*** specified here are the same as described in Table 24-1 for client permissions. These ***options*** will apply only to the file system specified on the **exportfs** command line, not to those in **/etc/exports**.
`-v`	Be verbose.

Following are sample usages of **exportfs** from the command line.

To export all file systems specified in the **/etc/exports** file, type this:

```
[master@server ~]$  sudo exportfs -a
```

To export the directory **/usr/local** to the host **clientA** with the **read/write** and **no_root_squash** permissions, type this:

```
[master@server ~]$sudo exportfs -o rw,no_root_squash clientA:/usr/local
```

In most instances, you will simply want to use **exportfs** with the **-r** option (meaning re-export all directories and complete other housekeeping tasks).

The showmount Command

When you're configuring NFS, you'll find it helpful to use the **showmount** command to see if everything is working correctly. The command is used for showing mount information for an NFS server. By using the **showmount** command, you can quickly determine whether you have configured **nfsd** correctly.

After you have configured your **/etc/exports** file and exported all your file systems using **exportfs**, you can run **showmount -e** to see a list of exported file systems on the local NFS server. The **-e** option tells **showmount** to show the NFS server's export list. Here's an example:

```
[master@server ~]$ showmount -e localhost
Export list for localhost:
/home *
```

Running the **showmount** command without any options will list clients connected to the server:

```
[master@server ~]$ showmount localhost
Hosts on localhost:
*
192.168.1.100
```

You can also run this command on clients by passing the server hostname as the last argument. To show the exported file systems on a remote NFSv3 server (serverA) from an NFS client (clientA), you can issue this command while logged into clientA:

```
[yyang@clientA ~]$ showmount -e serverA
Export list for serverA:
/home *
```

Troubleshooting Server-Side NFS Issues

When exporting file systems, you may sometimes find that the server appears to be refusing the client access, even though the client is listed in the **/etc/exports** file. Typically, this happens because the server takes the IP address of the client connecting to it and resolves that address to the fully qualified domain name (FQDN), and the hostname listed in the **/etc/exports** file isn't qualified. For example, this might happen if the server thinks the client hostname is *clientA.example.com*, but the **/etc/exports** file lists just *clientA*.

Another common problem is that the server's perception of the hostname/IP pairing is not correct. This can occur because of an error in the **/etc/hosts** file or in the Domain Name System (DNS) tables. You'll need to verify that the pairing is correct.

For NFSv2 and NFSv3, the NFS service may fail to start correctly if the other required services, such as the **portmap** service, are not already running.

Even when everything seems to be set up correctly on the client side and the server side, you may find that the firewall on the server side is preventing the mount process from completing. In such situations, you will notice that the **mount** command seems to hang without any obvious errors. On Red Hat–like systems that use **firewalld** for firewall rule management, you can permanently open the port for the NFS service on the server by using the **firewall-cmd** command, like this:

```
[master@fedora-server ~]$ sudo firewall-cmd --add-service=nfs --permanent
[master@fedora-server ~]$ sudo firewall-cmd --add-service=rpc-bind  --permanent
[master@fedora-server ~]$ sudo firewall-cmd --add-service=mountd --permanent
[master@fedora-server ~]$ sudo firewall-cmd  --reload
```

Configuring NFS Clients

NFS clients are remarkably easy to configure under Linux, because they don't require any new or additional software to be loaded. The only requirement is that the kernel be compiled to support the NFS file system. Virtually all Linux distributions come with this feature enabled by default in their stock kernel. Aside from the kernel support, the only other important factor is the options used with the **mount** command.

The mount Command

The **mount** command was originally discussed in Chapter 8. The important parameters to use with the **mount** command are the NFS server name or IP address, the local mount point, and the options specified after **-o** on the **mount** command line.

The following is an example of an NFS **mount** command invocation:

```
[master@clientA ~]$ sudo mount -o rw,bg,soft  serverA:/home  /mnt/home
```

Here, **serverA** is the NFS server name. Make sure that the name is resolvable via either DNS or the **/etc/hosts** file. The various **-o** options available are explained in Table 24-2.

These **mount** options can also be used (hardcoded) in the **/etc/fstab** file. This same entry in the **/etc/fstab** file would look like this:

```
serverA:/home   /mnt/home  nfs  rw,bg,soft   0 0
```

Again, **serverA** is the sample NFS server name, and the mount options are **rw**, **bg**, and **soft**, all explained in Tables 24-1 and 24-2.

mount -o Command Option	Description
bg	Background mount. Should the mount initially fail (for instance, if the server is down), the mount process will send itself to background processing and continue trying to execute until it is successful. This is useful for file systems mounted at boot time, because it keeps the system from hanging at the **mount** command if the server is down.
intr	Specifies an interruptible mount. If a process has pending I/O on a mounted partition, this option allows the process to be interrupted and the I/O call to be dropped. For more information, see "The Importance of the intr Option," later in this chapter.
hard	This is an implicit default option. If an NFS file operation has a major timeout, a "server not responding" message is reported on the console and the client continues retrying indefinitely.
soft	Enables a soft mount for this partition, allowing the client to time out the connection after a number of retries (specified with the **retrans=n** option). For more information, see "Soft vs. Hard Mounts," later in this chapter.
retrans=*n*	The value ***n*** specifies the maximum number of connection retries for a *soft* mount.
rsize=*n*	The value ***n*** is the number of bytes NFS uses when reading files from an NFS server. The default value is dependent on the kernel but is currently **4096** bytes for NFSv4. Throughput can be improved greatly by requesting a higher value (for example, **rsize=32768**).
wsize=*n*	The value ***n*** specifies the number of bytes NFS uses when writing files to an NFS server. The default value is dependent on the kernel but is currently **4096** bytes for NFSv4. Throughput can be greatly improved by asking for a higher value (such as **wsize=32768**). This value is negotiated with the server.
proto=*n*	The value ***n*** specifies the network protocol to use to mount the NFS file system. The default value in NFSv2 and NFSv3 is **udp**. NFSv4 servers generally support only TCP. Therefore, some valid protocol types are **udp**, **tcp**, **udp6**, and **tcp6**.
nfsvers=*n*	Allows the use of an alternative RPC version number to contact the NFS daemon on the remote host. The default value depends on the kernel, but the possible values are **2**, **3**, **4**, **4.1**, and so on. If you want to strictly enforce an NFSv4 mount, you can simply and explicitly state **nfs4** as the file system type (**-t nfs4**) during mount time.
sec=*value*	Sets the security mode for the mount operation to ***value***: • **sec=sys** Uses local UNIX UIDs and GIDs to authenticate NFS operations (**AUTH_SYS**). This is the default setting. • **sec=krb5** Uses Kerberos V5 instead of local UIDs and GIDs to authenticate users. • **sec=krb5i** Uses Kerberos V5 for user authentication and performs integrity checking of NFS operations using secure checksums to prevent data tampering. • **sec=krb5p** Uses Kerberos V5 for user authentication and integrity checking and encrypts NFS traffic to prevent traffic sniffing.

Table 24-2 Mount Options for NFS

Soft vs. Hard Mounts

By default, NFS operations are *hard,* which means the clients continue their attempts to contact the server indefinitely. However, this behavior is not always desirable! It causes a problem if an emergency shutdown of all systems is performed. If the servers happen to get shut down before the clients, the clients' shutdowns will stall while they wait for the servers to come back up. Enabling a *soft* mount allows the client to time out the connection after a number of retries (specified with the **`retrans=r`** option).

NOTE There is one exception to the preferred approach of using soft mounts: Don't use this arrangement when you have data that must be committed to disk no matter what and you don't want to return control to the application until the data has been committed. (NFS-mounted mail directories are typically mounted this way.)

Cross-Mounting Disks

Cross-mounting can best be described as the process of having serverA NFS-mounting serverB's disks and serverB NFS-mounting serverA's disks. Although this may appear innocuous at first, there is a subtle danger in doing this. If both servers crash, and if each server requires mounting the other's disk in order to boot correctly, you've got a chicken-and-egg problem. ServerA won't boot until serverB is done booting, but serverB won't boot because serverA isn't done booting!

To avoid this problem, avoid situations that require this interdependencies. Ideally, servers should be able to boot completely without needing to mount anyone else's disks for anything critical. However, this doesn't mean you can't cross-mount at all. There are legitimate reasons for cross-mounting, such as needing to make home directories available across all servers. In these situations, make sure you set your **/etc/fstab** entries to use the **`bg`** mount option. By doing so, you will allow each server to background the **`mount`** process for any failed mounts, thus giving all of the servers a chance to boot completely and then properly make their NFS-mountable file systems available.

The Importance of the intr Option

When a process makes a system call, the kernel takes over the action. During the time that the kernel is handling the system call, the process may not have control over itself. In the event of a kernel access error, the process must continue to wait until the kernel request returns; the process can't give up and quit. In normal cases, the kernel's control isn't a problem, because typically, kernel requests get resolved quickly. When there's an error, however, it can be quite a nuisance. Because of this, NFS has an option to mount file systems with the *interruptible* flag (the **`intr`** option), which allows a process that is waiting on an NFS request to give up and move on. In general, unless you have reason *not* to use the **`intr`** option, it is usually a good idea to do so.

Performance Tuning

The default block size that is transmitted with NFS version 3 is 8192 bytes (for NFSv4, it is 32,768 bytes). There might be situations where you need to tune these default values to take

advantage of faster network stacks or faster equipment available to you. This is where the **wsize** (write size) and **rsize** (read size) options come in handy. You will usually want to tune these values up or down (in cases where you have older hardware) to suit your environment.

Here is a sample entry in an NFS client's **/etc/fstab** file to tweak (or double the value to 65,536 bytes) the **wsize** and **rsize** options for NFSv4:

```
serverA:/home /mnt/home nfs nfsvers=4,rw,bg,wsize=65536,rsize=65536 0  0
```

Troubleshooting Client-Side NFS Issues

Like any major service, NFS has mechanisms to help it cope with error conditions. In this section, we discuss some common error cases and how NFS handles them.

Stale File Handles

If a file or directory is in use by one process when another process removes the file or directory, the first process gets an error message from the server. Typically, this error states something to the following effect: "Stale NFS file handle."

Most often, stale file handle errors can occur, for example, when you're using the graphical environment on a system and you have two GUI terminal windows open. If the first terminal window is in a particular directory—say, **/mnt/usr/local/mydir/**—and that directory gets deleted from the second terminal window, the next time you press ENTER in the first terminal window, you'll see the error message.

To fix this problem, simply change your directory to one that you know exists, without using relative directories (for example, **cd /tmp**).

Permission Denied

You're likely to see the "Permission denied" message if you're logged in as *root* and are trying to access a file that is NFS-mounted. Typically, this means that the server on which the file system is mounted is not acknowledging root's permissions.

This is usually the result of forgetting that the **/etc/exports** file will, by default, enable the **root_squash** option. So if you are experimenting from a permitted NFS client as the root user, you might wonder why you are getting access-denied errors even though the remote NFS share seems to be mounted properly.

The quick way around this problem is to become the user who owns the file you're trying to control. For example, if you're root and you're trying to access a file owned by the user yyang, use the **su** command to become yyang:

```
[root@clientA ~]# su  -  yyang
```

When you're done working with the file, you can exit out of yyang's shell and return to root. Note that this workaround assumes that yyang exists as a user on the system and has the same UID on both the client and the server.

A similar problem occurs when users clearly have the same usernames on the client and the server but still get permission-denied errors. This might happen because the actual UIDs

associated with the usernames on both systems are different. For example, suppose the user *mmellow* has a UID of 1003 on the host clientA, but a user with the same name on serverA has a UID of 6000. The simple workaround to this can be to create/maintain users with the same UIDs and GIDs across all systems. The scalable workaround to this may be to implement a central user database infrastructure, such as LDAP or NIS, so that all users have the same UIDs and GIDs, independent of their local client systems.

TIP Keep those UIDs in sync! Every NFS client request to an NFS server includes the UID of the user making the request. This UID is used by the server to verify that the user has permissions to access the requested file. However, in order for NFS permission-checking to work correctly, the UIDs of the users must be synchronized between the client and server. (The **`all_squash`** option can circumvent this when used in the **/etc/exports** file.) Having the same username on both systems is *not* enough, however. The numerical equivalent of the usernames (UID) should also be the same. A Network Information Service (NIS) database or a Lightweight Directory Access Protocol (LDAP; see Chapter 27) database can help in this situation. These directory systems help to ensure that UIDs, GIDs, and other information are in sync by keeping all the information in a central database.

Sample NFS Client and NFS Server Configuration

In this section, you'll put everything you've learned thus far together by walking through the actual setup of an NFS environment. We will set up and configure the NFS server. Once that is accomplished, we will set up an NFS client and make sure that the directories get mounted when the system boots.

In particular, we want to export the **/usr/local** file system on the host serverA to a particular host on the network named clientA. We want clientA to have read/write access to the shared volume and the rest of the world to have read-only access to the share. Our clientA will mount the NFS share at its **/mnt/usr/local** mount point. The procedure involves these steps:

TIP For a quick-and-dirty way to make sure that both server(s) and client(s) can resolve the other system's hostname to the correct IP address, you can create quick entries in the **/etc/hosts** file using these sample commands on the relevant systems. Here, clientA's IP is 172.16.0.113 and serverA's IP is 172.16.0.2:

```
$ echo "172.16.0.113   clientA" |  sudo tee -a /etc/hosts
$ echo "172.16.0.2   serverA" |  sudo tee -a /etc/hosts
```

1. On the server, serverA, edit the **/etc/exports** configuration file. We want to share **/usr/local**, so input the following into the **/etc/exports** file:

   ```
   /usr/local clientA(rw,root_squash) *(ro)
   ```

2. Save your changes to the file when you are done editing and then exit the text editor.
3. On the NFS server, first check whether the **rpcbind** is running:

   ```
   [master@serverA ~]$ systemctl status rpcbind
   ```

If it is not running, start it. If it is stopped or inactive, you can start it with this command:

```
[master@serverA ~]$ sudo systemctl  start   rpcbind
```

TIP On an openSUSE system, the equivalent of the preceding commands are **`rcrpcbind status`** and **`rcrpcbind start`**.

4. Start the NFS service, which will start all the other attendant services it needs. Use the **`systemctl`** command on **systemd**-enabled Linux distros to start the service:

   ```
   [master@serverA ~]$ sudo systemctl  start  nfs-server
   ```

5. Use the **`exportfs`** command to re-export the directories in **/etc/exports**:

   ```
   [master@serverA  ~]$ sudo exportfs -r
   ```

6. To check whether your exports are configured correctly, run the **`showmount`** command:

   ```
   [master@serverA ~]$ showmount -e localhost
   ```

7. If you don't see the file systems that you put into **/etc/exports**, check **/var/log/messages** for any output that **`nfsd`** or **`mountd`** might have logged. For **journald**-enabled systems, you can also use the **`journalctl -f -xe`** command to monitor the logs in real time.

   If you need to make changes to **/etc/exports**, don't forget to reload or restart the **nfsd** service and run **`exportfs -r`** when you are done making the changes. And, finally, run **`showmount -e`** again to make sure that the changes took effect.

8. Now that you have the server configured, it is time to set up the client. First, see if the **rpc** mechanism is working between the client and the server. You will again use the **`showmount`** command to verify that the client can see the shares. If the client cannot see the shares, you might have a network problem or a permissions problem to the server. From clientA, issue the following command:

   ```
   [master@clientA ~]# showmount -e serverA
   Export list for serverA:
   /usr/local (everyone)
   ```

TIP If the **`showmount`** command returns an error similar to "clnt_create: RPC: Port mapper failure - Unable to receive: errno 113 (No route to host)" or "clnt_create: RPC: Port mapper failure..." then you should ensure that a firewall running on the NFS server or between the NFS server and the client is not blocking the communications. Actual mounting might still work in spite of this error! You need to open up ports for the following services: **nfs**, **rpc-bind**, and **mountd**.

9. Once you have verified that you can view shares from the client, it is time to see if you can successfully mount a file system. First, create the local **/mnt/usr/local/** mount point and then use the **mount** command as follows:

```
[master@clientA ~]$ sudo mkdir -p /mnt/usr/local
[master@clientA ~]$ sudo mount -o rw,bg,intr,soft \
 serverA:/usr/local  /mnt/usr/local
```

10. Use the **mount** command to view only the NFS-type file systems that are mounted on clientA:

```
[master@clientA ~]# mount -t nfs
```

Or for NFSv4 mounts, you would run this:

```
[master@clientA ~]# mount -t nfs4
```

11. If these commands succeed, you can add the **mount** entry with its options into the **/etc/fstab** file so that the remote file system will get mounted upon reboot:

```
serverA:/usr/local  /mnt/usr/local   nfs   rw,bg,intr,soft  0 0
```

Common Uses for NFS

The following ideas are, of course, just ideas. You are likely to have your own reasons for sharing file systems via NFS.

- **To host popular programs** If you are accustomed to Windows, you've probably worked with applications that refuse to be installed on network shares. For one reason or another, these programs want each system to have its own copy of the software—a nuisance, especially if a lot of machines need the software. Linux rarely has such conditions prohibiting the installation of software on network disks. Thus, many sites install heavily used software on a special file system that is exported to all hosts.
- **To hold home directories** Another common use for NFS partitions is to hold home directories. By placing home directories on NFS-mountable partitions, it's possible to configure the Automounter (and a directory service) so that users can log into any machine in the network and have their home directory available to them. Heterogeneous sites typically use this configuration so that users can seamlessly move from one variant of Linux to another without worrying about the location of their personal data.
- **For shared mail spools** A directory residing on the mail server can be used to store all of the user mailboxes, and the directory can then be exported via NFS to all hosts on the network. In this setup, traditional UNIX mail readers can read a user's e-mail straight from the spool file stored on the NFS share. In the case of large sites with heavy e-mail traffic, multiple servers might be used for providing Post Office Protocol version 3 (POP3) mailboxes, and all the mailboxes can easily reside on a common NFS share that is accessible to all the servers.

Summary

In this chapter, we discussed the process of setting up an NFS server and client. This requires little configuration on the server side. The client side requires a wee bit more configuration. But, in general, the process of getting NFS up and running is relatively painless. Here are some key points to remember:

- NFS has been around for a long time now, and as such, it has gone through several revisions of the protocol specifications. The revisions are mostly backward-compatible, and each succeeding revision can support clients using the older versions.
- NFS version 4.* is the newest revision and is loaded with a lot of improvements and features that were not previously available. As of this writing, NFSv4.* might still not be the most widely deployed version of the protocol in the wild. However, it is the stock version implemented and shipped with the mainstream Linux distros. As such, it is quickly becoming the norm/standard in comparison to the aging NFSv3.
- The older NFS protocols (versions 2 and 3) are implemented as a stateless protocol. Clients can't tell the difference between a crashed server and a slow server; thus, recovery is automatic when the server comes back up. In the reverse situation, when the client crashes and the server stays up, recovery is also automatic.
- The key server processes in NFSv2 and NFSv3 are **rpc.statd**, **rpc.rquotad**, **rpc.mountd**, and **rpc.nfsd**. Most of these functions have been integrated into NFSv4.

NFS is a powerful tool for sharing storage volumes/file systems across network clients. Be sure to spend some time experimenting with it before using it to try to meet your environment's resource-sharing needs.

This page intentionally left blank

CHAPTER

25 Samba

Samba is a powerful suite of applications that provides an open source implementation of the Server Message Block/Common Internet File System (SMB/CIFS) protocol. Among other things, Samba helps Linux-based systems interoperate with Windows-based operating systems.

Samba transparently provides file and print sharing services to Windows clients as well as other networked clients running other operating systems. It does this through the use of the native Microsoft networking protocols SMB/CIFS. From a system administrator's point of view, this means you can deploy a Linux-based server and use it to provide file-sharing, authentication, print, and other services to other non-native Linux clients such as Microsoft Windows systems. Using Samba means that Windows systems can use their native tongue to talk to the Linux server—which means fewer hassles for you and seamless integration for your users.

This chapter covers the procedure for downloading, compiling, and installing Samba. As with any software of its caliber, Samba provides a rich set of configuration options that makes it suitable and flexible for use in various environments. And, thankfully, you should be able to get Samba up and running with very few changes to the default configuration. We'll concentrate on how to perform customary tasks with Samba and how to avoid some common Samba pitfalls. We'll also provide brief coverage of some common command-line utilities such as **`smbclient`**.

No matter what task you've chosen for Samba to handle, be sure to take the time to read the program's documentation. It is well written, complete, and thorough.

NOTE Samba has been ported to a significant number of platforms and almost any variant of UNIX you can imagine, and even several non-UNIX environments.

The Mechanics of SMB

For a complete understanding of the Linux/Samba/Windows relationship, you need to understand the relationships of the operating systems to their files, printers, users, and networks. To see how these relationships compare, let's examine some of the fundamental issues of working with both Linux-based systems and Windows in the same environment.

Usernames and Passwords

The Linux login/password mechanism is radically different from the Windows Active Directory model, which uses domain controllers (DCs). For interoperability, it is important for the system administrator to maintain consistency in the logins and passwords across both platforms. Users may need to work in heterogeneous environments and may need access to the different platforms for various reasons. It is thus useful to make working in such environments as seamless as possible without having to worry about users' needing to reauthenticate separately on the different platforms, worry about cached passwords that don't match between servers, and so on.

Relative to Samba, several options are available for handling username and password issues in heterogeneous environments, including the following:

- **Linux pluggable authentication modules (PAMs)** Allow you to authenticate users against a DC. This means you still have two user lists—one local and one on the DC—but your users need to keep track of their passwords only on the Windows system.
- **Samba as a DC** Allows you to keep all your logins and passwords on the Linux system, while all your Windows boxes authenticate with Samba. When Samba is used with a Lightweight Directory Access Protocol (LDAP) back-end for this, you will have a scalable and extensible solution.
- **Custom script** Allows you to use your own custom script. For sites with a well-established system for maintaining logins and passwords, it isn't unreasonable to come up with a custom script. This can be done using a scripting language with good cross-platform support. Such scripts can be coaxed to allow changes to the Security Access Manager (SAM) to update the DC's password list. For example, a Perl or Python script on the Linux side can communicate with a Perl or Python script on the Windows side to keep accounts synchronized. (Both Perl and Python have been successfully ported to work on Windows platforms through various mechanisms.)

In the worst-case situation, you can always maintain the username and password databases of the different platforms by hand (which some early system admins did indeed have to do!), but this method is error-prone and not much fun to manage.

Encrypted Passwords

Windows-based systems use encrypted passwords when communicating with the DC and any server requiring authentication (including Linux and Samba). The encryption algorithm used by Windows is different from Linux's, however, and, therefore, is not compatible.

Here are your choices for handling this conflict:

- Edit the Registry on Windows clients to disable the use of encrypted passwords. The Registry entries that need to be changed are listed in the **docs** directory in the Samba package. As of version 3 of Samba, this option is no longer necessary.
- Configure Samba to use Windows-style encrypted passwords.

The first solution has the benefit of not pushing you to a more complex password scheme. On the other hand, you may have to apply the Registry fix on all your clients. The second option, of course, has the opposite effect: for a little more complexity on the server side, you don't have to modify any of your clients.

Samba Daemons

The Samba code is actually composed of several components and daemons. We will examine three of the main daemons here: **smbd**, **nmbd**, and **winbindd**.

The **smbd** daemon handles the actual sharing of file systems and printer services for clients. It is also responsible for user authentication and resource-locking issues. It starts by binding to port 139 or 445 and then listens for requests. Every time a client authenticates itself, **smbd** makes a copy of itself; the original goes back to listening to its primary port for new requests, and the copy handles the connection for the client. This new copy also changes its effective user ID from *root* to the authenticated user. For example, if the user *yyang* authenticated against **smbd**, the new copy would run with the permissions of yyang, not the permissions of root. The copy stays in memory as long as there is a connection from the client.

The **nmbd** daemon is responsible for handling NetBIOS name service requests. **nmbd** can also be used as a drop-in replacement for a Windows Internet Name Server (WINS). It begins by binding itself to port 137; unlike **smbd**, however, **nmbd** does not create a new instance of itself to handle every query. In addition to name service requests, **nmbd** handles requests from master browsers, domain browsers, and WINS servers—and as such, it participates in the network resource browsing protocols. The services provided by the **smbd** and **nmbd** daemons complement each other.

Finally, the service provided by **winbindd** can be used to query native Windows servers for user and group information, which can then be used on purely Linux/UNIX platforms. It does this by using Microsoft Remote Procedure Call (RPC) calls, PAM, and the name service switch (NSS) capabilities found in modern C libraries. Its use can be extended through the use of a PAM module (**pam_winbind**) to provide authentication services. This service is controlled separately from the main **smb** service and can run independently.

NOTE With the advent of Microsoft's Active Directory, you shouldn't need the services of **nmbd** anymore, but the reality is that you will, especially if you intend to allow very old Windows hosts on your network to access your Samba shares.

Installing Samba via RPM

Precompiled binaries for Samba exist for most Linux distributions. This section will show how to install Samba on Red Hat Package Manager (RPM) based distros like Fedora, RHEL, CentOS, and so on. To provide the server-side services of Samba, three packages are needed:

- **samba*.rpm** This package provides an SMB server that can be used to provide network services to SMB/CIFS clients.
- **samba-common*.rpm** This package provides files necessary for both the server and client packages of Samba—such as configuration files, log files, man pages, PAM modules, and other libraries.
- **samba-client*.rpm** This package provides the SMB client utilities that allow access to SMB shares and printing services on Linux and non-Linux–type systems. The package is used on Fedora, openSUSE, and other RHEL-type systems.

Assuming you have a working connection to the Internet, installing Samba can be as simple as issuing this command:

```
[master@fedora-server ~]$ sudo dnf -y install samba
```

Install the **samba-common-tools** package by running the following:

```
[master@fedora-server ~]$ sudo dnf -y install samba-common-tools
```

You can similarly install the **samba-client** package like so:

```
[master@fedora-server ~]$ sudo dnf -y install samba-client
```

Installing Samba via APT

The essential components of the Samba software on Debian-like distros, such as Ubuntu, are split into **samba*.deb** and **samba-common*.deb** packages. Getting the client and server components of Samba installed in Ubuntu is as easy as running this:

```
master@ubuntu-server:~$ sudo apt-get -y install samba
```

As with installing most other services under Ubuntu, the installer will automatically start the Samba daemons after installation.

Samba Administration

This section describes some typical Samba administrative functions. You'll learn how to start and stop Samba, how to perform common administrative tasks with Samba, how to use **`smbclient`**, and so on.

Starting and Stopping Samba

On our sample **systemd**-enabled Linux server, the **`systemctl`** utility can be used to manage Samba's startup and shutdown.

For example, to start the **smbd** daemon, you can execute this command:

```
[master@fedora-server ~]$ sudo systemctl start smb
```

And to stop the service, type this:

```
[master@fedora-server ~]$ sudo systemctl stop smb
```

After making any configuration changes to Samba, you can restart it with this command to make the changes go into effect:

```
[master@fedora-server ~]$ sudo systemctl restart smb
```

Out of the box, the **smb** service might not automatically start up with the next system reboot. You can use the **`systemctl`** command on modern Linux distros to make sure that the **smb** service unit is enabled to automatically start up when the system boots, as follows:

```
[master@fedora-server ~]$ sudo systemctl enable smb
```

TIP Debian-based distros such as Ubuntu refer to the Samba daemon service unit as **smbd** instead of **smb**, which is used in the RPM world. Therefore, to start the Samba service in Ubuntu, you would instead type the following:

```
$ sudo systemctl start smbd
```

Creating a Share

We will walk through the process of creating a share under the **/tmp** directory to be shared on the Samba server. We'll first create the directory to be shared and then edit Samba's configuration file (**/etc/samba/smb.conf**) to create an entry for the share.

If you have any of the graphical desktop environments installed and running, you can often use some tools built into the environment to complete simple file-sharing tasks for Samba or use any of the available web-based utilities to manage your Samba server. However, it is useful for you to understand how to configure Samba in its rawest form, which makes it easier to understand what the GUI tools are doing in the back-end. Besides, you never know when you might be stranded in the Amazon jungle without any nice GUI configuration tools. So let's get on with it:

1. Create a directory under the **/tmp/** folder called **testshare**:
   ```
   [master@server ~]$ mkdir /tmp/testshare
   ```
2. Create some empty files (**foo1**, **foo2**, **moo3**) under the directory you created in step 1:
   ```
   [master@server ~]$ touch /tmp/testshare/{foo1,foo2,moo3}
   ```

3. Set up the permissions on the **testshare** folder so that its contents can be browsed by other users on the system:

```
[master@server ~]$ chmod -R 755 /tmp/testshare/*
```

4. As an administrator, open up Samba's configuration file (**/etc/samba/smb.conf**) for editing in any text editor of your choice. Then scroll all the way to the end of the file and append the entry listed next at the bottom of the file. (Please omit the line numbers 1–5, which are added *only* to aid readability.)

```
1. [samba-share]
2. comment=This folder contains shared documents
3. path=/tmp/testshare
4. public=yes
5. writable=no
```

   - Line 1 is the name of the share (or "service" in Samba parlance). This is the name that SMB clients will see when they try to browse the shares stored on the Samba server.
   - Line 2 is just a comment that users will see next to a share when browsing.
   - Line 3 is important. It specifies the location on the file system that stores the actual content to be shared.
   - Line 4 specifies that no password is required to access the share (access the share means "connecting to the service" in Samba-speak). The privileges on the share will be translated to the permissions of the guest account. If the value were set to **no** instead, the share would not be accessible by the general public but by authenticated and permitted users only.
   - Line 5, with the value of the directive set to **no**, means that users of this service may not create or modify the files stored therein.

TIP Samba's configuration file has options and directives that are too numerous to cover here. But you can learn more about the other possible options by reading the man page for **smb.conf** (`man smb.conf`).

5. Save your changes to the **/etc/samba/smb.conf** file and then exit the editor.

   Note that we have accepted all the other default values in the file. You may want to go back and personalize some of the settings to suit your environment.

   One setting you might want to change quickly is the **workgroup** directive, which defines the workgroup. It controls what workgroup your server will appear to be in when queried by clients or when viewed in Windows network browser.

   Also note that the default configuration may contain other share definitions. You should comment out (or delete) those entries if it is not your intention to include them.

6. Use the **testparm** utility to check the **smb.conf** file for internal correctness (that is, absence of syntax errors). Study the output for any serious errors, and try to fix them by going back to correct them in the **smb.conf** file. Type the following:

```
[master@server ~]$ testparm -s | less
...<OUTPUT TRUNCATED>...
[samba-share]
comment = This folder contains shared documents
path = /tmp/testshare
guest ok = Yes
```

7. Now restart (or start) Samba to make the software acknowledge your changes. You can use the **systemctl** command to restart the **smb** server (on an RPM-based distro like Fedora) by typing this:

```
[master@fedora-server ~]$ sudo systemctl restart smb
```

We are done creating our test share. In the next section, we'll attempt to access the share.

TIP On a system with SELinux running in enforcing mode (such as Fedora, RHEL, CentOS, and so on), you might need to properly label (with the correct security context) new files and directories on the Samba server to allow the remote clients to mount and access the Samba shares remotely.

To permanently label a directory so that SELinux allows Samba to read and write to it (on behalf of clients), you have to label it with the **samba_share_t** SELinux context. Use the **semanage** command to add a new SELinux record to do this, which for our example is the **/tmp/testshare** directory:

```
# semanage fcontext -a -t samba_share_t "/tmp/testshare(/.*)?"
```

To immediately apply the SELinux label that you just created to the **/tmp/testshare** directory and its content, use the **restorecon** command, like this:

```
# restorecon  -v -R  /tmp/testshare/
```

TIP On older versions (pre-**systemd**) of Debian-based distributions, (such as Ubuntu) you can restart the **smb** daemon by running the following:

```
master@ubuntu-server:~$ sudo  service samba  restart
```

Using smbclient

The **smbclient** program is a command-line tool that provides FTP-like access to SMB/CIFS resources. You can use this utility to connect to Samba servers or even to actual Microsoft Windows servers. **smbclient** is a flexible program and can be used to browse other servers, send and retrieve files from them, or even print to them. As you can imagine, this is also a great debugging tool, since you can quickly and easily check whether a new Samba installation works correctly without having to find a Windows client to test it.

In this section, we'll show you how to do basic browsing, remote file access, and remote printer access with **smbclient**.

NOTE The **smbclient** program is packaged separately in Ubuntu. You'll have to install it explicitly by running a command like the following:

```
master@ubuntu-server:~$ sudo apt-get -y install  smbclient
```

Browsing a Server

With so many graphical interfaces around, we've come to equate browsing with "point-and-click." But when your intention is simply to find out what a server has to offer, it's not enough of a reason in itself to support or run an entire GUI stack.

Using **smbclient** with the **-L** option allows you to view the service offerings of a Windows file server or Samba server. Here's the format of the command:

```
$ smbclient -L hostname
```

Here, ***hostname*** is the name of the server. For example, if we want to see what the local host (the Samba service running locally) has to offer, we type this:

```
$ smbclient -L localhost
```

You will be prompted for a password, but you can press ENTER to complete the command.

To list the shares on the Samba server again without being prompted for a password, you can use the **-U%** (or **-N**) option. This implies that you want to be authenticated as the guest user, which has no password. Type the following:

```
[master@server ~]$ smbclient -U% -L localhost
Sharename       Type      Comment
---------       ----      -------
print$          Disk      Printer Drivers
samba-share     Disk      This folder contains shared documents
IPC$            IPC       IPC Service (Samba Server Version 4*)
....<OUTPUT TRUNCATED>....
```

Notice the presence of the share we created earlier in line 4 of the preceding output.

Remote File Access

The **smbclient** utility allows you to access files on a Windows server or a Samba server with a command-line hybrid UNIX/FTP client interface.

For its most straightforward usage, you'll simply run the following:

```
$ smbclient   //server/share_name
```

Here, ***server*** is the server name (or IP address) and ***share_name*** is the share name to which you want to connect. By default, Samba automatically sets up all users' home directories as shares. For instance, the user *yyang* can access her home directory on the server named **fedora-server** by browsing **//fedora-server/yyang**.

TIP Watch out for the orientation of the / (forward slash) character used with the **smbclient** command to specify the server and share names! If you are used to doing similar things in Windows environments, you might absentmindedly use the \ (backslash) character instead!

The following are some command-line parameters you might need to use with **smbclient** to connect to a server:

Parameter for smbclient	Description
-I *destIP*	The destination IP address to which you want to connect.
-U *username*	The user you want to connect as. This will be used instead of the user you are logged in as.
-W *name*	Sets the workgroup name to ***name***.
-D *directory*	Starts from ***directory***.

Once connected, you'll be able to browse directories using the **cd**, **dir**, and **ls** commands. You can also use **get**, **put**, **mget**, and **mput** to transfer files back and forth. The online help explains the commands in detail. Type **help** at the prompt to see what is available.

TIP If your Samba server has a host-based or network-based firewall protecting it, don't forget to open the necessary ports to allow external systems to access the Samba services running on your server.

On Red Hat–like systems such as Fedora that use **firewalld**, you can temporarily punch a hole in the host-based firewall by running this:

```
$ sudo firewall-cmd --add-service=samba
```

And on Debian-like systems such as Ubuntu that use the Uncomplicated Firewall (UFW) to manage host-based firewall rules, you can instead run this:

```
$ sudo ufw allow Samba
```

Let's attempt an actual connection to the share we created earlier (**samba-share**). To demonstrate the process, the connection will be made from a different host named *clientB*.

We'll use the **smbclient** utility to connect to the server, connecting as a guest by specifying the **-U%** option. After connecting, we will be dropped down to an **smb** shell with the prompt `smb: \>`.

While we're connected, we'll do a listing of the files available on the share using the **ls** command. Then we'll try to download one of the files that resides on the share using the FTP-like command **get**.

Finally, we'll end the connection using **quit**. A sample session on clientB connecting to the host, serverA, is shown here:

```
[master@clientB ~]$ smbclient -U% //serverA/samba-share

smb: \> ls
  .                   D        0  Sun Mar 24 15:27:42 2026
  ..                  D        0  Sun Mar 24 15:27:14 2026
  foo1                A        0  Sun Mar 24 15:27:42 2026
  foo2                A        0  Sun Mar 24 15:27:42 2026
  moo3                A        0  Sun Mar 24 15:27:42 2026
 ….
smb: \> get foo2
getting file \foo2 of size 0 as foo2 (0.0 KiloBytes/sec) ...
smb: \> quit
```

The file (**foo2**) that was downloaded from serverA should be in the current working directory on the local file system of clientB.

TIP If you can't access the shares on your Samba server on Red Hat–like distros such as Fedora, it is possible that the SELinux security subsystem is getting in the way. To troubleshoot, you can temporarily put SELinux in permissive mode to eliminate it as being the cause of any issues. You can do this by running the following:

```
# setenforce  0
```

Mounting Remote Samba Shares

If your kernel is configured to support the SMB file system (as are most kernels that come with modern Linux distributions), you can actually mount a Windows share or Samba share onto your local system in much the same way you would mount an NFS export or a local volume. This is handy for providing easy access to a large disk/storage on a remote server across the network.

While logged into clientB, you can use the **mount** command with the proper options to mount a Samba share that resides on serverA.

1. First, create the mount point if it does not exist:

   ```
   [master@clientB ~]$ sudo mkdir -p /mnt/smb
   ```

2. Then run the **mount.cifs** command (via **mount -t cifs**) to do the actual mounting:

   ```
   [master@clientB ~]$ sudo mount -t cifs -o guest \
    //serverA/samba-share /mnt/smb
   ```

   Here, **//serverA/samba-share** is the remote share being mounted, and **/mnt/smb** is the local mount point.

   If the remote share that you want to mount is protected with a username/password combination, you can supply the username/password along with the **mount** command like this example using the user yyang's account:

   ```
   # sudo mount -t cifs -o username=yyang,password=19gan19 \
    //serverA/samba-share /mnt/smb
   ```

   To unmount this directory, use the **umount** command:

   ```
   [master@clientB ~]$ sudo umount /mnt/smb
   ```

   On Debian-based distros such as Ubuntu, you might have to install the **cifs-utils** package, if it is not already installed, so that you can use the **mount.cifs** command (via **mount -t cifs**). This can be done by running this command:

   ```
   master@ubuntu-server:~$ sudo apt-get install cifs-utils
   ```

Samba Users

When configured to do so, Samba will honor requests from users that are stored in other databases that it knows about. The user databases can, in turn, be stored in various back-ends such as LDAP (**ldapsam**) and Trivial database (**tdbsam**). Versions of Samba earlier than version 3.0.23 also supported exotic back-ends such as XML (**xmlsam**) and MySQL (**mysqlsam**).

Here, we will add a sample user that already exists in the local **/etc/passwd** file to Samba's user database. We will accept and use Samba's native/default user database back-end (**tdbsam**) for demonstration purposes.

Creating Samba Users

Let's create a Samba entry for an existing user (yyang). We will also set the user's Samba password.

Use the **pdbedit** command to create a Samba entry for the user yyang, and choose a good password when prompted to do so:

```
[master@server ~]$ sudo pdbedit -a yyang
new password:
retype new password:
…<OUTPUT TRUNCATED>...
Unix username:        yyang
User SID:             S-1-5-21-3312929510-1157075115-2701122738-1000
Full Name:            Ying Yang
```

The new user will be created in Samba's default user database, **tdbsam**. On Red Hat–like distros, the database file is **/var/lib/samba/private/passdb.tdb**.

With a Samba user created, you can make the shares available only to authenticated users, such as the one we just created for the user yyang.

If the user yyang now wants to access a resource on the Samba server that has been configured strictly for her use (a protected share or nonpublic share), the user can issue the **smbclient** command shown here, for example:

```
[master@clientB ~]$ smbclient  -U yyang  -L  //serverA
```

It is, of course, also possible to access a protected Samba share from a native Microsoft Windows box. You'll need to supply the proper Samba username and corresponding password when prompted on the Windows system.

Allowing Null Passwords

If you need to allow users to have no passwords (which is a bad idea, by the way, but for which you might have a legitimate use case), you can do so by using the **pdbedit** program with the **-c** option to set the "No password required (N)" flag, like so:

```
$ sudo pdbedit -c '[N ]' username
```

Here, ***username*** is the name of the user whose password you want to set to empty.

For example, to allow the user yyang to access a share on the Samba server with a null password, type the following:

```
[master@server ~]$ sudo pdbedit -c '[N ]'  yyang
```

You can also do this by using the **smbpasswd** program, like this:

```
[master@server ~]$ smbpasswd -n yyang
```

Changing Passwords with smbpasswd

Users who prefer the command line over the GUI tools can use the **smbpasswd** command to change their Samba passwords. This program works just like the regular **passwd** program, except this program does *not* update the **/etc/passwd** file by default; instead, it updates Samba's user/password database.

smbpasswd uses the standard Windows protocol/semantics for communicating with the server regarding password changes. Therefore, by using the right combination of parameters and options, you can coax the tool into changing a user password on a remote native Windows server—from a Linux-based host!

To change the user yyang's Samba password, issue this command:

```
[master@server ~]$ sudo smbpasswd  yyang
```

Samba can be configured to allow regular users to run the **smbpasswd** command themselves to manage their own passwords; the only caveat is that they must know their previous/old password.

TIP You can also use the **pdbedit** tool to change a user's Samba password. For example, to update the user yyang's Samba password using **pdbedit**, type
`$ sudo pdbedit -a -u  yyang`.

Using Samba to Authenticate Against a Windows Server

Thus far, we've been talking about using Samba in the Samba/Linux world. Or, to put it literarily, we've been using Samba in its native environment, where it is lord and master of its *domain* (no pun intended). What this means is that our Samba software stack, in combination with the Linux-based server, has been responsible for managing all user authentication and authorization issues.

The simple Samba setup that we created earlier in the chapter had its own user database, which mapped the Samba users to real Linux/UNIX users. This allowed any files and directories created by Samba users to have the proper ownership contexts. But what if we wanted to deploy a Samba server in an environment with existing Windows servers that are being used to manage all users in the domain? And we don't want to have to manage a separate user database in Samba? Enter the **winbindd** daemon!

NOTE Samba 4 is the latest and next-generation version of Samba. Among its many enhancements, Samba 4 implements an Active Directory (AD) compatible domain controller. It natively supports AD logon and administrative protocols (Kerberos, LDAP, RPC, and so on). This means that all current Windows clients can be transparently joined to a Samba 4 domain running on a Linux/UNIX-based Samba 4 server, and they will behave exactly as they would if they were members of native Windows AD domain.

winbindd Daemon

The **winbindd** daemon is used for resolving user accounts (users and groups) information from native Windows servers. It can also be used to resolve other kinds of system information.

It is able to do this through its use of **pam_winbind** (a PAM module that interacts with the **winbindd** daemon to help authenticate users), **libnss_winbind** (**winbind**'s Name Service Switch library) facility, and other relevant subsystems.

The process of setting up a Linux-based machine to consult a Windows server for its user authentication needs is also referred to as setting up Samba as a member server in an AD environment.

The steps to achieve this are straightforward and can be summarized in this way:

1. Install the **winbind** software if it isn't already installed.
2. Configure Samba's configuration file (**smb.conf**) with the proper directives.
3. Add **winbind** to the Linux system's name service switch facility (**/etc/nsswitch.conf**).
4. Join the Linux/Samba server to the Windows domain.
5. Test things out!

Here we present a sample scenario in which a Linux server named *serverA* will use a Windows server for its user authentication needs. The Samba server is going to act as a Windows domain member server.

The Windows server in this scenario is configured as a domain controller. Its hostname is *win-server* and its IP address is 192.168.1.100. The short Windows domain name is **WINDOWS-DOMAIN**.

We have deliberately excluded any share definitions in this sample Samba configuration, and as such you'll have to create or specify your own (if you require it; see the earlier parts of the chapter for how to do this). Let's break down the process in better detail:

1. Install the **winbind** package if it isn't already installed. On RPM-based systems such as Fedora, CentOS, and openSUSE, you can install it by issuing this command:

   ```
   [master@fedora-server ~]$ sudo dnf -y install samba-winbind
   ```

   On a Debian-based distro such as Ubuntu, you can install the package using this command:

   ```
   master@ubuntu-server:~$ sudo apt-get install winbind
   ```

2. Create an **/etc/samba/smb.conf** file similar to this one:

   ```
   #Sample smb.conf file for Linux AD Member Server
   [global]
      workgroup =  WINDOWS-DOMAIN
      password server = 192.168.1.100
      security = ads
      realm = WINDOWS-DOMAIN.COM
      dedicated keytab file = /etc/krb5.keytab
      kerberos method = secrets and keytab

      idmap config * : backend = tdb
      idmap config * : range = 1000-9999
      idmap config WINDOWS-DOMAIN : backend = ad
      idmap config WINDOWS-DOMAIN : schema_mode = rfc2307
      idmap config WINDOWS-DOMAIN : range = 10000-99999
   ```

```
winbind nss info = rfc2307
winbind trusted domains only = no
winbind use default domain = yes
winbind enum users  = yes
winbind enum groups = yes
winbind refresh tickets = Yes
winbind expand groups = 4
winbind normalize names = Yes
domain master = no
local master = no

vfs objects = acl_xattr
map acl inherit = Yes
store dos attributes = Yes
```

3. Edit the **/etc/nsswitch.conf** file on the Linux server so that it will have entries similar to this one:

```
passwd: files winbind
shadow: files winbind
group: files winbind
```

4. You will need to tell whatever subsystem (NetworkManager, systemd-resolved, Resolvconf, and so on) is in charge of managing your network name resolution routines your DNS search order and your name server.

   In other words, you want to tell your resolver routines that the remote Windows server is the primary DNS server and the domain search suffix matches the name of the Windows AD realm. The **/etc/resolv.conf** file or its equivalent should have entries similar to this:

```
search windows-domain.com
nameserver 192.168.1.100
```

5. Join the Samba server to the Windows domain using the **net** command. Assuming the Windows Domain Administrator account password is *windows_administrator_password*, type the following:

```
[master@fedora-server ~]$ sudo net join  -w WINDOWS-DOMAIN \
-S 'win-server'  -U  Administrator%windows_administrator_password
Using short domain name --  WINDOWS-DOMAIN
Joined 'FEDORA-SERVER' to dns domain 'WINDOWS-DOMAIN.COM'
```

6. On Red Hat–like systems, you can use the **authselect** utility to configure the other necessary user information and authentication sources. Launch the tool from the command line, pass it the **winbindd** option, and then apply the changes:

```
[master@fedora-server ~]$ sudo authselect select winbind
```

   Apply the changes by running the following:

```
[master@fedora-server ~]$ sudo   authselect  apply-changes
```

7. On Linux distributions using **systemd** as the service manager, you can start the **winbind** daemon and enable it's auto-startup using the **systemctl** command:

```
[master@fedora-server ~]$ sudo systemctl --now enable winbind
```

8. Use the **wbinfo** utility (provided with the **samba-winbind-clients*.rpm** package) to list all users available in the Windows domain to make sure that things are working properly:

```
$ wbinfo -u
```

That's it. The Linux server will now be able to consult and query the Windows Server for its user authentication needs.

Troubleshooting Samba

The following are a few typical solutions to simple problems you might encounter with Samba:

- *Restart Samba.* This might be necessary because either Samba has entered an undefined state or (more likely) you've made major changes to the configuration but forgot to reload Samba so that the changes take effect.
- *Make sure the configuration options are correct.* Errors in the **smb.conf** file are typically in directory names, usernames, network addresses, and hostnames. A common mistake occurs when a new client is added to a group that has special access to the server, but Samba isn't told the name of the new client being added. Don't forget that for syntax-type errors, the **testparm** utility is your ally.
- *Log files.* Samba can be configured to generate as few or as many logs as you want, and with varying levels of detail. The log files are your friend when troubleshooting Samba issues. Most distros store Samba-specific log files under the **/var/log/samba/** directory. Keep a watchful eye on the logs! You can also use the **journalctl** utility to view logs.

Summary

This chapter discussed the process of installing, configuring, and managing Samba so that your Linux-based server can integrate with a Windows-based network. Samba is a powerful application with the potential to replace Microsoft Windows servers dedicated to file and printer sharing and authentication services.

Reading through tons of documentation probably isn't your favorite way to pass the time, but you'll find the Samba documentation (https://wiki.samba.org) complete, helpful, and easy reading. At least skim through the files to see what's there, so you know where you can go to get additional information when you need it. With all the Samba texts available today (some free, some not), you should have everything you need to manage even the most complicated Samba environments.

This page intentionally left blank

CHAPTER

26 Distributed File Systems (DFS)

The NFS and SMB/CIFS network file systems discussed in earlier chapters are both venerable network file systems in the Linux/UNIX world. They both allow the sharing of file systems and other resources across the network. This chapter covers another class of network file systems that solves a slightly different set of problems in addition to the problems solved by traditional NFS and SMB network file systems. This class of network file systems is referred to as *Distributed File Systems (DFS).* DFS provides a mechanism to give you a unified view and access to file system resources located on different servers. The operative word here is "distributed," and it essentially means that the file system resources can be divided, shared, or distributed in a logical or physical manner across a network.

Plain-old vanilla network files systems do a wonderful job of making the local storage on a single server available to other client systems on a local area network (LAN). However, they do not often have the inherent ability to scale or grow easily beyond that single server or LAN. And they do not have the inherent ability to offer a unified (and consistent) view of a shared file system resource residing on multiple servers or across several physically or geographically disparate locations—to make all the data appear as a single resource to the end user.

NOTE We say "inherent" here because, in truth, some solutions/hacks do exist that can make the plain network file systems mimic the behavior of full-fledged DFS systems.

DFS Overview

"A picture," they say, "is worth a thousand words." We'll use a series of pictures to provide an overview of DFS. Suppose, for example, that we have a network of systems like the one depicted in Figure 26-1.

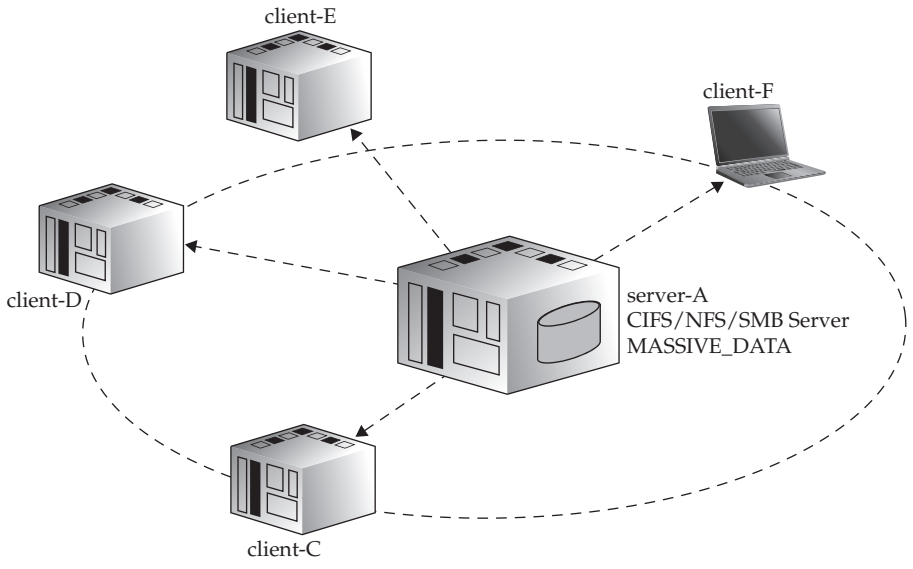


Figure 26-1 Traditional network file server and LAN clients

Our network consists of the following elements:

- A massive set of DATA (files, directories, and so on). In this example, all the DATA in its entirety is referred to as MASSIVE_DATA.
- A traditional NFS or Common Internet File System (CIFS) server hosting MASSIVE_DATA. This is server-A.
- A bunch of client systems and users that need to access MASSIVE_DATA hosted on server-A. These client systems are client-C, client-D, client-E, and client-F.
- With everything else being equal, server-A is fine and up to the task of serving MASSIVE_DATA to the clients—provided that the clients are connected to it via some high-speed network, such as a LAN.

In Figure 26-2, things are switched around a little: We've added more client systems. In addition to the elements and conditions shown in Figure 26-1, our new scenario has the following new elements:

- A bunch of new client systems—client-G, client-H, client-I, and client-J—that need access to MASSIVE_DATA on server-A.
- The new clients, however, are not directly connected to the same LAN as server-A. Instead they are connected over a WAN (Internet).
- The new clients are located in physically and geographically different locations.

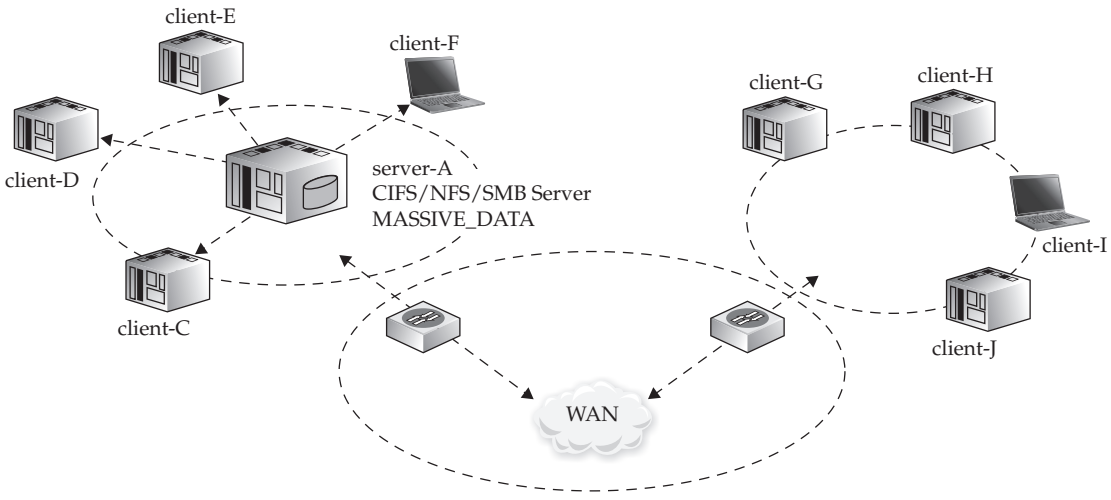


Figure 26-2 Traditional network file server, LAN clients, and WAN clients

Obviously, nothing much has changed as far as server-A is concerned—just a bunch of new clients requiring access to MASSIVE_DATA.

However, a problem is created by the new requirements: even though server-A may still be up to the task of making its file system available to the new additional clients, the intervening network between the server and the new clients might not be up to the task. The intervening network might be too slow, too congested, too unpredictable, and so on. It is also possible that server-A was simply not built to handle the additional demands and load of so many new clients.

The solution that DFS offers to these problems is shown in Figure 26-3 and described in the following scenario:

- Place an additional file server (server-B) in physical proximity to the clients that require access to MASSIVE_DATA.
- Ensure that a copy of MASSIVE_DATA exists on all file servers regardless of where they're located.
- Ensure that clients have fast, easy, and consistent access to MASSIVE_DATA.
- Ensure that bits and bytes that make up MASSIVE_DATA remain consistent and uncorrupted regardless of from where the client is accessing or modifying the data.
- Ensure that any changes to MASSIVE_DATA are immediately reflected across all the copies, regardless of where they physically reside.

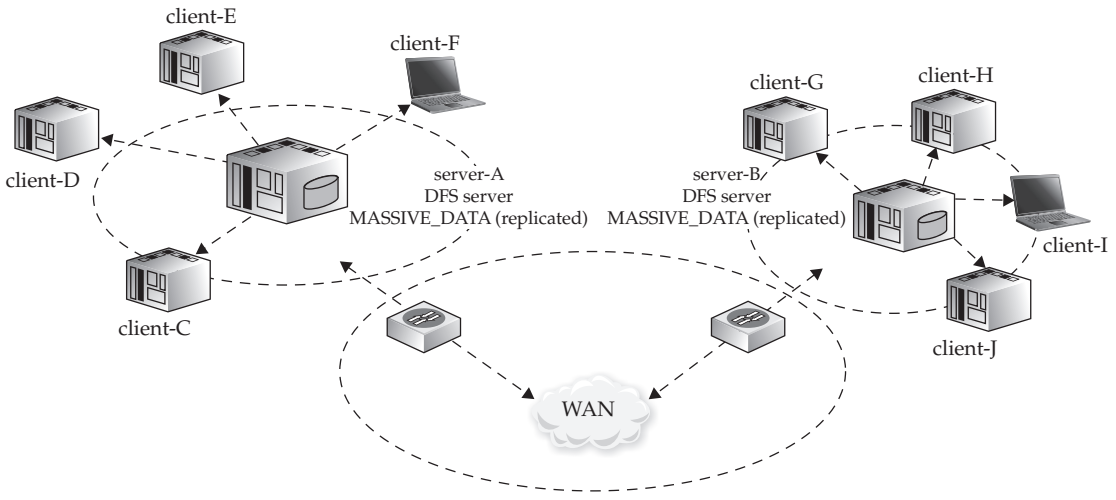


Figure 26-3 Distributed File Systems network

DFS Implementations

Numerous DFS solutions exist to tackle the scenarios described so far. These solutions vary in so many degrees: they vary in complexity, ease of manageability, ease of configuration, ease of deployment, supported platform, maturity, application, and, of course, features. The implementations also differ because they can include special features that make them more suitable for specific applications or workloads. Following are some DFS implementations:

- **GlusterFS (www.gluster.org)** This popular open source distributed/clustered file system is easy to set up and use. The source code is under active development. Packaged binaries are readily available for most of the popular Linux distros. It's well suited for high-performance and virtualized cloud (cloud computing) type workloads and applications.
- **Lustre (http://lustre.org)** This high-performance DFS implementation is frequently used for clustering types of workloads and LAN-type environments. Its architecture makes it more complicated to install, configure, and maintain in comparison to other DFS solutions. A standard Lustre setup includes these components: Metadata Server (MDS), Metadata Target (MDT), Object Storage Server (OSS), Object Storage Target (OST), and Lustre clients. For some features, Lustre requires a specially patched Linux kernel.
- **Microsoft DFS (MS-DFS)** Microsoft's implementation is a mature product that is relatively easy to install in a pure Windows environment. The open source project Samba can emulate some of the features of the proprietary MS-DFS in its implementation of the CIFS/SMB protocols.

- **OpenAFS (www.openafs.org)** One of the older DFS implementations, OpenAFS is robust and well supported on multiple platforms—Windows, Linux, and macOS. However, it is not easy to install and configure. It relies heavily on Kerberos.
- **CEPH (www.ceph.com)** Ceph is a mature and industrial-grade storage platform. It is both distributed and clustered and provides interfaces for most common DFS use cases. The interfaces provided to applications for storing data in a Ceph cluster are Ceph object storage, Ceph block device, and Ceph file system. Because Ceph is an object storage platform at its core, all data is stored as objects in a flat namespace; this means there is no traditional hierarchy of directories (except via one of the interfaces)!

  A typical Ceph cluster implementation consists of the following daemons: Monitors (ceph-mon), managers (ceph-mgr), Object Storage Daemon (ceph-osd), and Metadata servers (ceph-mds).

In the following sections, we'll cherry-pick and discuss and configure the popular GlusterFS DFS implementations.

DFS Terminology

The following are common concepts and terms used in the DFS world and DFS discussions:

- **brick** In DFS parlance, a physical server/system whose local storage resource is contributing to the overall storage capacity of the DFS is often referred to as a "brick." Combining two or more bricks in different fashions forms the basics of DFS systems. Clients or hosts access the file system resources stored collectively on bricks and need not know (or care) that they are accessing the data on one or multiple bricks.
- **metadata** Refers to all the other characteristics of data (or a file) except the actual data itself. This can include the file size, the file permissions, timestamps, filename, and other attributes that can be used to describe the file. Different DFS implementations typically have a special manner with which they organize metadata information for the data they host. Some DFS implementations require a separate metadata server to be dedicated for the function of managing metadata information for the storage.
- **fault tolerance** This is a characteristic or feature that a DFS may or may not provide. It allows transparent and continued access to the shared resource of the DFS in the event of a failure. Fault tolerance is akin to the peace of mind that comes with having multiple storage disks in a Redundant Array of Independent Disks (RAID) setup. In some RAID configurations, a system can continue

(*continued*)

functioning if a single disk (and sometimes multiple disks) breaks down. In the case of a fault-tolerant DFS involving multiple systems, the file system resources will continue to be available if one (or sometimes more) of the servers suddenly becomes unavailable or goes offline. Even though this may be a desirable feature in some DFS setups, it might not be useful in other setups where overall performance of the DFS is more important.

- **replication** Replication is the process of duplication of the contents of one file system onto another location. It is used for ensuring consistency and fault tolerance, and for improving accessibility to data. Replication can happen in one of two ways: *synchronously* or *asynchronously.* Synchronous replication can slow things down a bit, because it requires that certain file operations (read, write, and so on) that occur on one of the nodes be performed concurrently/completely on other nodes in order to be deemed successful. Asynchronous replication, on the other hand, is better suited for use on slow connections, because file system operations can happen on a node and be deemed successful while the other nodes in the setup sort of catch up later on.
- **POSIX compliance** POSIX, the Portable Operating System Interface for UNIX, is a family of Institute of Electrical and Electronics Engineers (IEEE) standards that defines software compatibility with variants of the UNIX operating systems (such as Linux). For our DFS discussion here, POSIX file system compliance in any DFS implementation simply means that the DFS software is able to present and preserve UNIX file system semantics, such as timestamps, case-sensitive file-naming, permissions, and so on, for the data stored on it. POSIX compliance in a DFS implementation is a good feature to have, because it means that existing users and applications won't notice any differences in the way they interact with data stored on the DFS volume.

GlusterFS

GlusterFS is a popular DFS implementation backed by Red Hat, Inc. The GlusterFS software is available as precompiled packages for several Linux distros, but as with most Free and Open Source Software (FOSS) projects, you can always download the source code, compile, and build it yourself. Compiling and building it yourself might be necessary if you want the very latest version of the software—which your distro maintainers may not yet have available.

In the following sections, we'll create a small DFS environment consisting of two servers and one client. The two servers will host a distributed replicated volume. This means that contents of the **glusterfs** volumes on both servers will be identical—this will provide redundancy similar to when using physical disks in a RAID 1 setup (mirroring). This helps to ensure high availability and reliability.

Because our sample GlusterFS environment consists of three different systems (server-A, server-B, and client-C), we have broken down the steps to build our GlusterFS

network into different sections and subsections. So wherever a section applies only to the servers or only to one of the servers, we label that section accordingly. Similarly, when the steps apply only to the client system, we label that section as such.

Ready? Let's go!

Installation

We will install the GlusterFS software and its dependencies on our sample servers and client systems running Fedora. The steps and commands to run are very similar on other RPM-based Linux distros such as RHEL, CentOS, and so on. Try as much as possible to deploy the same version/release of the gluster stack among a given set of servers and clients.

Install GlusterFS on server-A and server-B While logged into the system as a user with administrative privileges, at the console type the following:

```
[master@server-* ~]$ sudo dnf -y install glusterfs glusterfs-server
```

Install GlusterFS on the Client System (client-C) On the client systems that need to access only the data exported by gluster servers, we need to install the core GlusterFS package as well as the native GlusterFS client package. To install these, type the following:

```
[master@client-C ~]$ sudo dnf -y install glusterfs glusterfs-fuse
```

The client component of gluster is provided by the **glusterfs-client** package on Debian-based distros like Ubuntu. This can be installed in Ubuntu by running the following:

```
[master@ubuntu-client-C ~]$ sudo apt  install glusterfs-client
```

Managing Gluster

As with most of the other client/server–based services discussed in this book, managing the gluster service involves starting, stopping, and configuring the service.

Name Resolution It's nice and convenient for the GlusterFS software stack to be able to translate the hostnames of other systems reliably into their IP addresses. You can do this on the cheap by making sure the **/etc/hosts** file on all the systems contains the necessary name and IP address maps for all participating systems or by ensuring that all hostnames are resolvable via DNS (see Chapter 17). We will go the easy route here and make sure that the following entries that are applicable for our demo environment are in the **/etc/hosts** file of all three systems:

```
192.168.0.67       server-A
172.16.0.69        server-B
192.168.0.246      client-C
```

Start and Stop Gluster on server-A and server-B Check the status of the **glusterd** daemon by running the following:

```
[master@server-* ~]$ systemctl status glusterd
```

To stop **glusterd**, type this:

```
[master@server-* ~]$ sudo systemctl stop  glusterd
```

Once you've confirmed that the daemon is not currently running, you can start **glusterd** like so:

```
[master@server-* ~]$ sudo systemctl start  glusterd
```

Create a Trusted Storage Pool In GlusterFS parlance, a trusted storage pool consists of all the servers or bricks in your setup that you trust to be part of your DFS. The notion of storage pools help in managing different aspects of a GlusterFS environment—for example, you can increase the total capacity of the DFS by adding new storage bricks to the storage pool.

TIP The hosts in a gluster DFS communicate via the TCP port number ranges 24007 to 24008 and 49152 to <49152 plus number_of_bricks minus 1>. Be sure that these ports are open on any local or external firewalls between the GlusterFS servers and clients. For example, on an RPM-based distro gluster setup with 10 bricks using the **firewalld** firewall manager, you can open up these ports on the servers by running the following:

```
$ sudo firewall-cmd  --add-port=24007-24008/tcp \
 --add-port=49152-49161/tcp  --permanent
$ sudo firewall-cmd  --reload
```

For testing and troubleshooting purposes, you can also temporarily disable the entire firewall subsystem (**iptables**, **nftables**, **firewalld**, or **ufw**) on the server.

From server-A, add server-B to the storage pool:

```
[master@server-A ~]$ sudo gluster peer probe server-B
```

Check the status of the peers in the storage pool you just created:

```
[master@server-A ~]$ gluster peer status
```

TIP If you ever need to, you can remove a server (for example, server-B) from the storage pool by typing this:

```
[master@server-A ~]$  sudo gluster peer detach server-B
```

Create a Distributed Replicated Volume We will create and use directories under the /**data** directory as the storage space to be distributed among the trusted peers in our DFS. We will then create a replicated volume named **MASSIVE_DATA**.

On server-A, create the parent and subdirectory named /**data**/**A**:

```
[master@server-A ~]$ sudo mkdir -p /data/A
```

On server-B, create the parent and subdirectory named /**data**/**B**:

```
[master@server-B ~]$ sudo  mkdir -p /data/B
```

From server-A, create the **MASSIVE_DATA** volume:

```
[master@server-A ~]$ sudo gluster volume create MASSIVE_DATA replica 2 \
transport tcp  server-A:/data/A  server-B:/data/B  force
volume create: MASSIVE_DATA: success: please start the volume to access data
```

Use the **`info`** option with the **`gluster`** command to view information about the **glusterfs** volume you just created. This command can be issued from any of the gluster servers:

```
[master@server-* ~]$ sudo gluster  volume info MASSIVE_DATA
Volume Name: MASSIVE_DATA
Type: Replicate
Status: Created
Number of Bricks: 2
Transport-type: tcp
Bricks:
Brick1: server-A:/data/A
Brick2: server-B:/data/B
```

Start the GlusterFS Volume on the Servers After the logical volume has been created, you need to start the volume to enable the clients to access the data stored therein. You can start the volume from any of the trusted peers in the storage pool.

To start the **MASSIVE_DATA** volume, type the following:

```
[master@server-* ~]$ sudo gluster volume start MASSIVE_DATA
```

Mount the GlusterFS Volume from the Client (client-C) Client systems that need to access the **glusterfs** volume can do so using different methods and protocols. The recommended way is to use the so-called "native" **glusterfs** client to mount remote **glusterfs** volumes. This method provides the best performance, because it utilizes built-in optimizations for the software. However, it might not always be possible to use the native **gluster** client.

You can also access **glusterfs** volumes by enabling and configuring the appropriate protocol from the server side and sharing/exporting the **glusterfs** mount point as you normally would. For example, you can make the contents of the **glusterfs** mount point on our sample server-B available via the CIFS/SMB protocol to Windows, macOS, and Linux clients by creating a Samba (see Chapter 25) share definition similar to the following:

```
[MASSIVE_DATA]
      path = /data/B
      comment = Samba share on gluster DFS
      browseable = yes
      writable = yes
```

Similarly, you can make the **glusterfs** volume available to other Linux client systems that are capable of speaking NFS (see Chapter 24). This can be done on our sample **glusterfs** server-A by creating an entry similar to the following in the NFS configuration file (**/etc/exports**):

```
# /etc/exports file for server-A
/data/A  192.168.0.0/24(ro)
```

In this section, we'll access our remote **glusterfs** volume using the recommended native **glusterfs** client.

We will create and mount the remote **glusterfs** volume at the **/mnt/glusterfs/MASSIVE_DATA** directory of the client.

On the client system, create the mount point:

```
[master@client-C ~]$ sudo mkdir  -p /mnt/glusterfs/MASSIVE_DATA
```

Mount the exported volume on the **gluster** servers by running the following:

```
[master@client-C ~]$ sudo mount -t glusterfs server-A:/MASSIVE_DATA  \
 /mnt/glusterfs/MASSIVE_DATA/
```

Use the **df** command to run some basic checks on the remote volume:

```
[master@client-C ~]$ df -h /mnt/glusterfs/MASSIVE_DATA/
Filesystem              Size  Used    Avail  Use%  Mounted on
server-A:/MASSIVE_DATA  50G   746M    48G    2%    /mnt/glusterfs/MASSIVE_DATA
```

Access the GlusterFS Volume from the Client As soon as you've successfully mounted the remote **glusterfs** volume on the local client, the users on the client system can begin to read and write data onto the remote storage—provided, of course, that they have the correct file system permissions to do this.

From the client system, create some sample files on the **MASSIVE_DATA** volume:

```
[master@client-C ~$] sudo touch  /mnt/glusterfs/MASSIVE_DATA/{1,2,3,4}
```

Make sure that the files were properly created by performing a simple directory listing:

```
[master@client-C $ ls /mnt/glusterfs/MASSIVE_DATA/
1  2  3  4
```

TIP If you perform a directory listing on the **/data/A** and **/data/B** directories of server-A and server-B, respectively, you should see the same files that were created from the client replicated on both servers. Furthermore, if one of the servers (let's say server-A) were to suddenly go offline for whatever reason, the remaining server (server-B) would continue to host the replicated files as if nothing happened, and all this would be transparent to the client.

Summary

You were introduced to the world of distributed network file systems in this chapter. Among other things, a DFS provides a way of spreading data over multiple physical servers and making the data appear as one single file system resource to clients. Several DFS implementations exist today, both proprietary and open source variants. We provided a quick overview of some of the more popular and promising implementations that are readily available.

Finally, you learned how to install, configure, and manage a simple replicated DFS environment using the GlusterFS software. Our sample setup consisted of two servers and one client.

CHAPTER

27 Lightweight Directory Access Protocol (LDAP)

The Lightweight Directory Access Protocol (LDAP) has been referred to as many things, including "the best thing since sliced bread." However, it is actually a set of open protocols used to access, query, and modify centrally stored information over a network. LDAP is based on the X.500 standard (X.500 is an International Organization for Standardization [ISO] standard that defines an overall model for distributed directory services) but is a more lightweight version of the original standard. RFC 4511 explains the relationship like so: "LDAP provides access to distributed directory services that act in accordance with X.500 data and service models."

LDAP was developed by the University of Michigan in 1992 as a *lightweight* alternative to the Directory Access Protocol (DAP). LDAP itself does not define the directory service. It instead defines the transport and format of messages used by a client to access and maintain data in a directory (such as the X.500 directory). LDAP is extensible, relatively easy to implement, and based on an open standard. This chapter provides an introduction to the world of directory services, as implemented by OpenLDAP. Essential concepts governing the architecture and use of LDAP will be covered.

LDAP Basics

LDAP is a protocol for accessing a specially tailored database that is used for a variety of things, such as directory service. This directory is like a database of sorts and can be used to store all kinds of information. But unlike traditional databases, an LDAP database is especially suited for read, search, and browse operations instead of write operations. It is with *reads* that LDAP shines!

Some popular LDAP implementations are OpenLDAP suite, Microsoft Active Directory, 389 Directory Server, IBM Security Directory Server, and NetIQ eDirectory.

NOTE If we wanted to be pedantic, we would point out that some of these implementations are not pure/strict LDAP implementations because they don't strictly adhere to the LDAP standard, are missing parts of the LDAP schema, or simply have too many customizations.

LDAP Directory

Just as in the popular Domain Name System (DNS), the directory entries in LDAP are arranged in a hierarchical tree structure. As in most hierarchical structures, the further you go down the tree, the more precise the content stored therein. The hierarchical tree structure of LDAP is known formally as the *directory information tree (DIT).* The top of the directory hierarchy has a *root* element. The complete path to any node in the tree structure, which uniquely identifies it, is known as the *distinguished name (DN)* of the node or object.

Again, just as in DNS, the structure of an LDAP directory can reflect geographic and/or organizational boundaries. Geographic boundaries can be along country lines, state lines, city lines, or the like. Organizational boundaries can, for example, be along functional lines, departmental lines, or organizational units (OUs).

For example, suppose a company named Example, Inc., decides to structure its directory tree using a domain-based naming structure. This company has different subdivisions (OUs), such as the Engineering department, the Sales department, and the R&D department. The LDAP directory tree of the company is illustrated in Figure 27-1.

Here's the DN of a sample object in the directory tree shown in the figure:

```
dn: uid=yyang,ou=sales,dc=example,dc=org
```

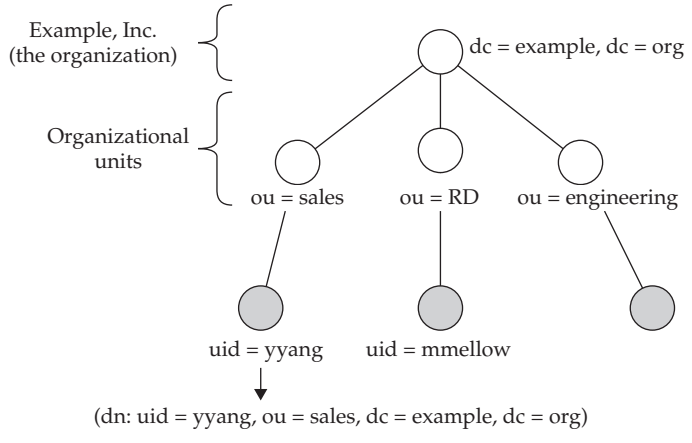


Figure 27-1 LDAP tree for example.org

Client/Server Model

As with most network services, LDAP adheres to the usual client/server paradigm. A typical interaction between the client and the server goes like this:

1. An LDAP client application connects to an LDAP server. This process is also referred to as "binding to a server."
2. Based on the access restrictions configured on the server, the LDAP server either accepts or refuses the bind/connection request.
3. Assuming the server accepts, the client has the choice of querying the directory server, browsing the information stored on the server, or attempting to modify/update the information on the LDAP server.
4. Again, based on access restrictions, the server can allow or deny any of the operations attempted by the client. In the event that the server cannot answer a request, it may forward or refer the client to another upstream LDAP server that may have a more authoritative response to the request.

Uses of LDAP

LDAP is a distributed directory service and can be used as storage for various types of information. Just about any kind of information can be stored in an LDAP directory—information as varied in nature as plain textual information, images, binary data, public key certificates, and so on. Over time, various LDAP schemas have been created to allow the storage of different data sources in an LDAP directory. Here are some example uses for LDAP:

- LDAP can serve as a complete identity management solution for an organization. It can provide authentication and authorization services for users.
- The information stored in DNS records can be stored in LDAP.
- LDAP can be used to provide "yellow pages" services for an organization (for instance, employees' contact information—phone numbers, addresses, departments, and so on).
- Mail routing information can be stored in LDAP.
- A Samba schema exists that allows a Samba server to store extensive object attributes in LDAP. This allows Samba to function as a drop-in replacement for Microsoft Windows domain controllers in environments where redundancy and replication are needed.

LDAP Terminology

If you are going to master LDAP-speak, you might as well know the essential LDAP technical jargon. This section defines some terms and explains some concepts that you will often encounter when dealing with LDAP.

- **Entry (or object)** One unit in an LDAP directory. Each entry is qualified by its distinguished name (DN). Here's an example:

  ```
  dn: uid=yyang,ou=sales,dc=example,dc=com
  ```

- **Attributes** These are pieces of information associated with an entry, such as an organization's address or employees' phone numbers.
- **`objectClass`** This is a special type of attribute. All objects in LDAP must have an **`objectClass`** attribute. The **`objectClass`** definition specifies which attributes are required for each LDAP object, and it specifies the object classes of an entry. The values of this attribute may be modified by clients, but the **`objectClass`** attribute itself cannot be removed. The **`objectClass`** definitions are themselves stored in *schema files.*
- **Schema** A collection of rules that determines the structure and contents of the directory. The schema contains the attribute type definitions, **`objectClass`** definitions, and other information. The schema lists the attributes of each object type and whether these attributes are required or optional. Schemas are usually stored in plain-text files.

  Following are a few examples of schemas:

  - **core.schema** Defines the basic LDAPv3 attributes and objects. It is a required core schema in the OpenLDAP implementation.
  - **inetorgperson.schema** Defines the **`inetOrgPerson`** object class and its associated attributes. This object can be used to store contact information.
- **LDIF** This stands for LDAP Data Interchange Format. It is a plain-text file for LDAP entries. Files used for importing or exporting data to and from an LDAP server should be in this format. The data used for replication among LDAP servers is also in this format.

OpenLDAP

OpenLDAP is the open source implementation of LDAP that runs on Linux/UNIX systems. OpenLDAP is a suite of programs, made up of **slapd**, **slurpd**, various utilities, and **libraries**, that implements the LDAP protocol along with various client- and server-side utilities.

Server-Side Daemons

The server side consists of two main daemons:

- **slapd** This is a stand-alone LDAP daemon that listens for LDAP connections from clients and responds to the LDAP operations it receives over those connections.
- **slurpd** This is a stand-alone LDAP replication daemon that is used to propagate changes from one **slapd** database to another. This daemon is used for synchronizing changes from one LDAP server to another. It is useful when more than one LDAP server is in use.

OpenLDAP Utilities

The OpenLDAP utilities are a set of command-line programs used for querying, viewing, updating, and modifying the data stored in the OpenLDAP directory. Some of the programs are listed and described in Table 27-1.

Utility	Description
ldapmodify	Used for modifying entries in LDAP. It accepts input either directly from the command line or via a file.
ldapadd	A hard link to the **ldapmodify -a** command. It is used to add new entries to an LDAP database. (The functionality provided by the **ldapadd** command can be obtained by adding the **-a** option to the **ldapmodify** command.)
ldapdelete	Used for deleting entries from an OpenLDAP directory.
ldappasswd	Sets the password for an LDAP user.
ldapsearch	Used for querying/searching an LDAP directory.
slapadd	Accepts input from an LDIF file to populate an LDAP directory. Located under the **/usr/sbin/** directory.
slapcat	Dumps the entire contents of the LDAP directory into an LDIF-type file. Located under the **/usr/sbin/** directory.
slapindex	Used for re-indexing the LDAP database according to the actual current database content. Located under the **/usr/sbin/** directory.
slappasswd	Used for generating properly hashed/encrypted passwords that can be used with various privileged directory operations. Located under the **/usr/sbin/** directory.

Table 27-1 OpenLDAP Utilities

Installing OpenLDAP

To get the OpenLDAP server and client components up and running, these packages are required on Fedora, RHEL, and CentOS systems:

- **openldap-2*.rpm** Provides the configuration files and libraries for OpenLDAP.
- **openldap-clients*.rpm** Provides the client programs needed for accessing and modifying OpenLDAP directories. This is only needed on the client side!
- **openldap-servers*.rpm** Provides the servers (**slapd**, **slurpd**) and other utilities necessary to configure and run LDAP.

We will use the **dnf** package manager to download and install the **openldap-servers** package on our sample Fedora server. The steps are listed here:

1. While logged in as root, first confirm which of the packages you already have installed by querying the RPM database:

```
[master@fedora-server ~]$ rpm -qa | grep -i openldap
openldap-2* ...
```

NOTE The operating system installation process of most Linux distributions will automatically include the base OpenLDAP software as a part of the minimum software installed. This is done so that the system can be configured as an LDAP client from the get-go without any additional hassle.

2. Predictably, our sample system already has the basic **openldap** libraries in place, so we will go ahead and install the OpenLDAP client and server packages using **dnf**:

```
[master@fedora-server ~]$ sudo dnf -y install openldap-servers \
openldap-clients
```

Once the installation completes successfully, you can go on to the configuration section.

Installing OpenLDAP in Ubuntu

The OpenLDAP server can be installed on Debian-based Linux distros, such as Ubuntu, by using Advanced Packaging Tool (APT). Here's the command to install the software:

```
master@ubuntu-server:~$ sudo apt-get -y install slapd
```

Among other things, the install process will helpfully start you off in setting up a basic LDAP server configuration by asking some questions (such as an admin password). The OpenLDAP server (**slapd**) process will also be automatically started after the installation.

The OpenLDAP client utilities on Debian-like distros are provided in the **ldap-utils*.deb** package. You can install this by running the following commands:

```
master@ubuntu-server:~$ sudo apt-get install ldap-utils
```

Configuring OpenLDAP

Depending on what you want to do with your directory, configuring your directory server can be a real pain or it can be a simple process. Setting up your directory is usually easy if you are working on a brand-new deployment, where you don't have to worry about any legacy issues, existing users or data, and similar issues. For environments with existing infrastructure, extra precautionary measures have to be taken.

CAUTION If you are deploying LDAP in an environment where you need to be concerned about backward-compatibility issues, legacy architectures, existing users, or existing data, you are advised to approach your OpenLDAP rollout with great caution. This can take months of planning (and *will* involve pain and suffering, according to this book's technical editor, David Lane) in some situations. The planning should include extensive testing and actual staging of the current environment on test systems.

Another important factor to consider when configuring your LDAP directory service is the structure of the directory. For example, before proceeding, you should have answers to the following questions: What are the organizational divisions in your establishment? Along what boundaries will the structure be built? How sensitive is the information you want to store in the directory? Will more than one LDAP server be required?

PAM Module, NSS Library, nslcd, and SSSD Daemon

A package named **nss-pam-ldapd*.rpm** on Red Hat–like systems such as Fedora provides two important modules/libraries, **pam_ldap** and **libnss_ldap**, as well as the **nslcd** daemon.

The **pam_ldap** module provides a means for Linux hosts to authenticate against LDAP directories. It allows PAM-aware applications to *authenticate* users using information stored in an LDAP directory. Examples of PAM-aware applications are the login program, some mail servers, some File Transport Protocol (FTP) servers, OpenSSH, and Samba.

The **libnss_ldap** library is a set of C library extensions that allows applications to *look up* users, groups, hosts, and other information by querying an LDAP directory. The module allows applications to look up certain information using LDAP and can function alongside traditional methods such as flat files, DNS, NIS, and NIS+.

The PAM modules and NSS libraries in the **nss-pam-ldapd** package can work in conjunction with other system daemons to provide a complete authentication and authorization solution. The LDAP name service daemon (**nslcd**) and the System Security Services Daemon (**sssd**) are popular examples of such system daemons. **nslcd** is more of a legacy daemon and is somewhat deprecated in favor of **sssd**. Also, **nslcd** is very LDAP specific, whereas **sssd** can provide access to multiple different account information sources, such as FreeIPA, files, LDAP, Active Directory, and so on. SSSD uses various sub-daemons (such as **sssd-ipa**, **sssd-ldap**, **sssd-ad**, and **sssd-files**) to manage access to remote directories and authentication mechanisms.

On systems where LDAP is to be used as a replacement for the traditional authentication mechanisms, the use of **nslcd** or **sssd** offers the following advantages:

- Helps avoid the need to load LDAP and SSL libraries in relevant programs
- Eases the load on the LDAP server, via the ability to cache looked-up information
- Provides better performance

The **nss-pam-ldapd** package provides the **nslcd** daemon, whose main configuration file is **/etc/nslcd.conf**. Modern RPM-based distros have the **sssd-common** package installed by default, which provides the **sssd** daemon. The main configuration file for **sssd** is **/etc/sssd/sssd.conf**.

To configure either **nslcd** or **sssd** to use LDAP, you will at a minimum need to set the location (URI) of the LDAP server, the LDAP search Base DN, as well as the user and group the daemon should run as.

Configuring slapd

The **slapd.conf** file is the configuration file for the **slapd** daemon. On Fedora and other Red Hat–like distros, the full path to the file is **/etc/openldap/slapd.conf**. In this section, we will dissect the default configuration file that comes with our Fedora server and discuss some of its interesting sections.

NOTE On Debian-like distros, the configuration file for **slapd** is located at **/etc/ldap/slapd.conf**.

Following is a truncated version of a **slapd.conf** file. Note that we've kept the comments in the file to a minimum and have eliminated the configuration directives that we don't need to address here. Only the stripped-down version of the file that is relevant to our current discussion is displayed. Line numbers have been added to aid readability.

```
1. # See slapd.conf(5) for details on configuration options.
2. # This file should NOT be world-readable.
3. #
4. include   /etc/openldap/schema/core.schema
5. include   /etc/openldap/schema/cosine.schema
6. include   /etc/openldap/schema/inetorgperson.schema
7. include   /etc/openldap/schema/nis.schema
8. #
9. pidfile        /var/run/openldap/slapd.pid
10. argsfile      /var/run/openldap/slapd.args
11. database      bdb
12. suffix       "dc=my-domain,dc=com"
13. rootdn       "cn=Manager,dc=my-domain,dc=com"
14. # Cleartext passwords, especially for the rootdn, should
15. # be avoided. See slappasswd(8) and slapd.conf(5) for details.
16. #
17. rootpw       {crypt}ijFYNcSNctBYg
18. #
19. # The database directory MUST exist prior to running slapd AND
20. # should only be accessible by the slapd and slap tools.
21. # Mode 700 recommended.
22. directory     /var/lib/ldap
```

Here are some items to note from the preceding listing:

- Lines 1–3 are comment entries. Any text after the pound sign (#) is a comment.
- Lines 4–7 are **`include`** statements. These **`include`** statements are used to instruct **slapd** to read additional configuration information from the files specified. In this case, the additional files being pulled in are the specified OpenLDAP schema files stored under the **/etc/openldap/schema/** directory. At a minimum, the **core.schema** file must be present.
- In line 9, the **`pidfile`** directive points to the path of the file that will hold **slapd**'s process ID.

- In line 10, the **`argsfile`** directive is used for specifying the path to a file that can be used to store command-line options used for starting **slapd**.
- In line 11, the **`database`** option marks the beginning of a new database instance definition. The value of this option depends on the back-end that will be used to hold the database. In our sample **slapd.conf** file, **`bdb`** (Berkeley DB) is used as the database type. Examples of other supported database back-end types are **`null`**, **`sql`**, **`passwd`**, and **`meta`**. Some database back-ends are described further in Table 27-2.
- In line 12, the **`suffix`** directive specifies the DN suffix of queries that will be passed to this particular database back-end. It defines the domain for which the LDAP server provides information or for which the LDAP server is authoritative. This entry should be changed to reflect your organization's naming structure.
- In line 13, the **`rootdn`** directive specifies the DN of the superuser for the LDAP directory. This user is to the LDAP directory what the root superuser is to a Linux system. The user specified here is not subject to any access controls or administrative restrictions for operations on the database in question. The DN specified here need not exist in the directory.
- In line 17, the **`rootpw`** directive specifies the password for the DN specified by the **`rootdn`** directive. Naturally, a *very* strong/good password should be used here. The password can be specified in plain text (very, very bad idea), or the hash of the password can be specified. The **`slappasswd`** program can be used to generate password hashes.
- Finally, in line 22, the **`directory`** directive specifies the path to the BDB files containing the database and associated indices.

Database Back-End Type	Description
`bdb`	Berkeley DB instance definition. This is the recommended database back-end type. It uses the Sleepycat Berkeley DB to store data.
`sql`	Uses a SQL database back-end to store data.
`ldap`	Used as a proxy to forward incoming requests to another LDAP server.
`meta`	Metadirectory database back-end. It is an improvement on the LDAP-type back-end. Performs LDAP proxying with respect to a set of remote LDAP servers.
`monitor`	Stores information about the status of the **slapd** daemon.
`null`	Operations to this database type succeed, but do nothing. This is the equivalent of sending stuff to **/dev/null** in Linux/UNIX.
`passwd`	Uses the system's **/etc/passwd** file to serve user account information.
`perl`	Uses a Perl interpreter that is embedded directly into **slapd**.

Table 27-2 OpenLDAP Database Back-Ends

Having gone through a few important directives in the **slapd.conf** file, we will now make a few changes to the file to customize it for our environment. But before starting, we'll use the venerable **openssl** utility to generate and make use of a self-signed certificate to help secure all communications between the LDAP server and any LDAP clients (**sssd**, **nslcd**, **ldap** client utilities, and so on).

NOTE We've bundled working copies of the configuration and/or script files used in this chapter at the following URL to save you some typing: http://linuxserverexperts.com/8e/chapter-27.tar.xz

To follow along using this bundle, just download the file, extract its contents, and copy the file(s) to the proper directories on your local system.

1. As a privileged user, use the **openssl** utility to generate a self-contained self-signed certificate. The resulting file will be stored in /**etc**/**openldap**/**server.pem**.

```
[master@fedora-server ~]$ sudo openssl req -newkey rsa:1024 \
 -x509 -nodes -out /etc/openldap/server.pem \
-keyout /etc/openldap/server.pem \
-days 365  -subj "/CN=$(hostname)"
```

2. If it exists, make a backup of the current **slapd.conf** file by renaming it (so that you can always revert back to it in case of mistakes):

```
[master@fedora-server ~]$sudo mv /etc/openldap/slapd.conf{,.original}
```

3. Move the existing **slapd** configuration directory (/**etc**/**openldap**/**slapd.d**/) out of the way and create a new one:

```
[master@fedora-server ~]$ sudo mv /etc/openldap/slapd.d{,.original}
[master@fedora-server ~]$ sudo mkdir -p /etc/openldap/slapd.d
```

4. Use any text editor to create a new /**etc**/**openldap**/**slapd.conf** file with the following:

```
include      /etc/openldap/schema/core.schema
include      /etc/openldap/schema/cosine.schema
include      /etc/openldap/schema/inetorgperson.schema
include      /etc/openldap/schema/nis.schema
pidfile      /var/run/openldap/slapd.pid
argsfile     /var/run/openldap/slapd.args
#
TLSCACertificateFile  /etc/openldap/server.pem
TLSCertificateFile /etc/openldap/server.pem
TLSCertificateKeyFile /etc/openldap/server.pem
TLSCipherSuite  ALL:!NULL
#
database     bdb
suffix       "dc=example,dc=org"
rootdn       "cn=Manager,dc=example,dc=org"
#
# The hashed password below was generated using the command:
# "slappasswd -s test". Run the command and paste the output here.
rootpw {SSHA}gJeD9BJdcx5L+bfgMpmvsFJVqdG5CjdP
directory     /var/lib/ldap
```

5. Save your changes to the file and exit the editor.
6. Use the **slaptest** command to convert the **slapd.conf** file that you created earlier into the new **openldap** configuration format:

```
[master@fedora-server openldap]$ sudo slaptest \
 -f /etc/openldap/slapd.conf  -F /etc/openldap/slapd.d
```

7. The resulting **slapd** daemon's configuration should be owned by the system user named *ldap*. Use the **chown** and **chmod** commands to ensure that the configuration files have the correct ownership and permissions:

```
[master@fedora-server ~]$ sudo chown -R ldap:ldap /etc/openldap/
[master@fedora-server ~]$ sudo chown -R ldap:ldap /var/lib/ldap/
[master@fedora-server ~]$ sudo chmod -R a-rwx,u+rwX \
/etc/openldap/slapd.d
```

TIP Watch out for the permissions of the OpenLDAP configuration files! The **slapd** daemon will refuse to start on a Fedora or RHEL system if the LDAP user cannot read the configuration file(s). Also, the contents of the database directory (**/var/lib/ldap**) must be owned by the user *ldap* to avoid odd errors.

Starting and Stopping slapd

After setting up **slapd**'s configuration file, our next step will be to start the daemon. Starting it on **systemd**-enabled systems is easy. We'll use the **systemctl** command to first check the status of the daemon:

```
[master@server ~]$ systemctl status slapd
```

If the output of the previous command shows that the daemon is not currently running, start it with this command:

```
[master@server ~]$ sudo systemctl start slapd
```

And if you find that the LDAP service is already running, you can instead issue the **systemctl** command with the **restart** option, like so:

```
[master@server ~]$ sudo systemctl restart slapd
```

TIP On some Linux distros, you may get a warning message about **DB_CONFIG** not existing under the **/var/lib/ldap** directory whenever you start the LDAP service. You can fix this warning by using the sample **DB_CONFIG** file that ships with the distribution. The sample file is stored under **/etc/openldap/** or **/usr/share/openldap-servers/**, and here's the command to copy and rename the file:

```
# cp /usr/share/openldap-servers/DB_CONFIG.example \
/var/lib/ldap/DB_CONFIG
```

On **systemd**-enabled systems, if you want the **slapd** service to start up automatically with the next system reboot, type this:

```
[master@server ~]$ sudo systemctl enable slapd
```

Besides the basic service/daemon admin tasks (configuration, enabling, starting, stopping, and so on), another important task involved with getting your LDAP server (**slapd**) ready to serve clients on the network is to make sure the service port for the service is reachable through any relevant host-based or network-based firewalls.

Red Hat–based distros such as Fedora often ship with host-based firewall rules that protect the system and its services/daemons. You can use a predefined **firewalld** service definition on Fedora to permanently open up the port for the LDAP service by executing the following commands:

```
$ sudo firewall-cmd --add-service=ldap --add-service=ldaps  --permanent
$ sudo firewall-cmd --reload
```

Configuring OpenLDAP Clients

The notion of clients takes some getting used to in the LDAP world. Almost any system resource or process can be an LDAP client. And, fortunately or unfortunately, each group of clients may have its own specific configuration files. The configuration files for OpenLDAP clients are generally named **ldap.conf**, but they may be stored in different directories, depending on the particular client in question.

Depending on the distro, some common locations for the OpenLDAP client configuration files are under these directories: **/etc/openldap/**, **/etc/ldap**, and **/etc/**. The client applications that use the OpenLDAP libraries (provided by the **openldap*.rpm** package)—applications such as **`ldapadd`**, **`ldapsearch`**—consult the **/etc/openldap/ldap.conf** file, *if* it exists. The **nss_ldap** libraries instead use the **/etc/ldap.conf** file as their configuration file.

In this section, we will set up the configuration file for the OpenLDAP client tools. This configuration file is straightforward; we will be changing only a few of its directives.

Open the **/etc/openldap/ldap.conf** file in any text editor, and make sure that the following configuration directives are set to the appropriate values shown next. You should create the entries if they don't exist already in the file.

```
BASE dc=example,dc=org
URI     ldaps://localhost
TLS_CACERT    /etc/openldap/server.pem
TLS_REQCERT allow
```

TIP One particular variable/directive that you might need to change in the **/etc/openldap/ldap.conf** file, if you are using the client tools from a host other than the LDAP server itself, is the *`URI`* directive. This should be set to the IP address of the remote LDAP server. But because we are using the LDAP clients directly on the LDAP server itself, a **`URI`** directive of **`ldaps://localhost`** will suffice in our example.

Creating Directory Entries

The LDAP Data Interchange Format (LDIF) is used to represent entries in an LDAP directory in textual form. As stated earlier, data in LDAP is presented and exchanged in this format. The data in an LDIF file can be used to manipulate, add, remove, and change the information stored in the LDAP directory. Here's the format for an LDIF entry:

```
dn: <distinguished name>
<attribute_description>: <attribute_value>
<attribute_description>: <attribute_value>

dn: <yet another distinguished name>
<attribute_description>: <attribute_value>
<attribute_description>: <attribute_value>
...
```

The LDIF file is slightly strict in its format. You should keep these points in mind:

- Multiple entries within the same LDIF file are separated by blank lines.
- Entries that begin with the pound sign (#) are regarded as comments and are ignored.
- An entry that spans more than one line can be continued on the next line by starting the next line with a single space or tab character.
- The space following the colon (**:**) is important for each entry.

In this section, we will use a sample LDIF file to populate our new directory with basic information to set up our directory information tree, as well as with information describing two users, named *bogus* and *testuser*.

1. The sample LDIF file is presented next. Use any text editor to input the text in the listing into the file. Be careful with the white spaces and tabs in the file, and make sure you maintain a newline after each DN entry, as shown in our sample file.

```
dn: dc=example,dc=org
objectclass: dcObject
objectclass: organization
o: Example inc.
dc: example

dn: cn=bogus,dc=example,dc=org
objectclass: organizationalRole
cn: bogus

dn: cn=testuser,dc=example,dc=org
objectclass: organizationalRole
cn: testuser
```

2. Save the file as **sample.ldif** and then exit your text editor.

3. Use the **ldapadd** utility to import the **sample.ldif** file into the OpenLDAP directory:

```
[master@server ~]$ ldapadd -x -D "cn=manager,dc=example,dc=org" \
 -W -f sample.ldif
Enter LDAP Password:
adding new entry "dc=example,dc=org"
adding new entry "cn=bogus,dc=example,dc=org"
adding new entry "cn=testuser,dc=example,dc=org"
```

   These are the parameters used in this **ldapadd** command:

   - **-x** Indicates that simple authentication should be used instead of Simple Authentication and Security Layer (SASL).
   - **-D** Specifies the distinguished name with which to bind to the LDAP directory (that is, the **binddn** parameter specified in the **slapd.conf** file).
   - **-W** Allows the user to be prompted for the simple authentication password instead of specifying the password in plain text on the command line.
   - **-f** Specifies the file from which to read the LDIF file.

4. Enter the password you created using the **slappasswd** utility earlier—that is, the password that was specified in the **/etc/openldap/slapd.conf** file for the **rootpw** directive (line 12: **slappasswd -s test**). We used "test" as the password in our example.

We are done populating the directory!

Searching, Querying, and Modifying the Directory

We'll use a couple of OpenLDAP client utilities to retrieve information from our directory.

1. We'll use the **ldapsearch** utility to search for and retrieve every entry in the database directory:

```
[master@server ~]$ ldapsearch -x -b 'dc=example,dc=org' '(objectclass=*)'
...<OUTPUT TRUNCATED>...
# example.org
dn: dc=example,dc=org
objectClass: dcObject
....
# bogus, example.org
dn: cn=bogus,dc=example,dc=org
objectClass: organizationalRole
cn: bogus
...<OUTPUT TRUNCATED>...
# numResponses: 4
# numEntries: 3
```

2. Let's repeat the search again, but without specifying the **-b** option and also making the output less verbose:

```
[master@server ~]$ ldapsearch -x -LLL '(objectclass=*)'
dn: dc=example,dc=org
...<OUTPUT TRUNCATED>...
dn: cn=bogus,dc=example,dc=org
...<OUTPUT TRUNCATED>...
dn: cn=testuser,dc=example,dc=org
objectClass: organizationalRole
cn: testuser
```

   Here, we didn't need to specify explicitly the **basedn** to search because that information is already defined in our **/etc/openldap/ldap.conf** file.

3. We'll narrow down our query by searching only for the entry for the object whose common name (**cn**) is equal to *bogus*. Issue this command:

```
[master@server ~]$ ldapsearch -x -LLL -b 'dc=example,dc=org' \
 '(cn=bogus)'
dn: cn=bogus,dc=example,dc=org
objectClass: organizationalRole
cn: bogus
```

4. We'll attempt to perform a privileged operation on a directory entry using the **ldapdelete** utility. Let's delete the entry for the object with the DN of **cn=bogus,dc=example,dc=org**. Issue this command:

```
[master@server ~]$ ldapdelete -x -W -D \
'cn=Manager,dc=example,dc=org' 'cn=bogus,dc=example,dc=org'
Enter LDAP Password:
```

   Enter the password for the **cn=Manager,dc=example,dc=org** DN to complete the operation.

5. Use **ldapsearch** again to make sure the entry has indeed been removed:

```
[master@server ~]$ ldapsearch -x -LLL -b 'dc=example,dc=org' \
'(cn=bogus)'
```

This command should return nothing.

Using OpenLDAP for User Authentication

Now we'll set up the OpenLDAP server (and client) that we configured earlier in the chapter to manage Linux user accounts. We will be using some of the migration scripts that come with the software to pull/migrate the users that already exist in the system's **/etc/passwd** file into LDAP.

Configuring the Server

Setting up a Linux system to use LDAP as the storage back-end for user account information is easy once you have taken care of all the other basic OpenLDAP configuration tasks.

On mainstream Linux distros, a bunch of useful scripts exists in a package called **migrationtools*** that can help with the migration of information (such as users, groups, aliases, hosts, netgroups, and services) in existing name services into an OpenLDAP directory. These scripts are often stored under **/usr/share/migrationtools/**.

On our Fedora server, we'll customize the **migrate_common.ph** file to suit our particular setup, but first we need to install the package that provides the script:

1. Start by installing the package that provides the migration scripts we will be using:

   ```
   [master@fedora-server ~]$ sudo dnf -y install migrationtools
   ```

   Among other things, this command will install the file under **/usr/share/migrationtools/migrate_common.ph**.

2. Open the file for editing and look for lines/entries similar to these:

   ```
   $DEFAULT_MAIL_DOMAIN = "padl.com";
   $DEFAULT_BASE = "dc=padl,dc=com";
   ```

   We will change these variables to read like so:

   ```
   $DEFAULT_MAIL_DOMAIN = "example.org";
   $DEFAULT_BASE = "dc=example,dc=org";
   ```

3. We'll use another one of the migration scripts (**migrate_base.pl**) to create the base structure for our directory:

   ```
   [master@fedora-server ~]$ cd /usr/share/migrationtools/
   ```

4. Now execute the script:

   ```
   [master@fedora-server migrationtools]$ ./migrate_base.pl > ~/base.ldif
   ```

   This command will create a file named **base.ldif** under your home directory.

5. Make sure **slapd** is running (via **`systemctl start slapd`**) and then import the entries in the **base.ldif** file into the OpenLDAP directory:

   ```
   [master@fedora-server migrationtools]$ ldapadd -c -x -D \
   "cn=manager,dc=example,dc=org" -W -f ~/base.ldif
   ```

6. Now we need to export the current users in the system's **/etc/passwd** file into an LDIF-type file. We will use the **/usr/share/migrationtools/migrate_passwd.pl** script:

   ```
   [master@fedora-server migrationtools]$ ./migrate_passwd.pl \
    /etc/passwd >  ~/ldap-users.ldif
   ```

7. We can begin importing all the user entries in the **ldap-users.ldif** file into our OpenLDAP database. For this, we will use the **`ldapadd`** command:

   ```
   [master@fedora-server migrationtools]$ ldapadd -x -D  \
   "cn=manager,dc=example,dc=org" -W -f ~/ldap-users.ldif
   ```

Enter the **rootdn**'s password (*test*) when prompted.

Configuring the Client

Configuring a client system to use an LDAP directory for user authentication is as easy as pie on a Fedora, RHEL, or CentOS system. Most distros have command-line utilities as well as distro-specific GUI tools that make it easy. We'll use the **authselect** utility on our Fedora system here.

1. Make sure the **authselect** package is installed by running the following:

```
[master@client ~]$  sudo dnf -y install authselect
```

2. Display the current and active system identity and authentication source:

```
[master@client ~]$ authselect current
Profile ID: sssd
...<OUTPUT TRUNCATED>…
```

   The output shows that the System Security Services Daemon (**sssd**) profile is currently being used. Next, we'll configure a new service domain called *newldap* to use with **sssd**.

3. As a privileged user, create (or overwrite) the **/etc/sssd/sssd.conf** (chmode 400) configuration file with the following content. We'll use Bash's Heredoc to do this.

```
[master@client ~]$sudo tee /etc/sssd/sssd.conf > /dev/null <<'EOF'
[domain/newldap]
id_provider = ldap
autofs_provider = ldap
auth_provider = ldap
chpass_provider = ldap
ldap_uri = ldaps://localhost/
ldap_search_base = dc=example,dc=org
ldap_tls_reqcert = allow
cache_credentials = True

[sssd]
services = nss, pam, ssh, autofs
domains = newldap
EOF
```

4. Set the proper permissions on **sssd.conf** and then ensure that the system is using the **sssd** profile:

```
[master@client sssd]$ sudo chmod 600 /etc/sssd/sssd.conf
[master@client sssd]$ sudo authselect  select sssd --force
```

5. Apply the changes to **sssd**:

```
[master@client sssd]$ sudo authselect apply-changes
```

6. Restart the **sssd** service unit:

```
[master@client sssd]$ sudo systemctl restart sssd
```

7. You use the syntax "<username@newldap>" to quickly test that the system can use the supplied LDAP server to provide authentication/authorization services to the client. For example, use the **id** command to query the LDAP server for info about the user yyang stored in LDAP:

```
[master@client ~]$  id  yyang@newldap
uid=1001(yyang) gid=1001(yyang) groups=1001(yyang)
```

That's it. You are done.

We just described a simple way to enable a Fedora client system to use an OpenLDAP server for its user authentication. We did not bother with a lot of details, because this is just a proof of concept. Following are some of the details you might have to deal with in a real production environment:

- **Home directories** Ensure that users' home directories are available to them when they're logging in from any system. One way to do this is by sharing user home directories via Network File System (NFS) and exporting the share to all client systems.
- **Security** Our sample setup did not use any super-secure measures; for example, we used and blindly trusted our self-signed TLS/SSL certificates! This should be of paramount importance in a production environment so that user passwords do not go flying across the network in plain text.
- **Miscellaneous** Other issues weren't addressed here, but we'll leave those as a mental exercise for you to stumble upon and ponder.

TIP Take a look at the FreeIPA project (www.freeipa.org) for a canned identity management solution that combines and extends various concepts and solutions discussed in this and other chapters.

Summary

This chapter covered some LDAP basics. It concentrated mostly on the open source implementation of LDAP known as OpenLDAP. You learned about the components of OpenLDAP—the server-side daemons and the client-side utilities used for querying and modifying the information stored in an LDAP directory. We created a simple directory and populated it with some sample entries.

Finally, we configured our LDAP server to store user and password information for users in its directory. This makes it possible for remote LDAP clients to consult the LDAP server for their own user authentication needs.

We've barely scratched the surface of the topic, but LDAP is too large a topic to be covered fully in a single chapter. Hopefully, we've whetted your appetite and got you started in the right direction with some essential concepts and ideas.

CHAPTER

28 Printing

Some users print for fun, some for work, and others for profit—whatever their reason or the season, you as the system administrator will have to support them. You'll be glad to know that this chapter has you covered.

Over time, as Linux platforms have become more widely adopted as viable desktop/workstation platforms, most printer manufacturers now provide reasonable support for printing on Linux-based systems.

The Common UNIX Printing System (CUPS) plays a very important and central role in the improved state of printing support in the Linux world that we enjoy today. With the advent of CUPS, Linux printing is much easier to configure, use, and manage. This chapter covers the installation of the CUPS system, along with the administrative tasks involved in maintaining your printing environment.

Printing Terminologies

Several printing systems are available in the Linux world today, and all are more or less based on the venerable Berkeley Software Distribution (BSD) printing system. Regardless of the printing system under discussion, certain terms and concepts apply across the board. Following are some terms that are used in printing-related discussions:

- **Printer** A peripheral device usually attached to a host computer or the network.
- **Job** The file or set of files that is submitted for printing.

- **Spooler** The software that manages print jobs. It is responsible for receiving print jobs, storing jobs, queuing jobs, and sending jobs to the physical hardware that will do the actual printing. Spoolers run as daemon processes that are always waiting to service print requests, and for this reason they are often referred to as "print servers." These are examples of spoolers:
  - **LPD** The original BSD Line Printer Daemon is the oldest printing system.
  - **LPRng** This enhanced, extended, and portable implementation of the Berkeley LPR spooler functionality merges the best features of the System V printing system with that of the Berkeley system.
  - **CUPS** Provides a portable printing layer for UNIX-based systems and uses the Internet Printing Protocol (IPP) as the basis for managing print jobs and queues.
- **PDL** The page description language is the language in which printers accept input. PostScript and PCL are examples of PDLs.
- **PostScript** PostScript files are programs, and PostScript is a stack-based programming language. Most UNIX/Linux programs generate output in PostScript format for printing. PostScript-based printers directly support this format.
- **Ghostscript** A software-based PostScript interpreter for non-PostScript printers, Ghostscript is used for software-driven printing. It will generate the language of a printer from PostScript. Examples are Aladdin Ghostscript (commercial version), GNU Ghostscript (free version), and ESP Ghostscript (CUPS).
- **Filter** This special program or script processes data (jobs) before it is sent to the printer. A spooler sends the job to the filter, and then the filter passes it on to the printer. File format translation and accounting usually occur at the filtering layer.

The CUPS System

CUPS has gained widespread adoption in the Linux community. Even the latest versions of Apple's macOS operating systems use CUPS. This is all because CUPS provides a ubiquitous printing environment no matter what operating system you are using. Along with the standard UNIX printing protocol of LPR, CUPS supports Samba printing and the Internet Printing Protocol. Using the concept of print *classes,* the CUPS system can print a document to a group of printers for use in high-volume printing environments. It can act as a central print spooler or simply supply the printing method for your local printer.

Running CUPS

CUPS was originally developed by Easy Software Products and is now maintained by Apple, Inc., as an open source project. The project is hosted at www.cups.org. There are two methods of installation: through your Linux distribution's package management system and by compiling from source. The first method is highly recommended, because the distributions typically have all of the popular printer support built into CUPS. When compiling by hand, you have to get drivers for your printers yourself.

Installing CUPS

Mainstream Linux distributions have pre-packaged versions of CUPS readily available as an RPM package or .deb package; in fact, CUPS is the default printing system used on those distributions. Therefore, sticking with your distribution's package version of CUPS is the recommended method for installation, particularly because your chosen distro vendor will have put in all the work to ensure that CUPS works well and is properly integrated into their distro.

Because most systems may already have some of the CUPS components (libraries, server, client, and so on) installed during the initial operating system installation, there might be very little else you need to do in the way of actual installation! Nevertheless, you should double-check to confirm which CUPS components are installed. On an RPM-based distro, type this:

```
[master@fedora-server   ~]$ rpm  -q cups
```

If this query does not include the stand-alone CUPS package (**cups*.rpm**) that provides the CUPS server functionality, you can quickly install CUPS on Fedora and other Red Hat–like distros (such as RHEL and CentOS) by typing the following:

```
[master@fedora-server   ~]$ sudo dnf -y install cups
```

For Debian-like Linux distros, such as Ubuntu, you can use **dpkg** to check whether the software is already installed:

```
master@ubuntu-server:~$ dpkg -l cups
```

If the software is not already installed on the Ubuntu server, you can install it using the Advanced Packaging Tool (APT):

```
master@ubuntu-server:~$ sudo apt-get install cups
```

On an openSUSE system, you should be able to use zypper to get CUPS installed:

```
[master@opensuse-server   ~]$ zypper  -i  cups
```

Once you have installed the CUPS software, you need to start the CUPS daemon. On Linux distros that rely on **systemd** as the service manager, you can use the **systemctl** command to start CUPS and enable it for auto-startup, like so:

```
[master@fedora-server ~]$ sudo systemctl --now  enable cups
```

To start the CUPS service on openSUSE, you would use the **rccups** command:

```
opensuse-server:~ # rccups start
```

Once you've started the CUPS printing system using the appropriate method on your system, you are ready to configure CUPS and provision printers for actual use.

Configuring CUPS

The main configuration file for the CUPS print daemon is called **cupsd.conf**. It is usually located in the **/etc/cups/** directory. It is a plain-text file with directives (syntax) similar to that of the Apache web server. The variables used in the file determine how the server operates.

The file is well commented, and all that usually needs to be done is to uncomment certain lines in the file to turn certain functions on or off.

Here are some interesting directives used in the **cupsd.conf** file:

- **`Browsing`** Controls whether network printer browsing is enabled. The options are **`Yes`** and **`No`**.
- **`BrowseLocalProtocols`** Specifies the protocols to use when collecting and distributing shared printer information on the local network. The options are **`all`**, **`none`**, and **`dnssd`**.
- **`ServerName`** Specifies the hostname that is reported to clients. Typically, it's the same name as the fully qualified hostname of the server.
- **ServerAlias** Used for HTTP host header validation. The host header provides an additional mechanism for clients to uniquely identify a web server or web application.
- **`Listen`** Defines the address and port combination on which the CUPS daemon should listen.
- **`Allow`** Allows access from given addresses, hostnames, domains, or interfaces. This directive is valid only within the **`Location`** and **`Limit`** sections.
- **`Limit`** Specifies an operation or method to be limited in the **`Policy`** section of the configuration file.
- **`Location`** Specifies the access control rules for the given HTTP resource or file system path.

NOTE The CUPS software is Internet Protocol version 6 (IPv6) ready. To make the CUPS daemon listen on both IPv4 and IPv6 sockets, you could set the listen directive to something like **`Listen *:631`**. And if you need to configure a specific IPv6 address (such as 2001:db8::1) explicitly for the CUPS server to listen on, you need to enclose the IPv6 address in square brackets (for example, **`Listen [2001:db8::1]:631`**).

A particularly interesting location is the root location, represented by the slash symbol (/). This location in the default **cupsd.conf** file looks like this (note that line numbers have been added to the listing to aid the discussion):

```
1. <Location />
2.          Order allow, deny
3.          Deny all
4.          Allow localhost
5. </Location>
```

- **Line 1** This is the start of the `Location` directive; here, it defines the start of /, which is the path for *all* get operations—that is, the topmost level of the web server.
- **Line 2** This is the `Order` directive. It defines the default access control for the location in question. These are the possible values for the `Order` directive:
  - `deny, allow` Allow requests from all hosts by default; then check the `Deny` directive(s), followed by the `Allow` directive(s).
  - `allow, deny` Deny requests from all hosts by default; then check the `Allow` directive(s), followed by the `Deny` directive(s).
- **Line 3** This `Deny` directive specifies the host(s) to be denied access. In this case, the `all` keyword means all hosts. Note that the preceding `Order` directive translates to "`Deny` requests by default but check/honor the `Allow` directive followed by any `Deny` directives."
- **Line 4** The `Allow` directive specifies the host(s) to be allowed access. In this case, the only host allowed is the localhost—that is, the loopback address (127.0.0.1).
- **Line 5** This is the closing tag for the `Location` directive.

TIP To change the default behavior of CUPS from allowing access only from the localhost to the **/admin** location in **cupsd.conf**, you could, for example, change the `Allow` directive from `Allow 127.0.0.1` to `Allow All` and then comment out the `Deny All` directive or change to `Deny None`.

Adding Printers

After you have finished installing and starting the CUPS service, you need to log into the web interface, which is available through port 631. In your web browser, type **https://localhost:631**. By default, you must be logged into the same server that you are trying to administer. An interesting point to note is that 631 is the same port that CUPS uses for accepting print jobs! When you connect to the web page, you will see a page similar to Figure 28-1.

Figure 28-1 CUPS administration web page

CUPS Remote Web Administration Considerations (Firewall Configuration Directives)

If you want to administer CUPS via the web interface from locations other than the local server that you are working on, you need to do a few things.

First, you need to modify the **cupsd.conf** file to allow other hosts to connect. In particular, you need to set the proper access controls to the **/admin** location in the file. Second, you need to permit access through any existing firewall to the CUPS service (port 631). Third, you might need enable the **remote_admin** directive and tweak the **ServerAlia**s directives in cupsd.conf.

To open up the proper ports on a system running **firewalld**, run the following:

```
$ sudo firewall-cmd --add-service=ipp    --permanent
$ sudo firewall-cmd --reload
```

To tweak the **remote_admin** directive, using the **cupsctl** command, type this:

```
$ sudo cupsctl  --remote-admin
```

To temporarily get around any issues related to the **cupsd** web server host headers or hostname, use **cupsctl** to make the **ServerAlias** directive less strict:

```
$ sudo cupsctl ServerAlias="*"
```

Local Printers and Remote Printers

Adding printers is easy in CUPS. At a minimum, you should know how the printer will be *connected* or *attached* to your system. Printers can be connected/attached to using two broad methods: locally attached and/or attached via the network. Several modes or possibilities exist under each method. The modes with which CUPS addresses the printer resources are specified by using the device *Uniform Resource Information (URI)* in CUPS. These are examples of possible device URIs that can be configured in CUPS:

- **Directly connected (local)** An example of a locally attached printer is when a stand-alone system is connected to a (modern) printer directly through a Universal Serial Bus (USB) cable. In CUPS lingo, some possible device URIs for a locally attached printer are specified as follows:
  - **serial:/dev/ttyS*** For a printer attached to the serial port
  - **usb:/dev/usb/lp*** For a printer attached to the USB port
- **IPP (network)** The Internet Printing Protocol allows a printer to be accessed over the network. Most modern operating systems and printers support this protocol. An example of an IPP device URI in CUPS is ipp://*hostname*/print_queue_name/.

- **LPD (network)** CUPS supports printers that are attached to other systems running the Line Printer Daemon (LPD). Most UNIX/Linux systems (even some Windows servers) support this daemon. So if a printer is attached to a host that supports LPD, CUPS can be used to make that printer available on the network to other hosts that do not necessarily support LDP. Virtually all HP laser printers with network connectivity natively support LPD. A sample device URI to address an LPD printer is lpd://*hostname*/queue, where *hostname* is the name of the machine where LPD is running.
- **SMB (network)** The Service Message Block is the foundation of file and printer sharing on Windows networks. Linux hosts also support SMB through the use of the Samba software. For example, if a Windows system (or a Samba server) has a printer shared on it, CUPS can be configured to access and make that printer available to its own clients. A sample device URI to address an SMB printer resource is smb://*servername*/*sharename,* where *servername* and *sharename* are respectively the DNS/ hostname and the name by which the printer has been shared on the Windows box or on the Samba server.
- **Networked printer** This refers to a class of printers that have built-in networking capabilities. These printers don't need to be connected to any stand-alone system. They usually have some form of network interface of their own—whether Ethernet, wireless, or some other method of connecting directly to a network. A popular type of this printer group is the HP JetDirect series. A sample URI to address such printers is socket://*ip_address*:*port,* where *ip_address* is the IP address of the printer, and *port* is the port number on which the printer listens for print requests. This is usually port 9100 on the HP JetDirect series.

Using the Web Interface to Add a Printer

Printers can be added and configured in CUPS in several ways: through a web interface using a browser of some sort, through the command line, and by using a purpose-built and distribution-specific GUI tool such as **`yast2 printer`** in openSUSE or **`system-config-printer`** in Fedora. The web interface method is probably the easiest, because it uses a wizard to walk you through the entire process. The command-line method is probably the most universal, because the syntax is similar regardless of the Linux distribution. And the GUI method is like a coin toss—you get what you get!

This section will walk you through setting up a printer through the CUPS web interface. We will set up an imaginary printer with the following properties:

```
Name:              Imagine-printer
Description:       You only need to imagine to print here.
Location:          Building 3
Connection Type:   Internet Printing Protocol (IPP)()
Make:              HP
Model:             HP Color Laserjet Series PCL
```

Let's begin the process.

NOTE To keep things simple, if you are prompted for a username and password at any point throughout the process, type **root** as the user and enter root's password.

1. While logged into the system running CUPS, launch a web browser and connect to CUPS at https://localhost:631.
2. In the top navigation area of the web page, click the Administration menu; then click the Add Printer button on the ensuing screen. You might be prompted with an authentication request. Enter **root** as the UserName and input root's password. Click OK to continue.
3. You'll see a screen listing the various available connection methods. Any automatically discovered networked printers will also be listed. Select the option that best describes the printer's connection to the system. We specify IPP here, for our imaginary printer. The full URI is **ipp://localhost/631**. Click Continue when done.
4. The next screen will vary (or be completely skipped) depending on the connection method selected in the previous step. You can usually accept the default setting or manually enter **ipp://localhost/631** in the connection field. Click Continue.
5. On the Add Printer page, enter the information for the printer that was provided earlier—the name, description, location, and so on. Click Continue.
6. The next screen will list various printer makes (or manufacturers). Select the make of the printer you are trying to install. We select HP on sample system. Click the Add Printer button to continue.
7. The ensuing screen will list various models for the manufacturer you selected. Select the model closest to that of the printer you are trying to install. We select HP Color Laserjet Series PCL* on our sample system. Click the Add Printer button to continue.

TIP The printer makes and manufacturers shown in the list obviously do not cover all the printer makes that exist. If you need more variety/coverage, on a Fedora or RHEL-type system, you can install the **gutenprint-cups** RPM package. That package provides additional drivers for various printer manufacturers beyond the ones that ship with the basic open source version of the CUPS software. The **printer-driver-gutenprint.deb** package will provide extra drivers on Debian-like systems, such as Ubuntu.

8. The final screen will allow you to set the default options for the new printer. Some sample options you might want to change are Media Size, Media Source, Output Mode, Banners, and so on. Accept the default options for now.

   Click the Set Default Options button.
9. You'll see a page confirming that the printer has been successfully added.
10. You'll see a page showing the properties for the printer that was just added.

NOTE For automatically detected printers (such as USB or network printers) for which you did not have to manually specify the device URI or connection string, you will notice that the software automatically generates a device URI with which to address the detected printer.

Now imagine that you have clicked the Print Test Page option under the Maintenance drop-down list of the properties page for the printer that was just added. And imagine the test page being printed successfully!

Using Command-Line Tools to Add a Printer

You can also use the command line to add a printer to CUPS. Once you are comfortable with how CUPS works, you may find managing the CUPS system through its command-line interface to be a little faster. To add a printer from the command line, you need some pertinent information, such as the printer name, the driver, and the URI.

This section will detail a simple example for setting up a printer through the use of CUPS command-line tools. As with the preceding example, we will set up an imaginary printer with most of the same properties used in that example; but here we'll change the name of the printer (also called the *printer queue*) to Imagine-printer-number-2. We will also use a different device URI to address the printer. This time, we will assume that the printer has built-in networking capabilities and that it has an IP address of 192.168.1.200 listening on port 9100—that is, the device URI will be socket://192.168.1.200:9100.

1. While logged into the system as the superuser, launch any virtual terminal and list the printer queues you currently have configured for your system. Use the **lpstat** utility:

```
[master@server ~]$ lpstat -a
Imagine-printer accepting requests since Fri 23....PST
```

2. Use the **lpinfo** command to get a list of printer models and drivers supported by the CUPS server. For our sample scenario, we are interested in the Fuji Xerox DocuPrint series of printers, so we'll pipe the long output of the command to the **grep** command to narrow down the list:

```
[master@server ~]$ lpinfo  -m  | grep -i fuji
lsb/usr/cupsfilters/Fuji_Xerox-DocuPrint_CM305_df-PDF.ppd Fuji Xerox
```

   We'll choose the first line of the output as the model and driver for our sample printer, because it is the closest generic match.

   The CUPS **lpadmin** command is smart enough to allow you to specify only an abbreviated form of the printer model/driver, instead of the full printer model and driver—as we will do in the next step.

3. Now issue the **lpadmin** command to add the printer. Type the following:

```
[master@server ~]$ lpadmin -p "Imagine-printer-number-2" -E  \
-v socket://192.168.1.200 \
-m  "lsb/usr/cupsfilters/Fuji_Xerox-DocuPrint_CM305_df-PDF.ppd" \
-D  "You only need to imagine to print here" \
-L  "Building 3"
```

4. Use the **lpstat** command again to list all the printers that are present:

```
[master@fedora-server ~]$ lpstat -a
Imagine-printer accepting requests since Fri 23 ... 8 PM EDT
Imagine-printer-number-2 accepting requests since Fri 23 … 6 PM EDT
```

5. You can also view the printer you just added on the CUPS web interface. Point your web browser to this URL:

```
https://localhost:631/printers
```

Routine CUPS Administration

Setting up the printer(s) is half of the battle in managing a printing environment. The preceding section gave you enough information to get you going in that regard. This section will discuss some routine printer administration tasks, such as deleting printers, managing the printer queue, and viewing print job statuses. We will use both the command-line tools and the web interface to manage some of these tasks.

Setting the Default Printer

On a system with multiple print queues set up, you may find it desirable to set up a particular printer (queue) as the default printer for clients to use. The default printer is the printer that is used whenever a printer name is not specified explicitly for printing by clients.

For example, to set up the printer named Imagine-printer-number-2 as the default printer on the system, you'd type the following:

```
[master@server ~]$ lpadmin -d imagine-printer-number-2
```

Enabling, Disabling, and Deleting Printers

Disabling a printer is akin to taking the printer temporarily offline. In this state, the printer queue can still accept print jobs, but it will not actually print them. The print jobs are queued up until the printer is put in an enabled state or restarted. This is useful, for example, when the physical print device is not working properly and the system administrator does not want to interrupt users' printing.

To disable a printer named imagine-printer-number-2, type the following:

```
[master@server ~]$ cupsdisable imagine-printer-number-2
```

To enable the printer named imagine-printer-number-2, type this:

```
[master@server ~]$ cupsenable imagine-printer-number-2
```

To delete a printer named bad-printer, from the command line, type this:

```
[master@server ~]$ lpadmin -x bad-printer
```

Accepting and Rejecting Print Jobs

Any printer managed by CUPS can be made to accept or reject print jobs. This is a departure from the disabled state of a printer in the sense that a printer that is made to reject print jobs will simply *not accept* any print requests. Making a printer reject print jobs is useful when, for example, a printer needs to be put out of service for a long period of time but not

deleted completely. Whenever a printer is made to reject print jobs, it will complete any print jobs in its current queue and will immediately stop accepting any new requests.

As an example, to make a printer named imagine-printer-number-2 reject print jobs, type the following:

```
[master@server ~]$ reject  imagine-printer-number-2
```

Use the **lpstat** command to view the state of this printer:

```
[master@server ~]$ lpstat -a imagine-printer-number-2
```

To make the printer named imagine-printer-number-2 resume accepting print jobs, type this:

```
[master@server ~]$ accept  imagine-printer-number-2
```

View the printer's status again to ensure it is now accepting requests:

```
[master@server ~]$ lpstat  -a  imagine-printer-number-2
```

Managing Printing Privileges

In its out-of-the-box state, any printer managed by CUPS can be sent print jobs by users. In large multiuser environments, it may be necessary to control which users or groups have access to which printer(s) for security reasons, accountability, and so on. CUPS offers a simple way to do this through the use of the **lpadmin** utility.

For example, to allow *only* the users named *yyang* and *mmellow* to print to the printer named *imagine-printer,* you'd type the following:

```
[master@server ~]$ lpadmin -p imagine-printer -u allow:yyang,mmellow
```

To perform the opposite of this command and deny the users yyang and mmellow access to the printer, type this:

```
[master@server ~]$ lpadmin -p imagine-printer -u deny:yyang,mmellow
```

To remove all the preceding restrictions and allow all users to print to the printer named imagine-printer, type this:

```
[master@server ~]$ lpadmin -p imagine-printer -u allow:all
```

Managing Printers via the Web Interface

Most of the preceding tasks can also be performed from the CUPS web interface. Using buttons and links, you can easily delete printers, control print jobs, modify the properties of a printer, stop printers, reject print jobs, and so on.

For example, as an administrator, you may need to check the print queues periodically to make sure that everything is going smoothly. Clicking the Jobs tab (or going directly to http://localhost:631/jobs) on the web interface will reveal options available for viewing and manipulating jobs in the queue.

Using Client-Side Printing Tools

When a client machine prints, the job is sent to the print server and is spooled. Spooling is simply the act of putting a print job into the print queue. A print job can be in different states, such as in progress, paused (for example, the administrator has paused printing to refill the paper tray), or cancelled (where the job has been manually cancelled for whatever reason). When something goes awry with the printer, print jobs can queue up and create a problem when the printer comes back online.

In this section, we'll look at some commands that can be used to print and manage print queues. We cover the end user's (client) view and interaction with the printing system.

lpr

The **lpr** command is used to print documents. Most PostScript and text documents can be printed by directly using the **lpr** command. If you are using a stand-alone GUI application (such as LibreOffice, GIMP, and so on), you might have to set up those applications to print to the correct device.

Let's create a plain-text file that we'll attempt to print. The file will contain the simple text "Hello Printer" and be named **test-page.txt**.

1. While logged into the system as an unprivileged user (for example, the user yyang), type the following:

   ```
   [yyang@client ~]$ echo "Hello Printer" >> test-page.txt
   ```

2. Find out the name of the default printer configured on the system:

   ```
   [yyang@client ~]$ lpstat -d
   system default destination: Imagine-printer
   ```

3. Send the **test-page.txt** file to the default printer:

   ```
   [yyang@client ~]$ lpr  test-page.txt
   ```

   This will print/send the document **test-page.txt** to the default printer, which is usually the first printer that was installed.

4. Now send the same document to the other imaginary printer that was installed earlier, the printer named Imagine-printer-number-2:

   ```
   [yyang@client ~]$ lpr -P Imagine-printer-number-2 test-page.txt
   ```

   Once you have entered this command, the printer should start printing fairly quickly, unless you are printing a large file.

To see the status of your print job, use the **lpq** command (discussed next).

lpq

After you have submitted the job, you can view what is on the print spooler by using the **lpq** command. If you've just printed a job and notice that it doesn't come out of the printer, use the **lpq** command to display the current list of jobs that are spooled on the printer. Typically, you'll see a bunch of jobs in the queue, and upon further investigation, you may discover that the printer is out of paper. If you need to unspool the print job from the printer, you can use the **lprm** command, discussed in the next section.

For example, to see the status of the print request that was sent to the default printer, type the following:

```
[yyang@client ~]$ lpq -al
```

To view the status of the print job sent to the second printer, type this:

```
[yyang@client ~]$ lpq -al  -P Imagine-printer-number-2
```

It's either our fault or yours (because you didn't use your *imagination* well enough!) if both print jobs are stuck in the imaginary print queue or if you don't see any physical pages being printed. Regardless of whose fault it is, we'll remove the print jobs next.

lprm

When you suddenly realize that you didn't mean to print the document you just sent to the printer, you might have a chance to delete it before it gets printed. To do this, use the **lprm** command. This will unspool the print job from the printer.

For example, to delete the print job that you created with an ID of 2 from the default printer, type the following:

```
[yyang@client ~]$ lprm 2
```

To remove a job from a specific printer, simply add the **-P** option. For example, to remove the job with an ID of 2 from the printer named Imagine-printer-number-2, type this:

```
[yyang@client ~]$ lprm 2 -P imagine-printer-number-2
```

If you are the root user or a privileged user, you can purge all print jobs from the printer named **imagine-printer** by issuing the **lprm** command:

```
[root@server ~]# lprm -P imagine-printer -
```

The dash (**-**) at the end of this command means "all jobs."

TIP Regular users can typically manage only their own print jobs—that is, *user A* cannot ordinarily go and delete a job submitted by *user B* from the print queue. The superuser can, of course, control everybody's print jobs. Also, you should understand that the window between sending a job to the printer and being able to delete the job is very narrow. Therefore, you may find that the **lprm** request fails because the command was issued too late. This will usually result in an error such as "lprm: Unable to lprm job(s)!"

Summary

This chapter discussed the Common UNIX Printing System (CUPS) and touched on simple printer management tasks, such as adding printers via the CUPS web interface and from the command line, managing printers, and managing print jobs under Linux. Usage of some common client tools to manage print jobs in Linux was discussed, with examples provided. You also learned about some of the configuration directives that are used in CUPS' main configuration file, **cupsd.conf**.

However, we only scratched the surface of the capabilities and features of CUPS. Fortunately, the software comes with extensive built-in documentation, which is a highly recommended read if you plan on using CUPS extensively to deploy and manage printers in your environment. The same documentation is also available online at the CUPS home page, www.cups.org.

Once you have printing set up on your Linux server, you'll find that it does its job quite nicely and lets you focus on other new and interesting challenges. Problems that arise with printing services typically stem from issues with the physical printer itself (such as paper jams and running out of paper), user abuse, office politics, and so on.

CHAPTER

29 Dynamic Host Configuration Protocol (DHCP)

Manually configuring IP addresses for a handful of systems is a fairly simple task. However, manually configuring IP addresses for an entire department, building, or enterprise of heterogeneous systems can be daunting and prone to errors! The Dynamic Host Configuration Protocol (DHCP) can assist with these tasks. A DHCP client machine can be configured to obtain its IP address automatically. When the DHCP client software is started, it broadcasts a request onto the network for an IP address. If all goes well, a DHCP server on the network will respond, issuing an address and other necessary information to complete the client's network configuration.

DHCP is also useful for configuring mobile or transient machines. Users who travel from place to place can connect their machines to the local (wired or wireless) network and obtain an appropriate address for their location.

This chapter covers the process of configuring a DHCP server and client. It includes obtaining and installing the necessary software and then walks through the process of creating a configuration file for it. At the end of the chapter, we'll step through a complete sample configuration.

NOTE DHCP is a standard. Thus, any operating system or device that conforms to or implements to this standard should, theoretically, be able to interoperate with a Linux-based DHCP server or client. So, for example, Microsoft Windows client systems can be configured to use DHCP and can contact a Linux-based DHCP server to get their IP addresses. The Windows clients will not necessarily know or care that their IP configuration information is being provided by a Linux server!

The Mechanics of DHCP

When a client is configured to obtain its address from the network, it asks for an address in the form of a DHCP request. A DHCP server listens for client requests. Once a request is received, it checks its local database and issues an appropriate response. The response can include any combination of an IP address, name servers, network mask, default gateway, and so on. The client accepts the response from the server and configures its local settings accordingly.

The DHCP server maintains a list of addresses it can issue. Each address is issued with an associated *lease*, which dictates how long a client is allowed to use the address before it must contact the server to renew the lease. When the lease expires, the client is not expected to use the address any more. As such, the DHCP server assumes that the address has become available and can be put it back in the server's pool of addresses.

The implementation of the Linux DHCP server includes several key features common to many other DHCP server implementations. The server can be configured to issue any free address from a pool of addresses or to issue a specific address to a specific machine. In addition to serving DHCP requests, the Linux DHCP server can service Bootstrap Protocol (BOOTP) requests.

The DHCP Server

Dynamic Host Configuration Protocol Daemon (**dhcpd**), the DHCP server, is responsible for serving IP addresses and other relevant information in response to client requests. Since DHCP is a broadcast-based protocol, a server will usually have to be present on each subnet for which DHCP service is to be provided.

Installing DHCP Software via RPM

The Internet Systems Consortium (ISC) DHCP server is the de facto DHCP implementation for Linux distributions. It is also available in most Linux distributions' software repositories in a prepackaged/binary format.

In this section, we run through the process of installing the ISC DHCP software on Red Hat–like Linux distros. On Linux systems running Fedora, Red Hat Enterprise Linux (RHEL), or CentOS, the ISC DHCP software is separated into two different packages:

- **dhcp-client*.rpm** The **dhcp-client** package provides the ISC DHCP client daemon.
- **dhcp-server*.rpm** The **dhcp-server** package provides the ISC DHCP server service.

On most Linux distributions, you will most likely already have the DHCP client-side software automatically installed during the original operating system installation. We can check to see what is already installed on our sample Fedora system by typing the following:

```
[master@fedora-server ~]$ rpm -qa | grep dhcp-client
dhcp-client-*
```

From the sample output, you can see that the client-side package is already installed.

To set up the DHCP server on a Fedora-based distro, we need to install the necessary package. We will use **dnf** to download and install the software automatically:

```
[master@fedora-server ~]$ sudo dnf -y install dhcp-server
```

Once this command completes successfully, we should have the necessary software installed.

Installing DHCP Software via APT in Ubuntu

On our Ubuntu server, we'll use **dpkg** to query the local software database for the DHCP client software:

```
master@ubuntu-server:~$ dpkg -l | grep dhcp
ii  isc-dhcp-client   *  ISC DHCP client
ii  isc-dhcp-common   *  common files used by all the isc-dhcp* packages
```

From the sample output, we can see that the DHCP client package is already installed.

To install the DHCP server software on an Ubuntu distro, type the following:

```
master@ubuntu-server:~$ sudo apt-get install isc-dhcp-server
```

Once this command completes, you should have the necessary DHCP server software installed. The DHCP server service will also be automatically started for you after installation.

Configuring the DHCP Server

The primary configuration file of the ISC DHCP server is **dhcpd.conf**, and most distros store the file under the **/etc/dhcp/** directory. Here are the main highlights of the configuration file:

- A set of declarations to describe the networks, hosts, or groups attached to the system and possibly the range of addresses that can be issued to each respective entity. Multiple declarations can be used to describe multiple groups of clients. Declarations can also be nested in one another when multiple concepts are needed to describe a set of clients or hosts.
- A set of parameters that describes the overall behavior of the server. Parameters can be global or local to a set of declarations.

NOTE Because every site may have a unique network with unique addresses, every site must be set up with its own configuration file. If this is the first time you are dealing with DHCP, you might want to start with the sample configuration file presented toward the end of this chapter and modify it to match your network's characteristics.

Like most configuration files in Linux, the file is ASCII text and can be modified using your favorite text editor. The general structure of the configuration file is as follows:

```
Global parameters;
Declaration1
[parameters   related  to   declaration1]
[ nested  sub  declaration]
Declaration2
[parameters   related   to   declaration2]
[nested   sub  declaration]
```

Each new declaration block applies to a set of clients. Different parameters can be applied to each block of the declaration.

Declarations

You might want to group different clients for various reasons, such as organizational requirements, network layout, and administrative domains. To assist with grouping these clients, you can make use of certain declarations. We discuss these in the following sections.

group Individually listing parameters and declarations for each host again and again can make the configuration file difficult to manage. The **group** declaration allows you to apply a set of parameters and declarations to a list of clients, shared networks, or subnets. The syntax for the **group** declaration is as follows:

```
Group label
      [parameters]
      [subdeclarations]
```

Here, ***label*** is a user-defined name for identifying the group. The ***parameters*** block contains a list of parameters that are applied to the group. The ***subdeclarations*** are used in case a further level of granularity is needed to describe any additional clients that might be a member of the current declaration. Ignore the ***parameters*** field for now. It will be covered in the upcoming section "Parameters."

host A **host** declaration is used to apply a set of parameters and declarations to a particular host in addition to the parameters specified for the group. This is commonly used for fixed address booting or for the BOOTP clients. The syntax for a **host** declaration is as follows:

```
Host label
      [parameters]
       [subdeclarations]
```

Here, ***label*** is the user-defined name for the host group. The ***parameters*** and ***subdeclarations*** are as described in the **group** declaration.

shared-network A **shared-network** declaration groups a set of addresses of members of the same physical network. This allows parameters and declarations to be grouped for administrative purposes. Here's the syntax:

```
shared-network  label
      [parameters]
      [subdeclarations]
```

Here, ***label*** is the user-defined name for the shared network. The ***parameters*** and ***subdeclarations*** are as described in the **group** declaration.

subnet The **subnet** declaration is used to apply a set of parameters and/or declarations to a set of addresses that match the description of this declaration. Here's the syntax:

```
subnet subnet-number netmask netmask
      [parameters]
      [subdeclarations]
```

Here, ***subnet-number*** is the network that you want to declare as being the source of IP addresses to be given to individual hosts. The ***netmask*** is the netmask (see Chapter 12 for more details on netmasks) for the subnet. The ***parameters*** and ***subdeclarations*** are as described in the **group** declaration.

range The **range** declaration specifies the range of addresses that are valid to issue to clients. The syntax is as follows:

```
range [dynamic-bootp] starting-address  [ending-address];
```

The **dynamic-bootp** keyword is used to alert the server that the following range of addresses is for the BOOTP protocol. The ***starting-address*** and optional ***ending-address*** fields are the actual addresses of the start and end blocks of IP addresses. The blocks are assumed to be consecutive and in the same subnet of addresses.

Parameters

We introduced this concept earlier in the chapter. Turning on these parameters will alter the behavior of the server for the relevant group of clients. We'll discuss these parameters in this section.

authoritative The DHCP server will normally assume that the configuration information about a given network segment is not known to be correct and is not authoritative. This is so that if a user unknowingly installs a DHCP server without fully understanding how to configure it, it does not send incorrect DHCPNAK messages to clients that have obtained addresses from a legitimate DHCP server on the network. This parameter's syntax is as follows:

```
authoritative;
not authoritative;
```

default-lease-time This specifies the default length of time (in seconds) for which IP address leases will be allocated to clients if the clients did not request any specific duration. Here is the parameter's syntax:

```
default-lease-time seconds;
```

filename In some applications/environments, the DHCP client may need to know the name of a file to use to boot. This is often combined with **next-server** to retrieve a remote file for installation configuration or diskless booting. This parameter's syntax is as follows:

```
filename filename;
```

fixed-address or fixed-address6 This parameter is only applicable under the **host** declaration. It specifies the set of addresses (IPv4 or IPv6) assignable to the client. This parameter's syntax is as follows:

```
fixed-address address [, address.];
fixed-address6 ip6-address [, ip6-address.];
```

get-lease-hostnames If this parameter is set to **true**, the server will resolve all addresses in the declaration scope and use that for the **hostname** option. This parameter's syntax is as follows:

```
get-lease-hostnames [true | false];
```

hardware For a BOOTP client to be recognized, its network hardware address must be declared using a **hardware** clause in the **host** statement. This parameter's syntax is as follows:

```
hardware hardware-type   hardware-address;
```

Here, ***hardware-type*** must be the name of a physical hardware interface type. Currently, only the Ethernet and Token Ring types are recognized. The ***hardware-address*** (sometimes referred to as the media access control, or MAC, address) is the physical address of the interface, typically a set of hexadecimal octets delimited by colons. The **hardware** statement can also be used for DHCP clients.

max-lease-time A client has the option to request the duration of the lease. The request is granted as long as the lease time doesn't exceed the number of seconds specified by this option. Otherwise, the client is granted a lease to the maximum of the number of seconds specified here. Here is the syntax:

```
max-lease-time seconds;
```

next-server The **next-server** statement is used to specify the host address of the server from which the initial boot file (specified in the **filename** statement) is to be loaded. This syntax is as follows:

```
next-server server-name;
```

Here, ***server-name*** is a numeric IP address or a domain name.

server-identifier Part of the DHCP response is the address for the server. On multi-homed systems, the DHCP server issues the address of the first interface. Unfortunately, this interface may not be reachable by all clients of a server or declaration scope. In those rare instances, this parameter can be used to send the IP address of the proper interface that the client should communicate to the server. The value specified must be an IP address for the DHCP server, and it must be reachable by all clients served by a particular scope. This parameter's syntax is as follows:

```
server-identifier hostname;
```

server-name The **server-name** statement can be used to inform the client of the name of the server from which it is booting. This parameter is sometimes useful for remote clients or network install applications. This parameter's syntax is as follows:

```
server-name Name;
```

Name should be the name that will be provided to the client.

Option	Description
`broadcast-address`	An address on the client's subnet specified as the broadcast address
`domain-name`	The domain name the client should use as the local domain name when performing host lookups
`domain-name-servers`	The list of Domain Name System (DNS) servers for the client to use to resolve hostnames
`host-name`	The string used to identify the name of the client
`routers`	A list of IP addresses for routers the client is to use, in order of preference
`subnet-mask`	The netmask the client is to use

Table 29-1 Common dhcpd.conf Options

use-lease-addr-for-default-route Some network configurations use a technique known as Proxy ARP so that a host can keep track of other hosts that are outside its subnet. If your network is configured to support Proxy ARP, you'll want to configure your client to use itself as a default route. This will force it to use *ARP* (Address Resolution Protocol) to find all remote (off-the-subnet) addresses.

This parameter's syntax is as follows:

```
use-lease-addr-for-default-route [true|false];
```

CAUTION The `use-lease-addr-for-default-route` command should be used with caution. Not every client can be configured to use its own interface as a default route.

Options

Currently, the DHCP server supports more than 60 options. Here is the general syntax of an option:

```
option option-name [modifiers]
```

Table 29-1 summarizes the most commonly used DHCP options.

A Sample dhcpd.conf File

The following is an example of a simple DHCP configuration file:

```
subnet 192.168.222.0 netmask 255.255.255.0 {
    # Options
    option routers 192.168.222.1;
    option subnet-mask 255.255.255.0;
    option domain-name "example.org";
    option domain-name-servers ns1.example.org;
    # Parameters
    default-lease-time 21600;
```

```
    max-lease-time 43200;
    # Declarations
    range dynamic-bootp 192.168.222.25 192.168.222.49;
    # Nested declarations
    host clientA {
          hardware ethernet 00:80:c6:f6:72:00;
          fixed-address 192.168.222.50;
      }
}
```

In this example, note the following:

- A single subnet is defined. The DHCP clients are instructed to use 192.168.222.1 as their default router (gateway address) and 255.255.255.0 as their subnet mask.
- DNS information is passed to the clients; they will use example.org as their domain name suffix and ns1.example.org as their DNS server.
- A lease time of 21,600 seconds is set, but if the clients request a longer lease, they may be granted one that can last as long as 43,200 seconds.
- The range of IP addresses issued starts at 192.168.222.25 and can go as high as 192.168.222.49. The machine with a MAC address of 00:80:c6:f6:72:00 will always get assigned the IP address 192.168.222.50.

To set up your DHCP server with the sample configuration file, all you need to do is create/edit a file named **/etc/dhcp/dhcpd.conf** and populate the file with the entries in our sample file.

We've thoughtfully (we're nice like that!) bundled working copies of the configuration file(s) used in this chapter at the following URL to save you some typing:

http://linuxserverexperts.com/8e/chapter-29.tar.xz

To follow along using this bundle, just download the file (**`wget <preceding URL>`**), extract its contents (decompress it using **`tar xvJf chapter-29.tar.xz`**), and copy the file(s) to the proper directories on your local system.

CAUTION It is of *paramount* importance that you make sure your DHCP server has at least one network interface configured with an IP address that belongs to the same network as the network defined in the subnet declaration of the **dhcpd.conf** file. This means, for example, that you shouldn't have a subnet declaration that reads **`subnet 192.168.222.0 netmask 255.255.255.0`** while your DHCP server has a single network interface (such as ens0) with an IP address of 172.16.0.10.

You should also ensure that the IP address of the DHCP server is excluded from the network range of issued/leased IP addresses specified in the **`range`** declaration. The DHCP server in our example should have a static IP address that lies in the range 192.168.222.1 to 192.168.222.24. If you ignore this warning, the **dhcpd** service will fail to start with errors similar to any of these:

```
No subnet declaration for eth0 (<current-ip-address>) Or
Not configured to listen on any interfaces!   Or
Failed to start DHCP Server Daemon.
```

Starting and Stopping dhcpd

On **systemd**-enabled Linux distros, you can use the **`systemctl`** utility to query the status of the **dhcpd** daemon and to manage the service.

Begin by checking if the service is running:

```
[master@server ~]$ systemctl status dhcpd.service
```

If the output shows that the service is not running (inactive, dead, and so on), use the **`start`** option with the **`systemctl`** utility to start the service:

```
[master@server ~]$ sudo systemctl start dhcpd.service
```

If you've made some **dhcpd.conf** configuration changes and you want the running **dhcpd** server instance to include/implement your changes, you can restart the service by running the following:

```
[master@server ~]$ sudo systemctl restart dhcpd.service
```

To stop the **dhcpd** service, type this:

```
[master@server ~]$ sudo systemctl stop dhcpd.service
```

General dhcpd Runtime Behavior

Once it's started, the daemon patiently waits for client requests to arrive prior to performing any processing. When a request is processed and an address is issued, the daemon keeps track of the address in a file called **dhcpd.leases**. On Fedora, RHEL, and CentOS systems, this file is stored in the **/var/lib/dhcpd/** directory.

On Debian-based distros, such as Ubuntu, the client leases are instead stored under the **/var/lib/dhcp/** directory.

The DHCP Client Daemon

The ISC DHCP client daemon (named **dhclient**), included with many popular Linux distributions, is the software component used to talk to a DHCP server described in the previous sections. If invoked, **dhclient** will attempt to obtain an address from an available DHCP server and then configure its networking configuration accordingly.

Configuring the DHCP Client

The client is typically run from the startup files, but it can also be run manually. It's normally started prior to other services/processes that are dependent on the availability of a functional/configured network stack.

As hinted, the client can be manually invoked from the command line any time after startup. The client daemon can be started without additional options, but it will attempt to obtain a lease on all interfaces configured on the system.

Option	Description
`-p`	Specifies a different User Datagram Protocol (UDP) port for the DHCP client to use instead of the standard port 68.
`-d`	Forces the DHCP client to run as a foreground process instead of its normal behavior of running as a background process; useful for debugging.

Table 29-2 dhclient Command-Line Options

Here is how to start the client from the command line in its most basic form:

```
[master@clientB ~]$ sudo dhclient
......<OUTPUT TRUNCATED>.....
Sending on LPF/ens4/00:0c:29:f8:b8:88
Sending on Socket/fallback
DHCPDISCOVER on lo to 255.255.255.255 port 67 interval 7
DHCPREQUEST on ens4 to 255.255.255.255 port 67
DHCPACK from 192.168.222.1
SIOCADDRT: File exists
bound to 192.168.222.36 -- renewal in 188238 seconds.
```

NOTE On most Linux distros, various mechanisms (such as Scripts, NetworkManager, Netplan, **systemd-networkd**, and so on) exist to help set up the system automatically as a DHCP client between each system reboot so that you won't need to manually run the **dhclient** daemon each time the system needs an IP address. See Chapter 13 to learn more.

Optionally, the client daemon can be started with additional flags that slightly modify the behavior of the application. For example, you can optionally specify the interface (such as eth0) for which an address lease should be requested.

Some of the options for **`dhclient`** are described in Table 29-2. The full syntax of the **`dhclient`** command is shown here:

```
Usage: dhclient [-4|-6][-SNTPI1dvrxi][-nw][-p <port>][-D LL|LLT]
[-s   server-addr]      [-cf   config-file]
[-df   duid-file]       [-lf   lease-file]
[-pf    pid-file]       [--no-pid] [-e VAR=val]
[-sf   script-file]     [interface]
```

Summary

DHCP is a useful mechanism for dynamically configuring the addresses for large groups of machines or mobile devices. Since DHCP is an open and standards-based protocol, the architecture and platform of the server and the client are normally irrelevant.

A Linux server running the popular and highly configurable **dhcpd** server software can service DHCP requests.

Client-side software also exists to easily configure the networking stack of a Linux-based machine as a DHCP client on the network. This client daemon has a number of options that make it able to speak to a variety of DHCP servers.

CHAPTER

30 Virtualization

Simply put, *virtualization* means making something look like something else. Technically speaking, virtualization refers to the abstraction of computer resources. This abstraction can be achieved in various ways: via software, hardware, or a mix of both.

Virtualization technologies have been around in various forms for a long time. This technology has been especially pervasive in recent years due to various factors discussed later on. This chapter discusses some common virtualization techniques on Linux platforms.

We also cover containers (containerization), which provide ways for packaging and delivering applications or even entire operating systems in self-contained environments without the overhead of traditional machine virtualization technologies. Specifically, we discuss the popular Docker platform as our container implementation, and we show how to deploy a web server in a container.

Why Virtualize?

The advantages and the reasons behind the proliferation of virtualization are many and can be grouped broadly into technical and nontechnical factors. The nontechnical factors include the following:

- *Desire for more sustainable (greener) computing model*. Common sense tells us that ten virtual machines running on one physical server have a smaller carbon footprint than ten physical machines serving the same purpose.
- *Cost-saving advantages*. Virtualization helps to save on costs of acquiring and maintaining hardware. Again, common sense tells us that ten virtual machines are, or should be, cheaper than ten physical machines.
- *Greater return on investment (ROI)*. Increased utilization and better leveraging of existing hardware leads to greater ROI for organizations and individuals.

Some of the technical factors include the following:

- *Virtualization improves server and application availability and reduces server downtimes*. Reduced server downtimes can be achieved by using different techniques such as live host migration.
- *Virtualization complements cloud computing*. Virtualization is arguably one of the most important enablers of today's cloud-centric approach to computing, data management, and application deployment.
- *Virtualization offers better cross-platform support*. For example, virtualization makes it possible to run a Microsoft Windows operating system within Linux or to run a Linux-based operating system within Microsoft Windows.
- *Virtualization provides a great environment for testing and debugging new applications and/or operating systems*. Virtual machines can be wiped clean quickly or restored to a known state. In the same vein, virtual machines can be used to test and run legacy or old software. Virtual environments are also often easier and quicker to set up.

Virtualization Concepts

In this section, we try to lay the groundwork for common virtualization concepts and terminologies that appear in this chapter and that are used in everyday discussions about virtualization:

- **Guest OS (VM)** Also known as a virtual machine (VM). The operating system that is being virtualized.
- **Host OS** The physical system on which the guest operating systems (VMs) run.
- **Hypervisor (VMM)** Also referred to as the virtual machine monitor (VMM). A hypervisor provides a CPU-like interface to virtual machines or applications. It is at the heart of the entire virtualization concept and can be implemented with support built natively into the hardware, purely in software, or a combination of both.
- **Full virtualization** Also known as bare-metal or native virtualization. The host CPU(s) has *extended instructions* that allow the VMs to interact with it directly. Guest OSs that can use this type of virtualization do not need any modification. As a matter of fact, the VMs do not know—and need not know—that they are running in a virtual platform. *Hardware virtual machine* (HVM) is a vendor-neutral term used to describe hypervisors that support full virtualization.

  In full virtualization, the virtual hardware seen by the guest OS is functionally similar to the hardware on which the host OS is running.

  Examples of vendor CPUs and platforms that support the required extended CPU instructions are Intel Virtualization Technology (Intel VT), AMD Virtualization (AMD-V), and IBM z Systems.

  Examples of virtualization platforms that support full virtualization are Kernel-based Virtual Machine (KVM), Xen, IBM's z/VM, VMware, VirtualBox, and Microsoft's Hyper-V.

- **Paravirtualization** Another type of virtualization technique. Essentially, this class of virtualization is done via software. Guest operating systems that use this type of virtualization typically need to be *modified*. To be precise, the kernel of the guest OS (VM) needs to be modified to run in this environment. This required modification is the one big disadvantage of paravirtualization. This type of virtualization is currently relatively faster than its full virtualization counterparts.

  Examples of virtualization platforms that support paravirtualization are Xen and UML (User-Mode Linux).

- **Containerization** This is a little difficult to explain/define because it is not a pure virtualization technique in the classical sense. *Containerization* may not even be a word for all we know! But we'll give the definition a shot: containerization refers to a technique for isolating very specific parts/components of the operating system to enable an application or feature to function almost autonomously. (Notice how we cleverly avoided using the word *virtualization* in that definition?)

Even though containerization is somewhat an emerging technique, we should be clear that it is by no means a new technique. The basic ideas and concepts around containerization have been around for a long while in various operating systems. The tools for implementing and managing containerization as well as the possible use cases are what are somewhat new and emerging. One such popular tool is *Docker*. We'll discuss application containerization as implemented in Docker in more detail later in this chapter.

Virtualization Implementations

Many virtualization implementations run on Linux-based systems (and Windows-based systems). Some are more mature than others, and some are easier to set up and manage than others, but the objective of virtualization remains pretty much the same across the board.

We'll look at some of the more popular virtualization implementations in this section.

Hyper-V

This is Microsoft's virtualization implementation. It currently can be used only on hardware that supports full virtualization (that is, Intel VT and AMD-V processors). Hyper-V has a great management interface and is well integrated with the newest Windows Server family of operating systems.

Kernel-Based Virtual Machine (KVM)

This is the first official Linux virtualization implementation to be implemented in the Linux kernel. It currently supports only full virtualization. KVM is *de rigueur* in this chapter.

QEMU

QEMU falls into the class of virtualization called "machine emulators and virtualizers." It can emulate a completely different machine architecture from the one on which it is running (for example, it can emulate an ARM architecture on an x86 platform). The code for QEMU

is open source and mature, and as such it is used by many other virtualization platforms and projects.

VirtualBox

This is a popular virtualization platform. It is well known for its ease of use and nice user interface. It has great cross-platform support. It supports both full and paravirtualization virtualization techniques.

VMware

This is one of the earliest and most well-known mainstream commercial virtualization implementations. It offers great cross-platform support, an excellent user and management interface, and great performance. Several VMware product families are available to cater to various needs (from desktop needs all the way to enterprise needs). Some versions of VMware are free (such as VMware Server and VMware Player), and some are purely commercial (such as VMware vSphere [ESXi], VMware Workstation, and so on).

Xen

This is another popular virtualization implementation in the FOSS world. The code base is quite mature and well tested. It supports both the full and paravirtualization methods of virtualization. Xen is considered a high-performing virtualization platform. The Xen project is sponsored by several large companies, and the open source interest is maintained at www.xenproject.org.

In the Xen world, *domain* is a broad term used to describe the access level in which a VM runs. The two common domains are as follows:

- **Domain 0 (dom0)** This is the control or management domain. It refers to a special VM with special privileges and capabilities to access the host hardware directly. It is often responsible for starting other VMs that run in the user domain (domU).
- **Domain U (domU)** This is the user domain. It is an unprivileged domain where the virtual machines (guests VMs) running within it do not have direct access to the host hardware.

KVM

KVM is the official Linux answer to providing a native virtualization solution via the Linux kernel. KVM gives the Linux kernel hypervisor capabilities. Current stable implementations of KVM are supported on the x86 platforms that support virtualization CPU extensions (such as those provided in the Intel VT and AMD-V lines).

Because KVM is implemented directly in the Linux kernel, it has great support across a wide variety of Linux distros. This means that on a bare-bones KVM setup, you should be able to use the same set of instructions provided in the following sections on any Linux distro.

The **/proc/cpuinfo** pseudo-file-system entry provides details about the running CPU on a Linux system. Among other things, the entry shows the special flags or extensions that the running CPU supports.

TIP You might need to toggle *on* the virtualization switch/option in the system BIOS or UEFI of some systems to enable support for full virtualization. The exact name of the option and sequence for doing this varies from manufacturer to manufacturer, so your mileage may vary. Your best bet is to consult the documentation for your specific hardware. On systems where this is necessary, the Linux kernel may not be able to see and make use of the virtualization flags in the CPU until the proper options are enabled.

On an Intel platform, the flag that shows support for full hardware-based virtualization is the **vmx** flag. To check whether an Intel processor has support for **vmx**, you could **grep** for the desired flag in **/proc/cpuinfo**, like so:

```
[master@intel-server ~]$ grep  -i  "vmx"  /proc/cpuinfo
flags  : fpu pae mce cx8 apic ...<OUTPUT TRUNCATED>... vmx
```

The presence of **vmx** in this sample output shows that necessary CPU extensions are in place on the Intel processor.

On an AMD platform, the flag that shows support for full hardware-based virtualization is the Secure Virtual Machine (**svm**) flag. To check whether an AMD processor has support for **svm**, you could **grep** for the desired flag in **/proc/cpuinfo**, like so:

```
[master@amd-server ~]$ grep  -i  "svm"  /proc/cpuinfo
flags  : fpu vme de pse 3dnowext ...<OUTPUT TRUNCATED>...svm
```

The presence of **svm** in this sample output shows that necessary CPU extensions are in place on the AMD processor.

KVM Example

As mentioned, KVM has great cross-platform/distro support. In this section, we will look at a sample KVM implementation on the Fedora distribution of Linux.

We'll use a set of tools that is based on the libvirt C library. In particular, we will be using the Virtual Machine Manager (**virt-manager**) application tool kit, which provides tools for managing virtual machines. It comprises both full-blown GUI front-ends and command-line utilities.

In this example, we will use the **virt-install** utility, a CLI tool that provides an easy way to provision virtual machines. It also exposes an API to the GUI **virt-manager** application, which then leverages that for various items, such as its graphical VM creation wizard.

Following are the specifications on our sample host system:

- Hardware that supports full virtualization (specifically AMD-V)
- 32GB of RAM and sufficient free storage space on the host disk
- Host OS running Fedora distribution of Linux

For our sample virtualization environment, our objectives are as follows:

- Use the built-in KVM virtualization platform.
- Set up a guest OS (VM) running a Fedora distribution of Linux. We will install Fedora using the install media in the form of an ISO file downloaded and saved as **/media/Fedora-Server-dvd-x86_64*.iso** on the host system. If you want to set up a guest VM running Fedora or some other Linux distro, substitute the sample ISO filename used here with the actual filename of a real ISO file that you possess (for example, **openSUSE-Tumbleweed-DVD-x86_64-Current.iso**, **Fedora-Workstation-Live-*.iso**, **rhel-8.*-x86_64-dvd.iso**, **ubuntu-20.04-desktop-amd64.iso**, and so on).
- Allocate a total of 10GB of disk space to the VM.
- Allocate 2GB RAM to the VM.

We will use the following steps to achieve our objectives:

1. Use **dnf** to install the Virtualization package group. This package group comprises various individual packages (such as **virt-install**, **qemu-kvm**, and **virt-manager**) that provide a virtualization environment.

   ```
   [master@fedora-server ~]$ sudo dnf -y groupinstall 'Virtualization'
   ```

2. On **systemd**-enabled distros, you can start the **libvirtd** service:

   ```
   [master@fedora-server ~]$ sudo systemctl start libvirtd
   ```

3. Use the **systemctl** utility to make sure that the **libvirtd** service starts up automatically during system boots:

   ```
   [master@fedora-server ~]$ sudo systemctl  enable libvirtd
   ```

4. Use the **virsh** utility to make sure that virtualization is enabled and running properly on the system:

   ```
   [master@fedora-server ~]$ sudo virsh  list
   ```

   As long as the previous output does not return any errors, we are fine.

5. On our sample server, we will store all the backing storage files pertaining to each VM under a custom directory path named **/home/vms/**.

   We will begin by creating the directory structure that will house our VM images:

   ```
   [master@fedora-server ~]$ sudo mkdir -p /home/vms/
   ```

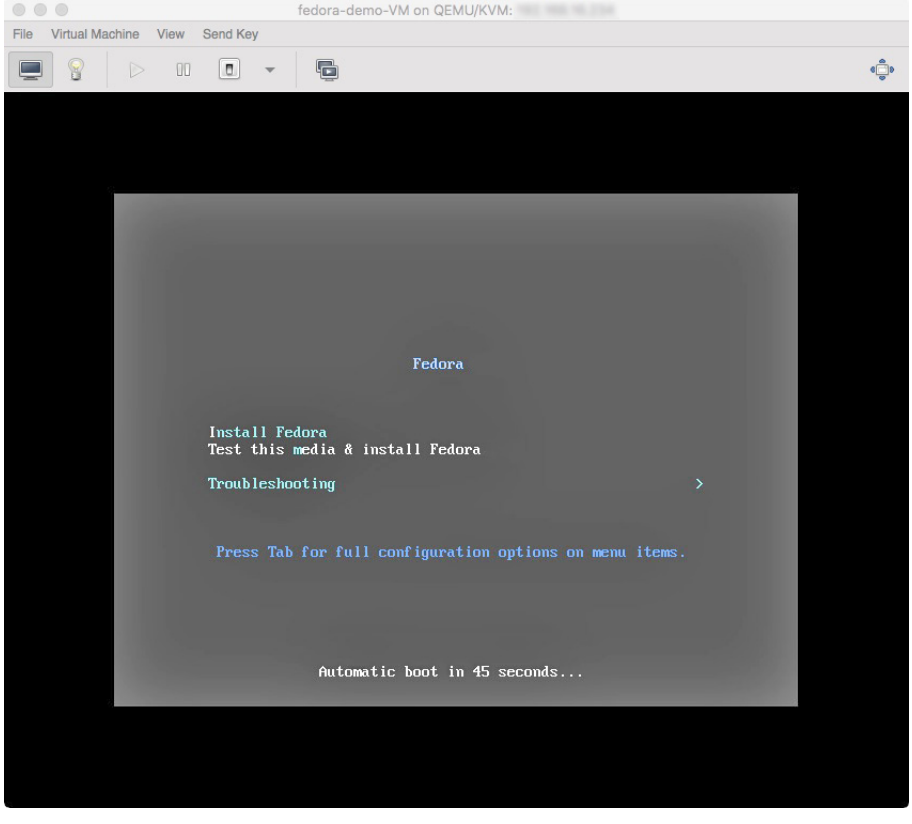
6. Use the **virt-install** utility to set up the virtual machine. The **virt-install** utility supports several options that allow you to customize the new VM at installation time. Launch **virt-install** by running the following:

   ```
   [master@fedora-server ~]$ sudo virt-install  \
    --name fedora-demo-VM --memory 2000 \
    --disk /home/vms/fedora-demo-VM.qcow2,size=10 \
    --cdrom /media/Fedora-Server-dvd-x86_64*.iso
   ```

The meaning of the parameters used in the previous command are explained here:

Option	Description	Value
`--name`	Name of virtual machine/guest instance.	`fedora-demo-VM`
`--memory`	Memory to allocate to VM guest, in mebibytes (MiB).	`2000`
`--disk`	Specifies the path and other storage-related options to the file/device for the VM. If no path or name is specified, the default storage pool path will be used: **/var/lib/libvirt/images/**. The disk will also be automatically named similarly to the VM if you don't specify a name.	`/home/vms/fedora-demo-VM.qcow2,size=10` (We specify the storage name and a size option of 10GB here.)
`--cdrom`	File or device used as a virtual CD-ROM device.	`/media/Fedora-Server-dvd-x86_64*.iso`

7. The newly configured VM should start up immediately in the Virt Viewer window. The VM will attempt to boot from the install media (ISO file) referenced in the value of the `--cdrom` option. A window similar to the one shown here will open:



From here on, you can continue the installation as if you were installing on a regular machine (see Chapter 2). That's it!

TIP The **virt-install** command offers a rich set of options. You should definitely take a minute to look over its manual (**man virt-install**). For example, it has options (**--os-variant**) that will allow you to optimize the configuration for different guest operating system platforms (such as Windows, Linux, UNIX, and so on) out of the box.

Managing KVM Virtual Machines

In the preceding section, we walked through initially setting up a virtual machine. In this section, we will look at some typical tasks associated with managing our guest virtual machines.

We'll use the feature-rich **virsh** program for most of our tasks. **virsh** is used for performing administrative tasks on virtual guest domains (machines), such as shutting down, rebooting, starting, and pausing the guest domains. **virsh** is based on the libvirt C library. **virsh** can run directly from the command line using appropriate options, or it can run directly inside its own command interpreter. We will use **virsh** inside its own shell in the following examples:

1. To start **virsh** in its own minimal interactive shell, type the following:

```
[master@fedora-server ~]$ sudo virsh
virsh #
```

2. **virsh** has its own built-in help system for the different options and arguments that it supports. To see a quick help summary for all supported arguments, type this:

```
 virsh # help
 attach-device      attach device from an XML file
 attach-disk        attach disk device
...<OUTPUT TRUNCATED>...
```

3. To list all the configured inactive and active domains on the hypervisor, type the following:

```
 virsh # list  --all
 Id Name            State
----------------------------
 - fedora-demo-VM      shut off
```

   The output shows that the fedora-demo-VM guest domain is currently shut off (inactive).

4. To view detailed information about the fedora-demo-VM guest domain, type this:

```
virsh # dominfo fedora-demo-VM
Id:             -
Name:           fedora-demo-VM
UUID:           dbbdcf36-6a9f-4f0e-b7a4-562b793c2097
```

```
OS Type:        hvm
State:          shut off
Max memory:     2048000 KiB
Used memory:    2048000 KiB
Persistent:     yes
Autostart:      disable
...<OUTPUT TRUNCATED>...
```

5. Assuming the fedora-demo-VM guest is not currently running, you can start it by running the following:

```
virsh # start fedora-demo-VM
```

6. Use the **shutdown** argument to shut down the fedora-demo-VM guest gracefully:

```
virsh # shutdown fedora-demo-VM
```

7. If the fedora-demo-VM guest has become wedged or frozen and you want to power it off ungracefully (this is akin to yanking out its power cable), type this:

```
virsh # destroy fedora-demo-VM
```

8. To undefine the fedora-demo-VM guest domain or remove its configuration from the hypervisor, type this:

```
virsh # undefine fedora-demo-VM
```

Setting Up KVM in Ubuntu/Debian

We mentioned that one main difference between the virtualization implementations on the various Linux distros is in the management tools built around the virtualization solution.

The KVM virtualization that we set up earlier used management tools (**virt-manager**, **virt-install**, and so on) that work relatively seamlessly across various platforms. Here, we will run through a quick-and-dirty setup of KVM virtualization using lower-level tools that should work with little modification on any Linux distro.

Specifically, we will look at how to set up KVM in a Debian-based distro, such as Ubuntu. The processor on our sample Ubuntu server supports the necessary CPU extensions. We will be installing on a host computer with an Intel VT–capable processor.

The target virtual machine will be any recent copy of the desktop version of Ubuntu and will be installed using the ISO image downloaded from http://releases.ubuntu.com.

1. Install the KVM and QEMU packages. On the Ubuntu server, type this:

```
master@ubuntu-server:~$ sudo apt-get -y install qemu-kvm
```

2. Manually load the **kvm-intel** module:

```
master@ubuntu-server:~$ sudo modprobe kvm-intel
```

NOTE Loading the **kvm-intel** module will also automatically load the required **kvm** module. On an AMD-based system, the required module is instead called **kvm-amd**.

3. We are going to run KVM as a regular user, so we need to add our sample user (*yyang*) to the **kvm** system group:

```
master@ubuntu-server:~$ sudo adduser yyang kvm
```

4. Log out of the system and log back in as the user yyang so that the new group membership can take effect.
5. Create a folder in the user's home directory to store the virtual machine and then change into that directory:

```
yyang@ubuntu-server:~$ mkdir  -p /home/yyang/vms/
yyang@ubuntu-server:~$ cd /home/yyang/vms/
```

6. We'll use the **qemu-img** utility to create a disk image for the virtual machine. The image will be 10GB in size. The file that will hold the virtual disk will be named **disk.img**. Type the following:

```
yyang@ubuntu-server:~/vms$ qemu-img create disk.img -f qcow2 10G
```

TIP The **-f** option specified with the **qemu-img** command is used to specify the disk image format. Here we use the **qcow2** format. This format offers space-saving advantages by not allocating the entire disk space specified up front. Instead, a small file is created, which grows as data is written to the virtual disk image.

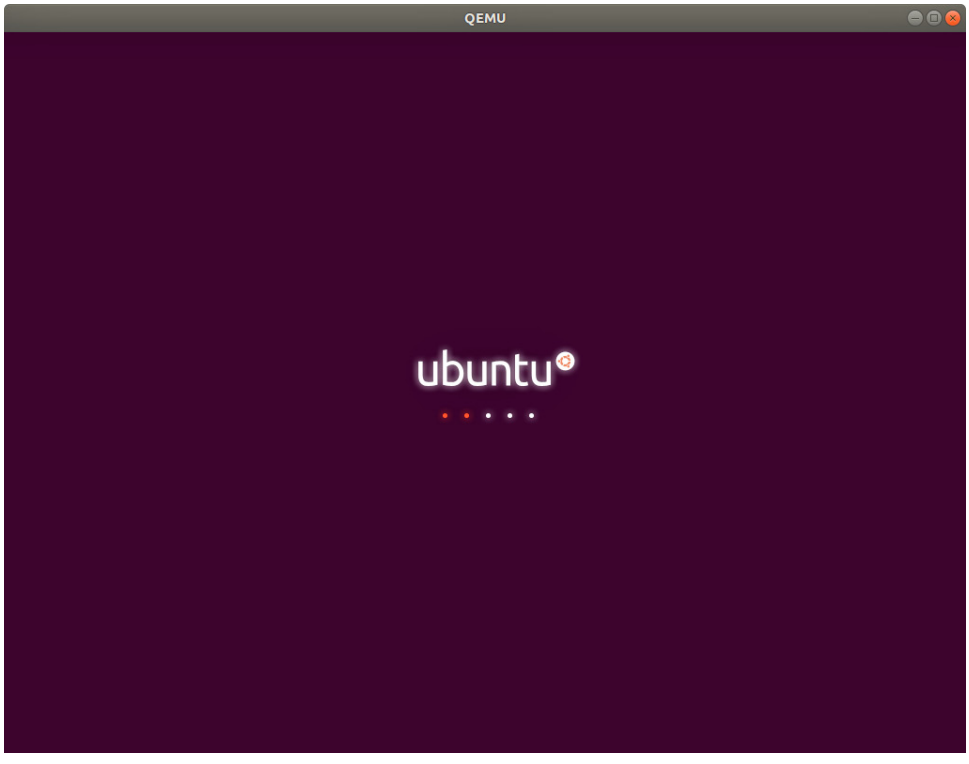
7. Once the virtual disk image is created, we can fire up the installer for the VM by passing the necessary options to the **kvm** command directly. Here's the command:

```
yyang@ubuntu-server:~/vms$ kvm -m 2048  \
-cdrom  ubuntu-20.04-desktop-amd64.iso \
-boot d  disk.img
```

Here are the options that were passed to the **kvm** command:

- **-m** Specifies the amount of memory to allocate to the VM. In this case, we specified 2048MB, or 1GB.
- **-cdrom** Specifies the virtual CD-ROM device. In this case, we point to the ISO image that was downloaded earlier and saved under the current working directory.
- **-boot d** Specifies the boot device. In this case, **d** means CD-ROM. Other options are floppy (**a**), hard disk (**c**), and network (**n**).
- **disk.img** Specifies the raw hard disk image. This is the virtual disk that was created earlier using **qemu-img**.

8. The newly configured VM should start up immediately in the QEMU window. The VM will attempt to boot from the ISO image specified by the **`-cdrom`** option. A window similar to the following will open.



9. From here on, you can continue the installation as if you were installing on a regular machine (see Chapter 2). The particular version of Ubuntu that we use in this example is Live Desktop. Among other things, this means that you can try out the operating system and use it without actually installing/writing anything to disk if you don't want to.
10. Once the operating system has been installed into the VM, you can boot the virtual machine by using the **`kvm`** command:

```
yyang@ubuntu-server:~/vms$ kvm -m 2048 disk.img
```

You will notice in the preceding steps that we didn't need to specify the ISO image as the boot media anymore since we are done with the installation. That's it!

Containers

Containers refers to the class of quasi-virtualization techniques that use various mechanisms to provide isolated runtime environments for applications (or even entire operating systems). The idea behind containers is to provide only the minimum and most portable set of requirements that an application or software stack requires to run. The virtualization solutions (KVM, VMware, and so on) that we discussed earlier in the chapter are geared toward virtualizing almost every aspect of the computing infrastructure, including the hardware, the operating system, and so on.

As already hinted, the basic premise behind containerization has been around in the UNIX world for a while; the only new things these days are the implementations and the supporting tools around the concept. Examples of old and current implementations (and engines) are **chroot**, LXE, **libvirt LXC**, **systemd-nspawn**, Solaris Zones, FreeBSD Jails, **containerd**, Docker, podman, and Kubernetes. Respectively, Docker and Kubernetes are popular container implementations and orchestration platforms. We'll focus on Docker in this section.

Containers vs. Virtual Machines

Even though containers and VMs appear to overlap in functionality, definition, and use cases, they are quite different beasts. Here are some of their similarities and differences:

VMs	Containers
Include an entire operating system.	Need not include an entire operating system.
Host independent. The host OS can be completely different from the guest OS. For example, you can run Windows, macOS, and Linux guest VMs inside a Linux host.	The host OS and the app guest container have to run similar OSs (like for like). For example, you can run a Debian-based container inside a Red Hat–based host system.
Larger system resource footprint and requirements.	Minimal system resource footprint and requirements.
Each VM often needs its own kernel.	Rely on the host system's kernel.
Need a hypervisor.	Do not need a hypervisor.
Slower creation and startup times.	Faster creation and startup times.

Docker

Docker is a set of tools and interfaces used for developing, managing, shipping, and running applications in the form of containers. Docker can make use of various mechanisms to access any required internals/interface of the Linux kernel. Broadly speaking, it uses the namespace and control groups (cgroups) interface exposed in the Linux kernel to provide its isolation and resource-sharing benefits.

Some terms and concepts are unique and commonly used in the Docker world. Here, we outline and explain some of these concepts:

- **Images** Docker images form the building blocks for the Docker ecosystem. They are the read-only blueprints from which containers are created. When Docker runs a container from an image, it adds a read-write layer on top of the image.
- **Containers** Docker containers are the actual workhorses created from Docker images to run the applications.
- **Registry** This refers to any repository or source of Docker images. The default/main registry is a public repository called Docker Hub, which contains thousands of images contributed by various open source projects, companies, and individuals. The registry can be accessed via its web interface (https://hub.docker.com) or via various client tools.
- **Docker Host** Host system on which the Docker application is installed.
- **Docker daemon** The daemon is responsible for managing the containers on the Docker Host where it is running. It receives commands from the Docker client.
- **Docker client** Consists of the user-land tools and applications that issue commands to the daemon to perform container-related management tasks.
- **dockerfile** This is a type of configuration file that describes the steps needed to assemble an image.

Docker Installation and Startup

On our sample Fedora server, we'll walk through installing Docker, starting the daemon service, and enabling it for future automatic startups in the following steps:

1. While logged in as a privileged user, use **`dnf`** to install Docker by running the following:

```
[master@fedora-server ~]$ sudo dnf -y install docker
```

2. After the software has been successfully installed, use **`systemctl`** to check the status of the docker service:

```
[master@fedora-server ~]$ systemctl is-enabled docker
```

3. If the output shows that the service is disabled from automatic startup, you can simultaneously enable it and start it by running this:

```
[master@fedora-server ~]$ sudo systemctl --now enable docker
```

TIP If the docker daemon fails to start on Red Hat–like distros with a Btrfs-formatted file system running SELinux in enforcing mode, you can try working around this issue by removing **`--selinux-enabled`** from the **`OPTIONS`** line in the **/etc/sysconfig/docker** file. Save your changes to the file and then try to start docker again.

TIP Some newer Linux distros have enabled Control Group V2 (cgroupv2) by default, but unfortunately not all applications have been ported to use and take advantage of its advanced features (such as unified hierarchy). To get older docker versions working on such systems during this weird transitionary period, you might need to disable full-fledged cgroupv2 and enable the older cgroupv1 via a kernel boot flag. For example, on an affected EFI Fedora distro, you can append the **systemd.unified_cgroup_hierarchy=0** flag to the **GRUB_CMDLINE_LINUX** parameter in the **/etc/default/grub** file and generate an updated GRUB boot configuration file using **grub2-mkconfig** and then reboot when done. To generate an updated GRUB configuration, run:

```
$ sudo grub2-mkconfig  -o  /boot/efi/EFI/fedora/grub.cfg
```

Using Docker Images

Docker images are the "bits and bobs" that make up containers. You will find thousands of images created by various projects, organizations, and individuals in the official Docker public registry. The registry stores images that are built for various uses, ranging from entertainment/multimedia uses, development environments, and various network servers (HTTP, DNS, VoIP, database and so on), all the way to complete operating systems that have been packaged into containers.

We will walk through setting up a container using a Docker image created by the official Apache HTTP Server Project (http://httpd.apache.org/):

1. Query your local Docker Host for any existing images:

   ```
   [master@fedora-server ~]$ sudo docker images
   ```

   The listing should be empty on a brand-new Docker Host/installation.

2. Search the public Docker registry for images that have the keyword *httpd* in their name:

   ```
   [master@fedora-server ~]$ sudo docker search httpd
   ```

   The output should display various available images. The **NAME** and **OFFICIAL** columns provide hints as to whether the image was uploaded by the official project maintainers or by individuals. The **STARS** column hints at the image popularity.

3. On our sample system, we will select and pull down the httpd image from the official source (**NAME = docker.io/httpd**), which also happens to be the one with the highest rating. Type the following:

   ```
   [master@fedora-server ~]$ sudo docker pull httpd
   Using default tag: latest
   a92eb49846a5: Pull complete
   bed150197318: Downloading [========>           ] 16.24 MB/51.36 MB
   ...<OUTPUT TRUNCATED>...
   ```

4. Query your local image repository to list available images now on your system:

   ```
   [master@fedora-server ~]$ sudo docker images
   REPOSITORY          TAG     IMAGE ID        CREATED        SIZE
   docker.io/httpd     latest  b7cc370ac278    2 weeks ago    132 MB
   ```

Using Docker Containers

We downloaded the official httpd Docker image in the previous section. The image is not terribly useful in its current form, as it is just a bunch of files on our file system. To make the image useful, we have to spin up our very own container from it and customize it for our own use.

1. Without further ado, let's fire up our httpd container:

```
[master@fedora-server ~]$ sudo docker run   \
 --net=host  \
 --name  my-1st-container  \
 -p  80:80  \
 -d    docker.io/httpd
```

   Here are the options we passed to the **docker** command:

   - **--net** Specifies the type of networking we want to use for the container. We've specified the **host** networking option.
   - **--name** This is just a descriptive name to assign to this container instance. We are using **my-1st-container** as the name in this example.
   - **-p** (abbreviated form of **--publish**) Maps or publishes a container's port(s) to the host. Here, we are mapping port 80 (default HTTP port) on the container to port 80 on the host. The syntax is **-p *HostPort*:*ContainerPort***.
   - **-d** (abbreviated form of **--daemon**) Runs the container in the background. This runs the container as a daemon.
   - **docker.io/httpd** Specifies the image name. Here, we are referring to the image that we downloaded earlier.

2. Query **docker** for the list of running containers:

```
[master@fedora-server ~]$ sudo docker ps
```

3. Once the container launches successfully, you can point your web browser to the IP address of the Docker Host to view the default/basic web page being served from the container or use any capable CLI program like **curl** or **wget**.

   The IP address of our sample Docker Host is 192.168.1.100, so we'll use **curl** to browse the web server running in the container, like so:

```
[master@fedora-server ~]$ curl  http://192.168.1.100
```

TIP The actual port that was mapped from the container to the host needs to be accessible through any host-based firewall rules. For this **httpd** container example, you'll need to open port 80 (host port) on the Docker Host. On Red-Hat like distros, you can use the **firewall-cmd** to do this, like so:

```
# firewall-cmd --permanent --add-service=http
# firewall-cmd --reload
```

4. If you've made some changes and you need to restart the container, type the following:

```
[master@fedora-server ~]$ sudo docker restart  my-1st-container
```

5. To immediately stop the **my-1st-container** container process, type this:

```
[master@fedora-server ~]$ sudo docker stop my-1st-container
```

6. To permanently delete the sample (not currently running) **my-1st-container** container, type the following:

```
[master@fedora-server ~]$ sudo docker rm my-1st-container
```

Summary

Numerous virtualization technologies and implementations exist today. Some of them have been around much longer than others, but the ideas and needs remain almost the same. Virtualization and container technologies are smack in the middle of the evolution and adoption of cloud computing technologies.

We looked at common virtualization offerings in the Linux world. We paid particular attention to the KVM platform because of its native and complete integration into the Linux kernel. We showed two examples of actually setting up and using KVM to create two virtual machines on a Fedora box and an Ubuntu server.

Finally, we covered Linux containers with a focus on the Docker application as our implementation. We showed how to find and pick from existing Docker images. We then launched a complete web server (Apache HTTP Server) in a small-sized container that can be easily reused on other Linux distros.

Building on the material in this chapter and other parts of the book, Appendix B provides an in-depth coverage on how to obtain and make use of virtual machine images and containers that were created specially to complement this book. We hope these resources will provide you with hands-on and real-world tools to explore some of the technologies discussed—in a safe virtual environment. We encourage you to download the images and containers, explore them, tear them apart, break them, fix 'em and make 'em better—in true system administrator style!

CHAPTER

31 Backups

Performing backups is a critical part of any server's maintenance, because a production server that is not backed up is a disaster waiting to happen. Managing backups is also one of the responsibilities of the system administrator. Of course, simply performing backups isn't the full extent of your responsibilities; you also have to periodically test the backups and ensure that they are accessible and restorable. These statements are true regardless of which underlying operating system the server is running.

In this chapter, we discuss the backup options that come with Linux distros—our coverage will be mostly at a high level. Many commercial backup solutions exist as well, to suit various budget sizes. The best solution for you depends on your site and its needs.

Evaluating Your Backup Needs

Developing a backup solution is no trivial task. It requires that you consider the interactions among all the parts of your network, the servers, and the resources distributed among them. And, of course, you must arrange for backups to occur regularly and to be verified regularly.

Every organization has different backup needs based on its site(s), its network, and its growth pattern. To make an educated decision, you need to consider the following:

- The amount of data that needs to be backed up
- Backup hardware and backup medium
- The amount of network throughput that needs to be supported
- The speed and ease of data recovery
- Data deduplication issues
- Tape management

We discuss these considerations and others in the following sections.

Amount of Data

Determining an accurate count of the data to be backed up is an important variable when estimating your backup needs. What makes this question tough to answer is that you must factor in anticipated future growth in your determination. And so when you're planning for backup, it's always wise to try and plan as far ahead as is financially feasible.

It is also important to consider how frequently your data changes. Data that changes often (such as data in databases) needs to be backed up frequently and quickly, whereas data that rarely changes (such as the contents of the **/etc** directory) doesn't need to be backed up quite as often. For these and other reasons, multiple backup policies might be needed to fit the types of data you are backing up.

When examining your requirements, take careful stock of compressible versus noncompressible data. With local disk capacities becoming so large, many individuals have taken to storing personal music/image/video collections on their work systems, and of course a lot of this personal data may have nothing to do with the primary business functions. It may be prudent to spell out your organization's policy on this issue so that expectations are set: If users think that all of the systems are being backed up, they may be taken aback when they find out that their personal music collection wasn't covered in that!

Backup Hardware and Backup Medium

The type of hardware you choose should reflect the amount and type of data you need to back up, the frequency with which you're backing it up, and whether it is necessary that backups get rotated to an offsite location.

The common choices available are tape, disk, USB drives, cloud-based solutions, storage area network (SAN), network-attached storage (NAS), and other networked storage arrays. Of these, tape has been around the longest and offers the widest available choices in media density, form factor, and mechanisms.

With all the choices for tape, selecting a specific type can be tricky. Many of the high-density options are appealing for the obvious reason that you can cram more data onto a single tape. Of course, high-capacity tapes and tape drives typically cost more. Work toward finding an optimum solution that backs up the most system data at the best price, balanced with your requirements for media capacity.

NOTE Many advertisements for tape drives boast impressive capacities, but keep in mind that these numbers are for the compressed data on the tape, not the actual data. The amount of uncompressed data on the tape is usually about half the compressed capacity. This is important to know, because compression algorithms achieve various levels of compression, depending on the type of data being compressed. For example, textual data compresses quite well, but certain graphics/audio/video formats get little to no compression at all!

Disk-based backups are very common. The concept is simple: Back up the contents of source drives A, B, and C to destination drives X, Y, and Z. The protocol or interface for communicating with the destination drives is another issue and can be implemented in a variety of ways. If the primary goal of backups is to protect against simple accidents (file deletions, primary disk going bad, and so on), this works well. Transfers are fast, media is

cheap, and building a home-brew Redundant Array of Independent Disks (RAID) system using a low-cost PC, a decent RAID controller, and a few commodity disks can be done relatively cheaply.

TIP If you don't mind getting your hands a little dirty and a very minimal learning curve, you should seriously consider using the free alternatives to the prepackaged commercial storage appliances. Several free and open source solutions are available. One incredibly solid example is the FreeNAS project www.freenas.org.

Using disk-based backups, you can automate scheduled file copies with no tape swapping as well as add additional capacity cheaply. The downside is that offsite storage is not so easy. While getting hot-swappable hard disks is possible, they are not nearly as robust as tape when it comes to being handled and moved. For instance, a tape can be dropped several times with practically no risk, but the same cannot be said of dropping a hard disk!

A combination between fixed (disk) and removable (tape) media is also increasing in popularity. In combination solutions, regular backups are made to disk, with periodic backups moved from the backup disk to tape.

In all of these backup options, you should always plan for possible media failure—that is, plan to move backed-up data to new media every few years. This is necessary to ensure that you're not just using new media, but that the drive itself is still correctly calibrated and modern equipment can still read and write it. In some cases, you might need to consider the data format. After all, data that can be read but not understood doesn't do anyone any good!

Network Throughput

Unfortunately, network throughput is easily forgotten in the planning of backup operations. A really fast backup server and storage subsystem isn't very efficient if you feed in the data through a thin straw! Take the necessary time to understand your network infrastructure. Look at where the data is coming from and where it's going. Use a network monitoring tool such as MRTG (www.mrtg.org) to collect statistics about your switches and routers. Implementing virtual local area networks (VLANs) dedicated to bulk data transfers is a great idea.

Gathering all this information will help you estimate the bandwidth necessary to perform backups. With your analysis done, you'll be able to figure out which new gear or improvements will net you the best returns.

Speed and Ease of Data Recovery

When requests to restore data arrive, you're likely to be under the gun to get the data back to the user(s) as quickly as possible. How long your users have to wait will depend on the tool used for backup. This means you need to incorporate the cost of response time into your backup evaluation. How much are you willing to spend to get the response time you need for a restore?

Disk-based restores are, of course, the fastest. They also offer the possibility of online backups, where users can visit the backup server themselves and copy the file back. Tape, by comparison, is much slower. The specific data on the tape needs to be found, the archive read, and the data extracted. This can all take a little bit of time.

So remember to factor the speed and ease of data recovery into the equation when choosing your backup solution, because there isn't much point in having a high-end and extensive backup solution when the backed-up data is not readily available to the people when they need it most!

Data Deduplication

Besides being a bit of a mouthful to say, data deduplication deals with reducing unnecessary redundant data in a storage system. It's another way of saying "data un-copying."

With the quantity and types of data that most organizations and individuals deal with on a daily basis in today's digital-data-obsessed world, it is easy to see how we could accumulate multiple copies of the same data or files. This unnecessary redundancy is not only inefficient, but it adds to the total costs of performing backups.

We will use the scenario described next to better explain how systems or solutions that are built for solving the problem of redundant data work. Let's pretend that there exists an organization named Penguins 'R' US.

Penguins 'R' US has a standard policy that backups of all user workstations within the organization be performed automatically every weekday at 1:00 A.M.

1. Assume an end user (user A) in the Graphics department downloads five image files from the company's central file server onto her home folder to work on a project at her workstation (station-A). The five files are named **A.jpg**, **B.jpg**, **C.jpg**, **D.jpg**, and **E.gif**.
2. Like clockwork, the next organization-wide backup of user workstations occurs on Monday at 1:00 A.M.
3. At some point during the backup process, the deduplication system will do its best to understand and digest the bits and bytes (or the signature) that make up files **A.jpg**, **B.jpg**, **C.jpg**, **D.jpg**, and **E.gif**.
4. Assume another end user (user B) in the Marketing department downloads seven image files from the company's central file server onto his workstation (station-B) to include in some company marketing material. The seven files are named **A.jpg**, **B.jpg**, **C.jpg**, **D.jpg**, **E.gif**, **X.bmp**, and **Y.gif**.
5. When the time comes to run the regular backups on Tuesday, the data deduplication system detects it already has some files with the exact same signature as the files **A.jpg**, **B.jpg**, **C.jpg**, **D.jpg**, and **E.gif** somewhere in the backup destination. It also notices that the files **X.bmp** and **Y.gif** are brand new on station-B. It creates a signature for the two new files.
6. During the backup process, the deduplication mechanism copies over only the two new files, **X.bmp** and **Y.gif**, on station-B to the destination.
7. Instead of re-copying the files **A.jpg**, **B.jpg**, **C.jpg**, **D.jpg**, and **E.gif** from station-B, the system instead uses some kind of a pointer or marker that points/links to the original and unchanged versions of the files already stored on the destination.
8. The deduplication system, therefore, saves Penguins 'R' US on backup time and storage space. Everybody therefore wins—including you, the brave Linux system administrator!

Tape Management

As the size of your backups grows, so will your need to manage the data you back up. This is where commercial tools often come into play. When evaluating your choices, be sure to consider their indexing and tape management. It does you no good to have 50 tapes' worth of data if you can't find the right file. And, unfortunately, this problem only gets worse as you start needing more tapes for each night's backups.

Managing the Tape Device

The tape device interacts with Linux just as most other devices do: as a file. The filename will depend on the type of tape drive, your chosen mode of operation (auto-rewind or non-rewind), and how many drives are attached to the system.

SCSI tape drives, for example, use the following naming scheme:

Device Name	Purpose
/dev/st*X*	Auto-rewinding SCSI tape device; *X* is the number of the tape drive. Numbering of tape drives is in the order of the drives on the SCSI chain.
/dev/nst*X*	Non-rewinding SCSI tape device; *X* is the number of the tape drive. Numbering of tape drives is in the order of the drives on the SCSI chain.

Let's say, for example, that you have a single SCSI tape drive. You can access it using either of these filenames: **/dev/st0** or **/dev/nst0**. If you use **/dev/st0**, the drive will automatically rewind the tape after each file is written to it. If you use **/dev/nst0**, on the other hand, you can write a single file to the tape, mark the end of file, but then stay at the tape's current position. This lets you write multiple files to a single tape.

NOTE Non-SCSI devices will obviously use a different naming scheme. Unfortunately, there is no standard for naming backup devices if they are not SCSI devices. The QIC-02 tape controller, for example, uses the **/dev/tpqic*** series of filenames. If you use a non-SCSI tape device, you will need to find its corresponding driver documentation to see what device name it will use.

You may find it handy to create a symbolic link from **/dev/tape** to the appropriate device name for the rewinding mode and a link from **/dev/nrtape** for the non-rewinding mode. You can use the following mapping as an example:

- **/dev/tape → /dev/st0**
- **/dev/nrtape → /dev/nst0**

This will make it easier to remember the name of the tape device when issuing commands.

What makes these backup device files different from disk files is that they have no file-system structure. Files are continuously written to the tape until it's full or until an end-of-file marker is written. If a tape device is in non-rewind mode, the write head remains in the position immediately after the last end-of-file marker, ready for the next file to be written.

You can think of tape devices as being similar to a book with chapters. The book's binding and the paper, like the tape itself, provide a place to put the words (the files). It's the markings of the publisher (the backup application) that separate the entire book into smaller subsections or

chapters (positions). If you (the reader) were an auto-rewinding tape drive, you would close the tape every time you were done with a single file and then have to search through the tape to find the next position (chapter) when you're ready to read it. If, however, you were a non-rewinding tape drive, you would leave the tape open to the last page you read.

Using mknod and scsidev to Create the Device Files

If you don't have the **/dev/st0** or **/dev/nst0** file, you can create one using the **mknod** command. The major number for SCSI tape drives is 9, and the minor number dictates which drive and whether it is auto-rewinding. The numbers 0 through 15 represent drive numbers 0 through 15, auto-rewinding. The numbers 128 through 143 represent drive numbers 0 through 15, non-rewinding. The tape drive is a character device.

So, to create **/dev/st0**, we would type this **mknod** command:

```
[master@server ~]$ sudo mknod /dev/st0 c 9 0
```

And to create **/dev/nst0**, we would use this command:

```
[master@server ~]$ sudo mknod /dev/nst0 c 9 128
```

Another choice for creating device names is to use the **scsidev** program (provided by **sg3_utils*.rpm** on Red Hat–like distros and provided by **scsitools*.dpkg** on Debian-like distros). This will create device entries under the **/dev/scsi** directory that reflect the current state of your SCSI hardware, with the appropriate device type (block or character) and corresponding major and minor numbers. This method, unfortunately, has yet another naming scheme.

The naming scheme for tape devices created using **scsidev** is as follows:

```
/dev/scsi/sthA-0cBiTlL
```

Here, **A** is the host number, **B** is the channel number, **T** is the target ID, and **L** is the logical unit (LUN) number.

All the different naming schemes can seem frustrating. The key to all of them, however, is that they are still using the same major and minor numbers. In other words, they all refer to the same driver! In the end, you could decide to call your rewinding and non-rewinding tape devices "Omolara" and "Adere," respectively, so long as they had the correct major and minor numbers.

Manipulating the Tape Device with mt

The **mt** program provides simple controls for the tape drive, such as rewinding the tape, ejecting the tape, or seeking a particular file on the tape. In the context of backups, **mt** is most useful as a mechanism for rewinding and seeking. The **mt** utility is provided as a part of the **mt-st** software package. You can install it on an RPM-based distro such as Fedora, CentOS, or RHEL by running the following:

```
[master@fedora-server ~]$ sudo dnf -y install mt-st
```

Install **mt-st** on Debian based distros via the following:

```
master@ubuntu-server:~$ sudo apt install mt-st
```

Command-Line Backup Tools

Linux distros come with several command-line interface (CLI) tools that can be used for backing up or restoring data. Though some of them lack administrative front-ends, they are still simple to use—and most important, they get the job done! Many formal and more expansive backup packages (even some GUI-based ones) actually use and rely on these utilities in their back-ends.

dump and restore

The **dump** tool works by making a copy of an entire file system. The **restore** tool can then pull any and all files from this copy.

To support incremental backups, **dump** uses the concept of *dump levels.* A dump level of 0 means a full backup. Any dump level above 0 is an incremental relative to the last time a **dump** with a lower dump level occurred. For example, a dump level of 1 covers all the changes to the file system since the last level 0 dump, a dump level of 2 covers all of the changes to the file system since the last level 1 dump, and so on—all the way through dump level 9.

Consider a case in which you have three dumps: the first is a level 0, the second is a level 1, and the third is also a level 1. The first dump is, of course, a full backup. The second dump (level 1) contains all the changes made since the first dump. The third dump (also a level 1) *also* has all the changes since the last level 0. If a fourth dump were made at level 2, it would have all the changes since the third level 1.

The **dump** utility stores all the information about its dumps in the **/etc/dumpdates** file. This file lists each backed-up file system, when it was backed up, and at what dump level. Given this information, you can determine which backup medium to use for a restore. For example, if you perform level 0 dumps on Mondays, level 1 incrementals on Tuesday and Wednesday, and then level 2 incrementals on Thursday and Friday, a file that was last modified on Tuesday but got accidentally erased on Friday can be restored from Tuesday night's incremental backup. A file that was last modified during the preceding week will be on Monday's level 0 tape.

NOTE The **dump** tool comes with most popular Linux distros. If it isn't installed by default in your distro, you should be able to install it easily using the distro's package management system (for example, via **dnf install dump**). This utility is file system dependent, and the version for Linux works only on Linux's native file systems (ext2, ext3, ext4, and Btrfs). For other file systems, such as ReiserFS, JFS, and XFS, be sure to use the appropriate **dump** tool (for example, **xfsdump** or **xfsrestore** for XFS file systems).

Using dump

The **dump** utility takes many parameters, but the most relevant are shown in Table 31-1.

For example, here is the command to perform a level 0 dump to **/dev/st0** of the **/dev/sda1** file system:

```
[master@server ~]$ sudo dump  -0  -f /dev/st0 /dev/sda1
```

dump Command Parameter	Description
`-level#`	The dump level, where ***level#*** is a number between 0 and 9.
`-a`	Automatically sizes the tape. This is the default behavior of dump if **-b**, **-B**, **-d**, or **-s** (as documented later in this table) is not specified.
`-j`	Uses **bzlib** library-based compression. Note that **bzlib**, while being an excellent compression scheme, comes at the expense of needing more CPU. If you use this method of compression, make sure that your system is fast enough to feed the tape drive without the tape drive pausing. Also note that this option might break compatibility with other UNIX systems.
`-z`	Uses **zlib** library-based compression. Note that this option might break compatibility with other UNIX systems.
`-b blocksize`	Sets the dump size to ***blocksize***, which is measured in kilobytes.
`-B count`	Specifies a number (***count***) of records per tape to be dumped. If there is more data to dump than there is tape space, **dump** will prompt you to insert a new tape.
`-f filename`	Specifies a location (***filename***) for the resulting dumpfile. You can make the dumpfile a normal file that resides on another file system, or you can write the dumpfile to the tape device.
`-u`	Updates the **/etc/dumpdates** file after a successful dump.
`-d density`	The ***density*** of the tape in bits per inch.
`-s size`	The ***size*** of the tape in feet.
`-W`	Displays what file systems need to be dumped without actually performing any dumps. This is based on the information in the **/etc/dumpdates** and **/etc/fstab** files.
`-L label`	Labels the dump with a name that can be read by the **restore** command.
`-S`	Performs a size estimate without performing the actual dump.

Table 31-1 Parameters for the dump Tool

Using dump to Back Up an Entire System

The **dump** utility works by making an archive of one file system. If your entire system comprises multiple file systems, you need to run **dump** for every file system. Since **dump** creates its output as a single large file, you can store multiple dumps to a single tape by using a non-rewinding tape device.

Assuming we're backing up to a SCSI tape device, **/dev/nst0**, we must first decide which file systems we're backing up. This information is in the **/etc/fstab** file. Obviously, we don't want to back up files such as **/dev/cdrom**, so we skip those. Depending on our data, we may or may not want to back up certain partitions (such as **swap** and **/tmp**).

Let's assume this leaves us with **/dev/sda1**, **/dev/sda3**, **/dev/sda5**, and **/dev/sda6**. To back up these to **/dev/nst0**, compressing them along the way, we would issue the following series of commands:

```
[master@server ~]$sudo mt -f /dev/nst0 rewind
[master@server ~]$sudo dump -0uf - /dev/sda1 | gzip --fast -c >> /dev/nst0
[master@server ~]$sudo dump -0uf - /dev/sda3 | gzip --fast -c >> /dev/nst0
[master@server ~]$sudo dump -0uf - /dev/sda5 | gzip --fast -c >> /dev/nst0
[master@server ~]$sudo dump -0uf - /dev/sda6 | gzip --fast -c >> /dev/nst0
[master@server ~]$sudo mt -f /dev/nst0 rewind
[master@server ~]$sudo mt -f /dev/nst0 eject
```

The first **mt** command makes sure the tape is completely rewound and ready to accept data. Then come all the **dump** commands run on the partitions, with their outputs piped through **gzip** before going to the tape. To make the backups go a little faster, the **--fast** option is used with **gzip**. This results in compression that isn't as good as normal **gzip** compression, but it's much faster and takes less CPU time. The **-c** option on **gzip** tells it to send its output to the standard out. We then rewind the tape and eject it.

Using restore

The **restore** program reads the dumpfiles created by **dump** and extracts individual files and directories from them. Although **restore** is a command-line tool, it does offer a more intuitive interactive mode that lets you go through your directory structure from the tape. Table 31-2 shows the command-line options for the **restore** utility.

restore Option	Description
-i	Enables interactive mode for **restore**. The utility will read the directory contents of the tape and then give you a shell-like interface in which you can move directories around and tag files you want to recover. When you've tagged all the files you want, **restore** will go through the dump and restore those files. This mode is handy for recovering individual files, especially if you aren't sure which directory they're in.
-r	Rebuilds a file system. If you lose everything in a file system (via a disk failure, for instance), you can simply re-create an empty file system and restore all the files and directories of the dump.
-b *blocksize*	Sets the dump's block size to ***blocksize*** kilobytes. If you don't supply this information, **restore** will figure this out for you.
-f *filename*	Reads the dump from the file ***filename***.
-T *directory*	Specifies the temporary workspace (***directory***) for the restore. The default is **/tmp**.
-v	The verbose option; it shows you each step **restore** is taking.
-y	In the event of an error, automatically retries instead of asking the user if he or she wants to retry.

Table 31-2 Command-Line Options for the restore Utility

Here is a typical invocation of **restore**:

```
[master@server ~]$ sudo restore -ivf /dev/st0
```

This will pull the dumpfile from the device **/dev/st0** (the first SCSI tape device), print out each step **restore** takes, and then provide an interactive session for you to decide which files from the dump get restored.

Should a complete file system be lost, you can re-create the file system using the appropriate **mkfs.*** command and then **restore** to populate the file system. For example, let's say our external SATA drive (**/dev/sdb**) fails, and it had a single partition on it (**/dev/sdb1**) formatted with the ext4 file system. After replacing the failed drive with a new drive and partitioning it, we would re-create the file system like so:

```
[master@server ~]$ sudo mkfs.ext4  /dev/sdb1
```

Next, we have to mount the file system at the appropriate location. We'll assume this is the **/home** file system, so we type the following:

```
[master@server ~]$ sudo mount /dev/sdb1 /home
```

Finally, with the dump tape in the SCSI tape drive (**/dev/st0**), we perform the restoration using the following command:

```
[master@server ~]$ cd /home; sudo restore -rf /dev/st0
```

TIP If you used **gzip** to compress your dump, you'll need to decompress it before **restore** can do anything with it. Simply tell **gzip** to uncompress the tape device and send its output to the standard out. Standard out should then be piped to **restore**, with the **-f** parameter set to read from standard in (stdin). Here's the command:

```
# gzip  -d  -c  /dev/st0 |  restore  -ivf  -
```

tar

Chapter 4 discussed the use of **tar** for creating archives of files. What wasn't discussed, however, is the fact that **tar** was originally meant to create archives of files onto tape (**tar** = **t**ape **ar**chive). Because of Linux's flexible approach of treating devices the same as files, we've been using **tar** as a means to archive and unarchive a group of files into a single disk file. Those same **tar** commands could be rewritten to send the files to tape instead.

The **tar** command can archive a subset of files much more easily than **dump** can. The **dump** utility works only with complete file systems, but **tar** can work on mere directories. Does this mean **tar** is better than **dump** for backups? Well, yes, sometimes.

Overall, **dump** turns out to be much more efficient than **tar** at backing up entire file systems. Furthermore, **dump** stores more information about the file, requiring a bit more tape space, but making recovery that much easier. On the other hand, **tar** is truly cross-platform—a **tar** file created under Linux can be read by the **tar** command under any other Linux platform. In fact, **gzip**-ed **tar** files can even be read by the popular Windows WinZip and WinRAR programs! Whether you are better off with **tar** or **dump** depends on your environment and needs.

rsync

No discussion of traditional open source backup solutions is complete without mentioning the **`rsync`** utility, which is used for synchronizing files, directories, or entire file systems from one location to another. The location could be from a local system to another networked system, or it could be within the local file system. It does its best to minimize the amount of data transmitted by using so-called delta encoding (differences between sequential data) when appropriate. **`rsync`** lends itself to being scriptable and, as such, is easily included in cron jobs or other scheduled tasks that run regularly/automatically on systems.

Many CLI-based and GUI front-ends rely heavily on **`rsync`** (or the librsync library) in their back-end to do the grunt work. Duplicity (http://duplicity.nongnu.org/) is one popular example that provides encrypted bandwidth-efficient backups using the **`rsync`** algorithm/library.

Miscellaneous Backup Solutions

Several open source projects exist that aim to provide enterprise-grade backup solutions. A few of them are AMANDA, Bacula, Dirvish, Mondo Rescue, and BackupPC. They all have their strengths and weaknesses, but all provide robust backup solutions nonetheless. They have nice GUI front-ends to them, which make them attractive to administrators with varying skill levels.

The Advanced Maryland Automatic Network Disk Archiver (AMANDA) is a backup system that allows the use of a single master backup server to back up multiple hosts over the network to tape drives, disks, or optical media. You can learn more about AMANDA at www.amanda.org.

Bacula is a network-based backup suite of programs that allows the backup, recovery, and verification of data across a heterogeneous network of systems. The project web site for Bacula is www.bacula.org.

Dirvish is a disk-based, rotating network backup system. Dirvish is especially interesting because it is adept at and refined for backing up to disk rather than tape. You can learn more about Dirvish at www.dirvish.org.

While not exactly a traditional backup solution, Mondo Rescue is another software product worthy of mention. It is more of a disaster-recovery suite. It supports Logical Volume Management (LVM), RAID, and other file systems. It is usually best to create Mondo archives right after a system has been built and configured. The created Mondo images or archives can be used easily and quickly to restore an OS image back to a bare-bones system. Mondo Rescue can be found at www.mondorescue.org.

BackupPC is another popular open source backup software product for backing up Linux and Microsoft Windows PCs and laptops to a server's disk. The project is hosted at https://backuppc.github.io/backuppc/.

Summary

Backups are one of the most important aspects of system maintenance. Your system may be superbly designed and maintained, but without solid backups, the whole system could be gone in a flash. Think of backups as your site's insurance policy.

Although tape drives are a slightly aging/unfashionable medium, we covered the fundamentals of tape drives under Linux, along with some of the command-line tools for controlling tape drives and for backing up data to tape drives. With this and other information, you should be able to perform a complete backup of your system.

Thankfully, **`dump`**, **`restore`**, **`tar`**, and **`rsync`** are not your only options for backup under Linux. Many commercial and noncommercial backup packages exist as well. Open source projects such as AMANDA, Dirvish, Mondo, and BackupPC are mature and robust choices.

Whichever way you decide to go about the task of backing up your data, just make sure that you do it!

PART

VI

Appendixes

APPENDIX A
Creating a Linux Installer on Flash/USB Devices

APPENDIX B
Demo Virtual Machine and Container

This page intentionally left blank

APPENDIX

A Creating a Linux Installer on Flash/USB Devices

In Chapter 2, we explained the installation process of Fedora and Ubuntu Linux distributions using bootable flash/USB media. However, we didn't go into the details of how to actually create bootable install media on a flash/USB-based device! This appendix presents some methods for quickly creating a Linux distribution installer on inexpensive and readily available and reusable flash-based media, including the various types of USB sticks, SD cards, microSD cards, and so on.

Overview

There are two popular approaches to creating a Linux installer. One approach is to use a native or distro-specific solution; the other approach is more universal. The native or distro-specific approach assumes you have access to an existing and running GNU/Linux-based distro. While this is probably the most ideal or convenient approach, we admit that it is not always possible. On the other hand, the universal approach assumes the opposite—that is, you only have access to *other* platforms, such as Windows or macOS.

Regardless of the approach, the objective remains the same across all platforms (Windows, Linux, and macOS): create bootable and installable flash/USB drives for Fedora, RHEL, openSUSE, Debian, Ubuntu, and other Linux distributions without burning a CD or DVD. We cover all these solutions in more detail in the following sections.

Native Solutions

The native approach involves using common GNU/Linux CLI tools that are almost always available on every Linux/UNIX-based system.

Creating a Linux Installer on Flash/USB Devices (via GNU/Linux OS)

In this section, we assume you have ready access to a system currently running a GNU/Linux distro and can easily run any of the commands from the shell. This is an example of an ideal native solution. Here's what you need for this solution:

- A system that can boot from a flash-based device
- A target flash-based device (at least 4GB in size)
- An ISO or other image file of your Linux distro of choice
- A system running any Linux-based operating system (Fedora, Ubuntu, openSUSE, RHEL, CentOS, and so on)

And here are the steps to follow:

1. While logged into a terminal or shell prompt of your Linux distro as a user with administrative privileges (for example, root), change to a directory on the file system with sufficient free space (use the **`df -sh`** command to check). We'll use the **/tmp** folder in this example:

   ```
   # cd /tmp
   ```

2. Download an ISO file of the Linux distribution onto an area on your file system with sufficient free space. For example, you can download the latest 64-bit Server version of Fedora from https://fedoraproject.org (or https://getfedora.org) by typing

   ```
   # wget  https://<URL>/Fedora-Server-dvd-x86_64-<VERSION>.iso
   ```

   where **<URL>** is the full path to the directory on the download server (for example, https://dl.fedoraproject.org/pub/fedora/linux/releases /<VERSION>/Server/x86_64/iso/).

   The placeholder labeled **<VERSION>** in the filename (**Fedora-Server-dvd-x86_64-<VERSION>.iso**) and in the URL represents the latest version number for Fedora that is available at the time you are reading this. For example, for Fedora server version 34, the actual ISO filename will be something like **Fedora-Server-dvd-x86_64-34-1.2.iso**.

   Similarly, you can find the latest development Rawhide streams of Fedora—which should **NEVER BE USED IN PRODUCTION**—under this tree: https://dl.fedoraproject.org/pub/fedora/linux/development/rawhide/Server/x86_64/iso/. The latest rawhide ISO images are named with the current date stamp appended to their filename (for example, **Fedora-Server-dvd-x86_64-Rawhide-20210130.n.0.iso**).

   For the purposes of this example, assume that the ISO file is 2.1GB. Therefore, we need to make sure that the target flash/USB-based device is at least as big. A 4GB flash drive is perfect in this instance.

3. Plug the flash-based device into the appropriate port on the host system.
4. You need a way to uniquely identify the device file the Linux kernel associates with the flash device that's plugged in. It is very *important* that you properly identify the

target device so that you don't accidentally overwrite the wrong device! CLI utilities such as **mnt**, **blkid**, **fdisk**, **lsusb**, and **dmesg** can help in identifying your desired target device. You might have to run some of these tools before plugging in the new device to note the current state of the system, then run them again after plugging in the device, and then compare both outputs. The difference in the output is almost always the new device! The **dmesg** command allows you to examine the Linux kernel ring buffer and provides one method for identifying the device. The most recent contents or activities in the buffer can be viewed at the tail end of the output. To view the ten most recent entries (lines) of the buffer, type the following:

```
# dmesg | tail
[98492.373438] sd 4:0:0:0: [sdb] Write Protect is off
...<OUTPUT TRUNCATED>...
[98492.381994]  sdb: sdb1
[98492.391192] sd 4:0:0:0: [sdb] Attached SCSI removable disk
```

The output indicates that the kernel has just noticed and enumerated a new device, and the device name is sdb (/dev/sdb). As an aside, we also notice that the device also has one partition—sdb1 (/dev/sdb1).

5. With the previous information, we are now ready to write the downloaded ISO image file to the detected target flash medium (/dev/sdb). We will use the venerable **dd** utility for this purpose, as shown next. Don't forget to replace **<VERSION>** in the command with the correct version number for your ISO file.

```
# dd  if=/tmp/Fedora-Server-dvd-x86_64-<VERSION>.iso   of=/dev/sdb
```

6. Once the previous command runs to completion, you will be returned to your shell prompt. You may now unplug the flash device and insert it into the appropriate port of the new system on which you want to install Linux.
7. If necessary, configure the BIOS or the UEFI of the target system to boot off the flash-based device. Once you have booted into the installer, follow the installation steps as described in Chapter 2, or as specified by the distro vendor.

lsblk, findfs, hwinfo, and Other Utilities

On most Linux distros, several utilities (such as **lsblk**, **blkid**, **findfs**, **hwinfo**, and **lsusb**) are available that enable you to query for and display the various information about block devices attached to a system.

For example, to use **lsblk** to view information (LABEL, UUID, device name, and so on) about all detected devices, type the following:

```
# lsblk -f
```

(*continued*)

Assuming you already know the file system label or UUID of a block device, you can use the **findfs** utility to search the disks attached to the system by their label or UUID. It prints the full device name that matched the specified label or UUID. For example, if you have a flash or USB device with a file system label of DATA, you can use **findfs** to figure out the kernel device file by typing this:

```
# findfs LABEL="DATA"
```

Similarly, when the **hwinfo** utility is installed, you can use it to deduce the full device name of any attached flash or USB devices by running the following command:

```
# grep -Ff <(hwinfo --disk --short) <(hwinfo --usb –short)
```

Distro-Specific Solutions

The distro-specific approach exists because various popular Linux distributions have developed easy-to-use GUI tools that can help end users successfully write the specific distro's ISO image to flash/USB devices. Some popular distro-specific solutions are listed here:

Linux Distribution	Application Name	Binary/Executable Name
Ubuntu	Startup Disk Creator	usb-creator-gtk usb-creator-kde
GNOME variants of Fedora/RHEL/CentOS/ Ubuntu/OpenSuse	Disks	gnome-disks
KDE variants of Fedora/RHEL/CentOS/ openSUSE/Ubuntu/Kubuntu	ISO Image Writer	isoimagewriter

Universal Solutions

The universal approach is dependent on third parties (*not* on specific Linux distribution vendors) creating and maintaining GUI tools that can be used by end users to easily write/ convert the installer image files of different distros into a format that is bootable/usable on a flash or USB device.

Table A-1 shows a mix of old and new applications that can be used for creating bootable install mediums from flash/USB-based devices. We have carefully curated this list based on factors such as availability at the time of writing, cross-platform support across the major OSs, coolness factor, and of course functionality!

Here are the general procedure and requirements for making use of any of these tools:

- A base system you can install or run the chosen application on (see Table A-1)
- A suitably sized flash/USB device (big enough to hold the ISO file for the distro)
- An ISO or other image file of the desired Linux distribution
- A target system that can boot from a flash-based device

Name	URL	Supported Platforms		
		Linux	macOS	Windows
Etcher	https://etcher.io/	Yes	Yes	Yes
Universal USB Installer (UUI)	pendrivelinux.com	No	No	Yes
UNetbootin	https://unetbootin.github.io	Yes	Yes	Yes
Rufus	https://rufus.ie/	No	No	Yes

Table A-1 Third-party Install Media Applications

The following sections walk you through three use cases in which a Linux installer is created on a flash/USB-based device using a system running Windows or macOS.

NOTE Bootable flash/USB-based drive creators come and go. And we literarily mean they *come and go!* Over the years, many FOSS projects have been spun up solely for the purpose of making it easier for people coming from other operating systems to try and/or install any Linux distribution. Overtime, several of these projects have gained mind share and stood the test of time, and others have fizzled away. It's no wonder that some of these applications struggle to keep up with the break-neck speed at which development is done in the FOSS world.

Creating a Fedora Installer Using Etcher on macOS

In this section, we walk through the process of using a sample universal solution (Etcher) to create a Fedora installer on flash/USB media via a system running macOS.

Here's what you'll need:

- A system running any recent macOS operating system—that is, macOS 10.10 (Yosemite) and newer versions
- An ISO or other image file of the desired Fedora version

And these are the steps to follow:

1. While logged into a macOS system, use a web browser to download the latest version of Etcher from the project web site at https://www.balena.io/etcher/.
2. Install the downloaded disk image file (dmg) file for Etcher as you would any other macOS application.

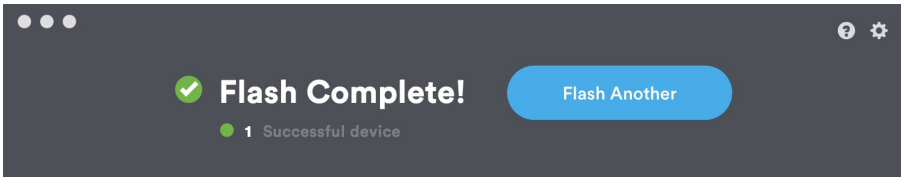
TIP You can also use the built-in macOS "Disk Utility" app to write ISO images to USB or flash medias. In macOS, you can launch Disk Utility via Applications → Utilities or use Spotlight to search for the app.

3. If you haven't already done so, download a copy of the ISO image file for the version of the Fedora distro that you want to write onto the flash/USB device. You can download ISO images for different Fedora versions from https://getfedora.org.
4. Plug the flash/USB-based device into the appropriate port of the host macOS system.

5. Launch Etcher from the Applications menu or via Spotlight Search.
6. Click the Select Image button and then use the Finder app to navigate to and select the distro ISO you downloaded. Click Open after selecting the ISO file.
7. The application will automatically transition to the target selection stage. Click the Select Target button.

   A screen will pop up showing a list of automatically detected eligible removal drives.
8. Identify and select the correct USB/flash medium. A checkmark should appear beside your selection. Click Continue.
9. The application will proceed to the final flash stage. Click the Flash! button.

   You might be presented with another pop-up screen prompting you for a username and password to authorize Etcher (or balenaEtcher) to make changes. Enter the credentials for a privileged user account on the system and click OK.
10. The process will begin and show you a flashing progress and validation indicator.
11. If all goes well, you'll be presented with a "Flash Complete!" screen like the one shown here:



12. Close the application. You may now unplug the flash/USB device and insert it into the appropriate port of the new system on which you want to install the Linux distribution.
13. If necessary, configure the BIOS or the UEFI of the target system to boot off the flash/USB-based device. Once booted into the installer, follow the installation steps as described in Chapter 2, or as specified by the distro vendor.

TIP You might be pleasantly surprised that you can actually boot, install, and run most popular Linux distros on a wide variety of native Apple hardware (MacBooks, Mac minis, iMacs, and so on) using any bootable Linux distro install media.

Creating an OpenSUSE Installer via Universal USB Installer on Windows

In this section, we walk through the process of using a sample universal approach. We'll install and use the Universal USB Installer (UUI) program to create an openSUSE installer on flash/USB media via a system running Windows. Here are the steps:

1. While logged into a system running Windows, use a web browser to download the latest version of UUI from the project web site at www.pendrivelinux.com/universal-usb-installer-easy-as-1-2-3/.

2. If you haven't already done so, download a copy of the ISO image file for the version of the openSUSE distro you want to write onto the flash/USB device. You can start from the main openSUSE home page (www.opensuse.org) to browse and search for the ISO image for your desired openSUSE stream.
3. Plug the flash/USB-based device into the appropriate port on the host system.
4. Double-click the UUI executable (**Universal-USB-Installer-<VERSION>.exe**) you downloaded earlier. Follow all the usual steps as you normally would when installing or launching any Windows-based program.
5. You will next be presented with a screen where you can select the Linux distro to set up. Click the drop-down field labeled Step 1 to reveal a plethora of Linux distros. Search for and select OpenSUSE 64bit from the list.
6. Select the Show All ISOs check box. Use the Browse button in the second field (labeled Step 2) to find and select the openSUSE ISO image you downloaded earlier.
7. Select the Show All Drives check box in Step 3 and then select the appropriate drive letter associated with the flash/USB device from the list. If available, select the check box to format the USB drive you selected.
8. The completed dialog box should reflect the choices made (for example, the path to and name of the ISO image, the drive letter of the target USB device, and so on).
9. Click the Create button to start the process.
10. Click Yes at the next summary screen showing what the program will do.
11. UUI will do its thing and present you with an "Installation Complete" screen if everything goes well. Click the Close button in the final screen to exit the program.

    You may now unplug the flash/USB device and insert it into the appropriate port of the new system on which you want to install Linux.
12. If necessary, configure the BIOS or the UEFI of the target system to boot off the flash/USB-based device. Once booted into the installer, follow the installation steps as described in Chapter 2, or as specified by the distro vendor.

Creating an Ubuntu Installer Using UNetbootin on Windows

UNetbootin is the Swiss Army knife of universal bootable flash/USB device creators. It is supported on multiple platforms—Linux, Windows, and macOS. It has built-in support for creating bootable flash/USB versions of all the popular Linux distros (as well as the not-so-popular ones). It even supports various esoteric utilities (such as FreeDOS, Super Grub Disk, Parted Magic, BackTrack, NTPasswd, Ophcrack, and so on) that often come in handy for system administration tasks. UNetbootin can be used as a stand-alone executable and, as such, does not need to be installed before using it.

As an added bonus, UNetbootin can be used in two modes. You can either explicitly supply an existing ISO image or let the program automatically download your chosen distribution version for you—on the fly!

Assuming you have an Ubuntu ISO version already downloaded, here's what you need:

- A system running any supported and recent member of Microsoft's family of operating systems (Windows 7, 8, 8.1, or 10, XY, and so on)
- A suitably sized flash/USB device (big enough to hold the ISO file for the distro)
- A target system that can boot from a flash-based device

Once you have all the prerequisites in place, download the software from https://unetbootin.github.io. Launch or install the application as you would any other Windows program.

We are going to use an existing distro ISO image here, so select the DiskImage radio button and browse to the existing Ubuntu distro ISO. Follow the prompts and complete the various self-explanatory fields with any information required (Target USB Device Type, Target Device Drive Letter, and so on).

Click OK after completing all the needed fields to start the process. When the process completes, close the UNetbootin window.

You may now unplug the flash/USB device and insert it into the appropriate port of the system on which you want to install your GNU/Linux OS. Boot with the device and continue with the installation as needed.

APPENDIX

B Demo Virtual Machine and Container

We've created and made available to you a virtual machine (VM) image that aims to mimic most of the exercises and projects covered in this book, from the installation chapter (Chapter 2) all the way through the backup chapter (Chapter 31). This VM is based on the native Linux Kernel-based Virtual Machine (KVM) hypervisor implementation. What we've ended up with is a really bloated and ill-advised server that's running/serving various services and daemons. This server can offer web, LDAP, DNS, VoIP, tax advice, file, teeth-whitening, DHCP, fortune-telling, SMTP, accounting/bookkeeping, IMAP and POP, FTP, and legal advice services!

Throughout this book, we've consistently emphasized the importance of running lean servers that run absolutely only the minimum number of services and applications required. This is good for both security and management reasons. As the previous paragraph indicates, the server we've packaged here for you is *not* the kind of lean server that we advocate or you would want in a production environment; however, it is a perfect server for presenting most of the topics and concepts covered in this book in a nice, safe, and controlled environment. Once the VM is powered up, you'll find enshrined in it the following:

- A history of most of the commands we've run throughout this book
- Files, users, groups, daemons, configs, scripts, and applications we created or installed
- Some Easter eggs, as well as some of our very own sysadmin mistakes and typos!

The virtual machine is just a file (albeit a large one), so you can easily duplicate it, destroy it, re-create it, or start from scratch when you make any mistakes—without meaningfully affecting the underlying physical system/computer.

Here is a summary of the steps to take in order to use this VM image:

1. Ensure that your host system meets the minimum system requirement to run the KVM hypervisor. For best performance, the CPU of the host system should at least support the relevant extended CPU virtualization instructions. See Chapter 30 for more details.
2. Install the applications that can be used for managing the hypervisor.

3. Download the VM image file.
4. Make a backup copy of the original VM image file that you downloaded. Make extra copies of the KVM image file if you intend to run multiple instances.
5. Decide how much hardware resources you want to assign to the VM(s). For example, if you only have 8GB of physical RAM on the physical host system, you can safely allocate 2GB of virtual RAM to up to two individual VM instances (such that your host will have approximately 4GB left for itself).
6. Decide which networking options you want to assign to the VM(s), such as bridged networking, NAT-only, host-only, and so on.
7. Fire up the VM(s)!

The following sections cover the requirements and steps in more detail.

Basic Host System Requirements

The host system is the system you will run the VM on. The VM will be constrained by whatever constraints the host has. You should have enough storage/disk space available on the host to download the VM image. The host CPU should support either Intel Virtualization Technology (Intel VT) or AMD Virtualization (AMD-V) extensions. Which tests/commands you execute in the following section depends on your host system platform (Intel or AMD), which corresponds with the CPU type of your host system.

While logged into an Intel-based host system running Linux, check whether the Intel processor shows support for the vmx instruction, by using the **grep** command to search for the desired flag in **/proc/cpuinfo**, like so:

```
[intel-server ~]$ grep  -i  "vmx"  /proc/cpuinfo
flags  : fpu pae mce cx8 apic ...<OUTPUT TRUNCATED>... vmx
```

The presence of vmx in this output shows that the necessary Intel CPU extensions are in place.

While logged into an AMD-based host system running Linux, check whether the AMD processor shows support for the Secure Virtual Machine (svm) instruction, by using the **grep** command to search for the desired flag in **/proc/cpuinfo**, like so:

```
[amd-server ~]$ grep  -i  "svm"  /proc/cpuinfo
flags  : fpu vme de pse 3dnowext ...<OUTPUT TRUNCATED>...svm
```

The presence of svm in this output shows that necessary AMD CPU extensions are in place.

If you aren't sure what your host platform or architecture is, you can use the following command to test for either of the flags:

```
[server ~]$ egrep  -oh  '\w*(vmx|svm)\w*' /proc/cpuinfo
```

Depending on the output and by using a simple process of elimination, you can determine whether your host platform architecture is Intel or AMD. However, if the command does not show either vmx or svm, then your CPU does not support (or you haven't enabled) the necessary extensions that will provide the best experience for using the VM.

Installing the Virtualization Applications and Utilities

Most modern Linux distros have support for KVM built into the kernel by default, so all that's left is to ensure you have all the supporting *userland* tools, utilities, and libraries installed for interacting with the KVM subsystem.

Several such tools are available for creating or managing VMs. We'll keep things simple and use the readily available **`virt-install`** utility, which is based on the libvirt C library, to provision the new VM from an existing image.

Use the following steps to install the needed applications:

1. On a Red Hat–like distro, use **`dnf`** to install the **`virt-install`** application:

   ```
   [master@fedora-host ~]$ sudo dnf -y install virt-install
   ```

2. On a Debian-based distro like Ubuntu, you can install the packages and load the necessary modules by first running

   ```
   master@ubuntu-host:~$ sudo apt-get -y install qemu-kvm
   ```

   and then using the **`modprobe`** command to load the appropriate module for your platform (**kvm-intel** or **kvm-amd**).

3. On a **systemd**-enabled distro, you can simultaneously enable the **`libvirtd`** service for automatic startup and start the **`libvirtd`** service by using the **`systemctl`** utility like:

   ```
   [master@host ~]$ sudo systemctl enable --now libvirtd.service
   ```

4. Use **`virsh`** to make sure virtualization is enabled and running properly:

   ```
   [master@host ~]$ virsh  list
   Id   Name        State
   ----------------------
   ```

As long as the previous output does not return any errors, you are fine.

Download and Prep the Demo VM Image File

The original, uncompressed VM image file is large and contains the entire operating system and lots of applications. We compressed the original file to make it faster/easier for you to download.

Let's start with the steps for downloading and decompressing the file:

1. On our sample host system, we will store all the backing storage files pertaining to each VM under a custom directory path named **/home/vms/**. Let's create that directory:

   ```
   [master@host ~]$ sudo mkdir -p /home/vms/
   ```

2. You will need to obtain the exact download location (URL) for the demo VM image. For the sake of redundancy, we've published this information on a few web sites (just in case any of them become unavailable). You can use your favorite web browser to get the latest/exact download location by navigating to any of the following web sites:
   - https://linuxserverexperts.com/8e/.
   - https://www.mhprofessional.com/9781260441703-usa-linux-administration-a-beginners-guide-eighth-edition. (Click the **Downloads** & **Resources** tab.)
3. Once you've found the exact download link, copy the link to your clipboard (try right-clicking the final link and selecting an option similar to Copy Link...).
4. Substitute ***<DOWNLOAD-LINK>*** with the proper URL in the following command:

   ```
   [master@host ~]$ sudo wget -c -O \
   /home/vms/abg-8e-vm.qcow2.tar.xz  <DOWNLOAD-LINK>
   ```

   For example, if the link you obtained from the first web site is https://linuxserverexperts.com/8e/abg-8e-vm.qcow2.tar.xz, you can use the **wget** program to download the single file by running the following:

   ```
   [master@host ~]$ sudo wget -c -O /home/vms/abg-8e-vm.qcow2.tar.xz \
   https://linuxserverexperts.com/8e/abg-8e-vm.qcow2.tar.xz
   ```

5. Untar (extract and decompress) the image file that you just downloaded:

   ```
   [master@host ~]$ sudo tar -C /home/vms/ -xvJf abg-8e-vm.qcow2.tar.xz
   ```

6. Once this is done, you should end up with a single VM disk image file. Verify this by running the following:

   ```
   [master@host ~]$ ls -lh  /home/vms/abg-8e-vm.qcow2
   ```

Hardware and Networking Notes for VMs

Virtual machines running on a host have to share the finite resources of the host. This is true for almost all the virtual components (disk space, memory, network cards, CPU, USB ports, and so on) that you assign to the VM. If your host system has total storage capacity of, say, 500GB, it would be unwise to allocate a virtual disk of 250GB to more than one VM. Similarly, if the host system has 8GB of physical RAM, it would be unwise to allocate virtual RAM of 4GB to more than one VM. Our point here is that you have to apply a liberal dose of common sense when building your VMs and assigning them virtual hardware from the host.

Different hypervisor platforms handle network provisioning for VMs in various fashions. Particularly, the KVM and libvirt platforms support several methods for handling network connectivity for VMs. Some of these methods revolve around the concept of bridging on the host system, laced with a lot of pure black magic.

(*continued*)

Probably the simplest method is the so-called bridge networking approach. Next in level of complexity are the NAT-only networking and user-mode networking (SLIRP) methods, the latter of which is good for allowing non-privileged users to run and manage their own VMs with some networking functionality. Finally are the more exotic solutions, such as Virtual Distributed Ethernet (VDE). The demo VM walkthrough in this appendix uses bridged networking, which gives the guest VMs full outbound and inbound connectivity to and from the same LAN as the host system.

Import the Demo VM Image and Create a New VM Instance

With all the prerequisites out of the way, we are now ready to instantiate the VM. We'll use the **`virt-install`** utility that was installed earlier. Here are the steps:

1. Make backup copies of the original/pristine VM image, just so that you will always have a clean image to fall back to if necessary. The copies can also be used to spin up additional VM instances if, for example, you decide to set up an entire network of VMs for various purposes.

   While in the **/home/vms** directory, make a copy of the original VM image and name the copy **abg-8e-vm.qcow2.original** by running

   ```
   [master@host vms]$ sudo cp  abg-8e-vm.qcow2{,.original}
   ```

2. Use the **`virt-install`** utility to import the VM image. The **`virt-install`** utility supports several options that allow you to customize the new VM. The requirements/parameters that we use are described in Table B-1. Import the VM by running

   ```
   [master@host vms]$ sudo virt-install  \
   --name  abg-8e-vm  \
   --memory  2000  \
   --disk /home/vms/abg-8e-vm.qcow2,bus=virtio,format=qcow2   \
   --graphics vnc,listen=0.0.0.0,port=5910,password=abg  \
   --noreboot  \
   --import
   ```

Managing the Demo Virtual Machine

In the preceding section, we walked through the process of importing and creating a new virtual machine from an existing image. In this section, we will look at some typical tasks associated with managing guest virtual machines.

We'll use—and encourage you to learn how to use—the feature-rich **`virsh`** program for most of our tasks. **`virsh`** is used for performing administrative tasks on virtual guest domains (machines), such as shutting down, rebooting, starting, and pausing the guest domains.

Option	Description	Value(s)
`--name`	Name of virtual machine/guest instance.	`abg-8e-vm`
`--ram`	RAM (memory) to allocate to VM guest, in MiB.	`2000`
`--disk`	Specifies the path to the VM image you are importing, as well as other storage-related options. Here are the options: `bus` Virtual disk bus type `format` Disk image format	`/home/vms/abg-8e-vm.qcow2,bus=virtio,format=qcow2` (Specifies the path to the VM image file, a bus option of virtio, and format option of qcow2.)
`--graphics`	Specifies the graphical display configuration.	`vnc,listen=0.0.0.0,port=5910,password=abg` (These options will set up a VNC server listening on port 5910 on the host system. The password to use to connect is **abg**.)
`--noreboot`	Prevents the new domain from automatically rebooting after the install or import has completed.	
`--import`	Skips the OS installation process and imports a guest from an existing disk image.	

Table B-1 The virt-install Options

`virsh` can run directly from the command line using appropriate options, or it can run directly inside its own command interpreter. It is also based on the libvirt C library.

We will use **`virsh`** inside its own shell in the following examples:

1. To start **`virsh`** in its own minimal interactive shell, type the following:

```
[master@host ~]$ sudo virsh
virsh #
```

2. **`virsh`** has its own built-in help system for the different options and arguments that it supports. To see a quick help summary for all supported arguments, type this:

```
virsh # help
attach-device      attach device from an XML file
attach-disk        attach disk device
...<OUTPUT TRUNCATED>...
```

3. To list all the configured inactive and active domains on the hypervisor, type the following:

```
virsh # list  --all
Id Name            State
----------------------------
- abg-8e-vm       shut off
```

   The output shows that the **abg-8e-vm** guest domain is currently shut off (inactive).

4. To view detailed information about the **abg-8e-vm** guest, type this:

```
virsh # dominfo abg-8e-vm
Name:           abg-8e-vm
UUID:           54fdc236-4738-4b51-9500-74da8e8f5228
OS Type:        hvm
Max memory:     2048000 KiB
...<OUTPUT TRUNCATED>...
```

5. Assuming **abg-8e-vm** is not currently running, you can start it by running this:

```
virsh # start abg-8e-vm
```

TIP You can run **virsh** commands directly from the Linux shell (for example, Bash) without dropping down into the **virsh** interactive shell by using the **--connect** option with **virsh**. For example, to start our demo VM running on the local hypervisor, we can run either

```
#  virsh --connect qemu:///system start abg-8e-vm
```

or

```
#  virsh  start  abg-8e-vm
```

It is possible to connect to and manage hypervisors on remote hosts, through the use of rich connection Uniform Resource Identifiers (URIs) supported by **virsh** using the **--connect** option.

For example, to connect to and manage KVM on a remote host with the IP address 10.0.0.88, you would run

```
# virsh  --connect qemu+ssh://root@10.0.0.88/system
```

6. Use the **shutdown** argument to shut down the **fedora-demo-VM** domain gracefully:

```
virsh # shutdown abg-8e-vm
```

7. If the **abg-8e-vm** guest domain has become wedged or frozen and you want to power it off ungracefully (this is akin to yanking out its power cable), type this:

```
virsh # destroy abg-8e-vm
```

8. We still need this VM, so do *not* complete this step yet! If you do, you will have to go back to the earlier sections and reimport the original VM image to follow along with the rest of this appendix.

   However, if you ever need to undefine the **abg-8e-vm** guest domain or remove its configuration from the hypervisor, type this:

```
virsh # undefine abg-8e-vm
```

Connecting to the Demo VM

Your demo VM should be up and running at this point (if you stopped at Step 5 in the previous section). However, you shouldn't just take our word for it. In this section, we'll walk through some of the methods you can use to connect to, view, manage, and interact directly with the operating system running in the VM. Here are the options:

- Virtual Network Computing (VNC)
- Virtual serial TTY console
- Secure Shell (SSH)
- Cockpit application
- Just use it!

To make the entire process easy, you should have the following IP addresses readily available to you ahead of time (using any means possible):

IP Address	Tips
Host/hypervisor system	Use **`ifconfig`** or **`ip a`** on the host system to view its current IP address(es).
Demo VM	Check your LAN's DHCP server logs for the newest DHCP requests/leases. Look for clues in the following output: **`"virsh net-dhcp-leases default"`** Log in at the VM's console and use the **`ifconfig`**, **`"ip a"`**, or **`nmcli`** command to view the IP address.

Virtual Network Computing (VNC)

VNC has been implemented as a graphics display option in KVM, as well as *Quick Emulator* (*QEMU*). Normally, a new VNC connection will be set up on every new VM that specifies the VNC graphics option, using a TCP port starting from 5900, 5901, 5902, and so on. To avoid clashing with possibly existing ports, and to better guarantee that you can connect successfully to the VM via VNC, we hard-coded the VNC listening port in the imported VM profile to be port 5910. We also specified a simple password of **abg**.

NOTE The VNC password for the virtual machine is **abg**.

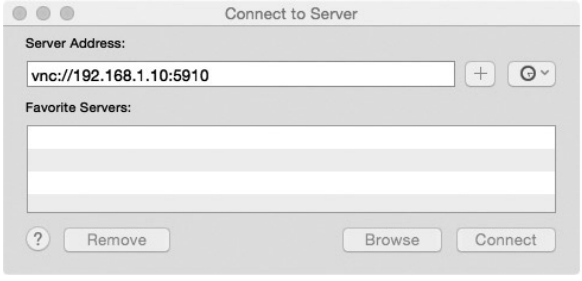
Numerous free VNC clients are available on the Windows, macOS, and Linux operating systems. Most of these platforms even have VNC clients built directly into the standard OS application suite so that you won't need to install any third-party VNC clients (if you don't want to). Both of the popular GNOME and KDE desktop environments in Linux have built-in VNC clients. Same goes for recent macOS versions. Other popular VNC clients include TightVNC, RealVNC, and Chicken of the VNC.

We'll demonstrate using the built-in VNC client on a macOS system here:

1. While logged into a macOS system, launch the Finder application.
2. In the Finder menu, navigate to

3. Finder → Go → Connect to Server.
4. Complete the fields in the ensuing Connect to Server window with the proper information, as shown next. For example, to connect to the demo VM running on a remote Linux hypervisor with the IP address 192.168.1.10, the proper information for the Server Address field is

   vnc://192.168.1.10:5910



5. You will be prompted for the VNC password. Type **abg** when prompted.
6. After connecting and authenticating to the VNC server successfully, you will see a login or console screen within the VNC client window.

   To log into the system as *master*, type **master** at the login prompt and press ENTER.

   At the Password prompt, type **72erty7!2** (master's password) and press ENTER.
7. Alternatively, if you want to log into the system as the superuser (root), type **root** at the login prompt and press ENTER.

   At the Password prompt, type **72erty7!2** (root's password) and press ENTER.

From here on, you can continue to use and interact with the VM just as you would a physical machine or any other VM.

TIP We mentioned the Address Resolution Protocol (ARP) in Chapter 11. ARP provides a mechanism for mapping Ethernet addresses, or Media Access Control (MAC addresses), to IP addresses. MAC addresses are supposed to be universally unique. The tool we used for provisioning the demo VM automatically generates and assigns unique MAC addresses to the network interfaces in a VM.

The MAC addresses for VMs are stored/saved along with other information in a virtual machine profile, usually stored in an Extensible Markup Language (XML) format, under **/etc/libvirt/qemu/**. By combining various command-line utilities that can query, filter, parse, and sort through all of this readily available information, we can obtain the IP address of a running VM from the host.

For example, assuming you've booted up the demo VM and that the VM instance name is **abg-8e-vm**, you can obtain the MAC address and the current IP address for the VM by running the following:

```
#  arp -an | grep \
"`virsh dumpxml abg-8e-vm  | grep "mac address" |  \
 sed "s/.*'\(.*\)'.*/\1/g"`"
```

The sample corresponding output is

```
? (192.168.1.202) at 99:99:00:fe:0f:75 [ether] on br4
```

From this output, we can tell that the IP address for **abg-8e-vm** is 192.168.1.202.

Virtual Serial TTY Console

This connection method is a virtual implementation of a really *old-school* way of connecting to devices using a serial connection or port. No actual physical serial ports or devices are required on the host or the VM for this to work—it's all *virtual*.

The beauty of this connection method is that you don't need to know the IP address of the demo VM ahead of time. However, you do need to be connected to and logged into the host system using any means possible. Follow these steps:

1. While logged into the host server, connect and launch a **`virsh`** shell by running the following:

```
[root@host ~]# virsh -c   qemu:///session
```

2. While at the **`virsh`** shell, list any running VM domains:

```
virsh # list
 Id    Name                           State
----------------------------------------------------
1     abg-8e-vm                        running
```

   Take note of the name of the VM under the **Name** column of the output. In our example, the name is **abg-8e-vm**.

3. Once you've ensured that the demo VM is running, view the virtual TTY console that has been allocated to the running demo VM:

```
virsh # ttyconsole   abg-8e-vm
/dev/pts/8
```

4. Finally, connect to its virtual serial console by executing the following:

```
virsh # console abg-8e-vm
Connected to domain abg-8e-vm
Escape character is ^]
```

5. Press ENTER to see the actual console and the login prompt.
6. To log into the system as *master*, type **master** at the login prompt and press ENTER.

   At the Password prompt, type **72erty7!2** (master's password) and press ENTER.
7. Alternatively, if you want to log into the system as the superuser (root), type **root** at the login prompt and press ENTER.

   At the Password prompt, type **72erty7!2** (root's password) and press ENTER.
8. Once you are logged in successfully, you might want to issue commands to view the VM's IP address and/or change the network settings of the VM to suit your environment.

That's it!

Connecting via SSH

Chapter 23 covered the indispensable Secure Shell (SSH) protocol. As a reminder, SSH is the de facto GNU/Linux protocol for securely accessing and administering remote systems. It has been ported for use on numerous operating systems and platforms, including Microsoft Windows.

Until something better comes along, *every* system/network administrator should have SSH in their toolkit and (of course) know how to *use* it.

The demo VM that you imported earlier already has an SSH server running and listening for connection requests on the default SSH port 22. Remember that, once fired up, the demo VM is just like any other physical machine on your LAN—albeit *virtual.* Therefore, you can connect and manage it remotely. To connect to the SSH server on the demo VM, you will need to know the IP address of the VM instance. You can glean this information from various sources, such as the DHCP server on your LAN, the VM's console, and so on.

In the following steps, we'll assume that you've discovered the IP address of the demo VM and that it's 192.168.1.202. From any system with an SSH client, use **`ssh`** to connect to the VM. Here we use a built-in SSH client from a Linux workstation:

1. Connect to the demo VM as the *root* user using **`ssh`** from the command line:

   ```
   [you@client ~]$ ssh   root@192.168.1.202
   ```

2. If you get a warning about the *authenticity* and fingerprint of the remote host, type **yes** to continue.
3. Enter root's password (**72erty7!2**) when prompted for the password.
4. Once logged in successfully, you'll be at the shell prompt of the demo VM, where you can run numerous commands. For example, to view or change the hostname of the demo VM using the **`systemctl`** command, type the following:

   ```
   [root@fedora-server ~]# hostnamectl set-hostname \
   "kick-ass-server" --static
   ```

5. Use the **`exit`** command to log out.
6. Use SSH to log back into the system formerly known as demo VM to see the new hostname:

   ```
   [you@client ~]$ ssh   root@192.168.1.202
   ```

After logging in successfully, you'll see the new prompt, which should look like this:

```
[root@kick-ass-server ~]#
```

That's it!

Cockpit Application

The latest Fedora, RHEL, CentOS distributions ship with the web-based Cockpit management console installed and enabled by default. Cockpit provides a web-based interface to aid system management. The Cockpit project is still somewhat in its infancy but is very usable and production ready in its current form. You can expect that a lot of new features and capabilities will be added to it over time.

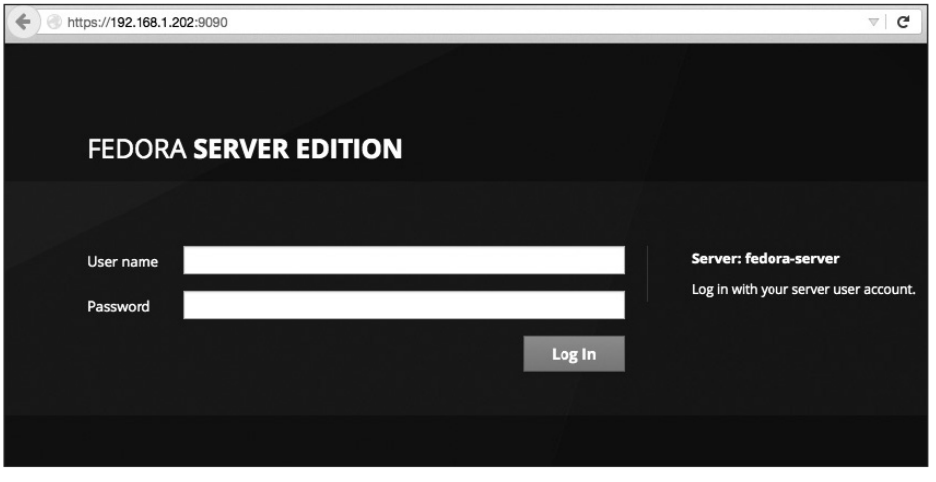
As we've alluded to throughout this book, there's nothing quite like the good ol' command line to get your system/network admin tasks done quickly and efficiently. We've suggested that you forget about all these new fancy GUI-based "point-and-click" tools (like Cockpit) and get familiar/comfortable with using the CLI for System Administration tasks.

But darn it, when we were all poised to dislike it, it turns out Cockpit has a web-based shell console built into it!

The demo VM you imported earlier has its own Cockpit instance already running and listening for connection requests on the default port, 9090. Remember that, once fired up, the demo VM is just like any other physical machine on your LAN. Therefore, you can connect and access its services remotely.

To connect to and use Cockpit, you will need to know the current IP address of the demo VM. The IP address might even be printed for you right at the login console/screen. You can view the login console/screen of the demo VM using any of the methods discussed in the previous sections. Assuming you've discovered the IP address of the demo VM (and that it's 192.168.1.202), from any system within the same network as the demo VM, launch your favorite web browser and visit the following web site/URL (ignore or accept any certificate-related warnings): **https://192.168.1.202:9090/**

You will be greeted with a web page similar to the one shown here:



Type **master** in the **User Name** field. Then type the corresponding password (**72erty7!2**, if you haven't changed it) in the **Password** field. If it's available, select the option to elevate your session so that you can perform privileged actions while using Cockpit. Click the **Log In** button.

After authenticating and logging in successfully to Cockpit, you can view the built-in web-based console/shell/terminal by navigating to Host → Terminal. You will be presented with a familiar command-line terminal and a shell prompt.

TIP We cobbled together a little script that can be used for restoring the demo VM to an almost clean or pre-customization state. Running the script will remove most of the software packages, source code, history of commands, users/groups, configuration files, and other server customizations that were done after the initial operating system installation. You can use **wget** to download the script by running the following:

```
$ wget -c -O clean-up.sh \
 https://linuxserverexperts.com/8e/appendix-b/clean-up.sh
```

After downloading the script, and making double sure you want to clean up (restore) the demo VM server, you can do so by running this:

```
$ chmod 755 clean-up.sh
$ sudo ./clean-up.sh
```

Just Use It!

Out of the box and with the exception of the DHCP service, your demo VM should be running virtually every service/daemon and command that we've covered in this thick book. And so, to take the "Just Use It!" approach, you will need to at least know a little about whichever of the many components, services, or daemons you want to use! In order to use one, you will need to know how to connect to the demo VM via its IP address. The previous sections talked about the various ways to obtain this information.

Say, for example, you want to test the DNS server component we discussed in Chapter 16 inside the demo VM. Assuming the IP address of the demo VM is 192.168.1.202, you can use the **dig** command on your workstation or on the host system to query the DNS server running in the demo VM by running the following:

```
[you@client ~]$ dig  @192.168.1.202  example.org MX
```

Suppose you want to list the Samba shares we configured in Chapter 24 and the IP address of the demo VM (Samba server) is 172.16.1.202. You can use the **smbclient** command on your workstation or on the host system to query the Samba server by running this:

```
[you@client~]$ smbclient -U% -L 172.16.1.202
```

As a final test of the immense and magical capabilities of the demo VM that we lovingly put together for you, let's query it for some random fortune-telling and wise sayings. Assuming the IP address of the demo VM is 10.0.0.202, use the **telnet** program from any computer to connect to port 17 on the demo VM by running the following:

```
you@anywhere ~$  telnet  10.0.0.202    17
Trying 10.0.0.202...
Connected to 10.0.0.202.
Escape character is '^]'.
...<OUTPUT TRUNCATED>...
```

Demo Containers (Docker, podman, buildah, and kubectl)

For extra points, we also created a demo container that *unashamedly* runs way too many of the services we covered in this book. Please note that this container is completely contrary to the function of containers. Containers are supposed to be single-purpose abstractions of a very specific function. The image for the container is located on Docker hub at https://hub.docker.com. First, make sure you have all the client-side Docker tools installed (see Chapter 30). You can also follow along if you instead have other Open Container Initiative (OCI)–compatible tools (**podman**, **buildah**, and so on) installed. Follow these steps to pull down and start using the demo container:

1. While logged into a system as a user with Docker privileges, download the **Dockerfile** into your PWD, as shown next. The directives in the Dockerfile will help pull down and build the correct image from the registry.

```
$ wget -c -O  Dockerfile \
https://linuxserverexperts.com/8e/appendix-b/Dockerfile
```

2. Use **docker** (or **buildah**) to pull down, build, and tag the container image:

```
$ docker  build -t abg8_demo_image:soyinka .
```

   To alternatively use **buildah**, type this:

```
$ buildah bud   -f  Dockerfile  -t abg8_demo_image:soyinka
```

3. Once you have successfully done this, query your local image repository to list available images:

```
$ docker image ls | grep -i soyinka
```

   To alternatively query **buildah** for the images it knows about, type this:

```
$ buildah  images  | grep soyinka
```

4. Create a sample container from the new image:

```
$ docker run --rm -it --name abg8_demo_container \
-p 2222:22  abg8_demo_image:soyinka
```

   To use **podman** to create a container from the **buildah** image, type the following:

```
$ podman  run  -it --rm  --name abg8_demo_container \
-p 2222:22  abg8_demo_image:soyinka
```

5. Use the **exec** option with Docker or **podman** to launch an interactive Bash session inside the running container. For the Docker container, type the following:

```
$ docker  exec  -it abg8_demo_container /bin/bash
```

   To alternatively use **podman**, type this:

```
$ podman  exec -it abg8_demo_container  /bin/bash
```

6. Once safely inside the container, you should be able to explore the completely containerized environment almost as though you were inside a regular system. For example, run the **ps** command inside the container, as follows:

```
[root@baa927a5eeb1 /]# ps ax
PID TTY      STAT   TIME COMMAND
1 pts/0      Ss+    0:00 /usr/sbin/init
6 pts/1      Ss     0:00 /bin/bash
28 pts/1     R+     0:00 ps ax
```

   Because this is a container, you might only see a handful of processes running instead of the usual long list of processes in a regular server or VM.

7. The container has many of the daemons and services discussed in this book preconfigured and running. For example, you can use any SSH client to access the container's SSHD daemon that was mapped to the host's port 2222 (via the **-p 2222:22** option) by running the following:

```
$ ssh  -p 2222  master@localhost
```

8. If you have **docker-compose** installed and want an easy way to play with *all* the services container style, you can download the **docker-compose.yml** file from the following URL:

```
$ wget -c -O  docker-compose.yml \
https://linuxserverexperts.com/8e/appendix-b/docker-compose.yml
```

   Once downloaded into your PWD, use **docker-compose** to gracefully start the container and map all the ports of the various services to ports on the host system:

```
$ docker-compose up
```

9. And if you are feeling super-adventurous and want a slightly more elaborate Kubernetes-style approach, where some of the services/components have been defined declaratively and are segregated into different containers and pods, you can download the following Kubernetes-compatible pod definition:

```
$ wget -c -O  abg8_demo_cluster.yaml \
https://linuxserverexperts.com/8e/appendix-b/abg8_demo_cluster.yaml
```

10. Once this is downloaded, you can bring up the entire cluster using **podman**:

```
$ podman play kube  abg8_demo_cluster.yaml
```

11. And if you already have the entire Kubernetes toolchain installed, you can bring up a native Kubernetes cluster using **kubectl** by running this:

```
$ kubectl apply -f abg8_demo_cluster.yaml
```

   And that's it.

Feedback

We're glad you made to the end of this book! We'd love to hear from you about how you're successfully applying and building upon the concepts/topics in this book to develop your own Linux system/network administration repertoire. One way we'd like for you to get in touch with us is to construct the proper sequence of commands (and options), tweak the SMTP server configs (Chapter 19) on your demo VM or demo container(s), and use the SMTP server *on* the demo VM or demo container to send us a "Hello, World ..." e-mail (showing your e-mail address) at feedback@linuxserverexperts.com.

Thank you and good luck in your system administration journey!

Index

A

A and AAAA (Address) DNS records, 371
a2dismod command, 412
a2dissite command, 415–416
a2enmod command, 412
a2ensite command, 415–416
abbreviations for commands, 276
accept command for printer jobs, 569
access permissions, 123–125
account modules, 127
ACK flags in TCP connections, 250–252
acknowledgment numbers in TCP headers, 248
action field in rsyslog.conf, 193–194
Active Directory (AD), 12–13
active FTP mode, 390
Address (A and AAAA) DNS records, 371
Address of Record (AoR) section in Asterisk, 454
Address Resolution Protocol (ARP), 253–255
adept tool, 102
Advanced Maryland Automatic Network Disk Archiver (AMANDA) system, 609
Advanced Packaging Tool (APT)
    Apache installation, 406
    overview, 99
    Postfix server, 423
    Samba installation, 518
AF_INET sockets, 177
AF_LOCAL sockets, 177
AF_UNIX sockets, 177
aggressive parallelization, 176
Alias option for Apache web server, 415
aliasing IP addresses, 277–278
AMANDA (Advanced Maryland Automatic Network Disk Archiver) system, 609
Amazon Machine Image (AMI) identifiers, 295
Amazon Web Services (AWS)
    overview, 42–43
    VPCs, 292–295
ampersand (&) symbols for multiple commands, 53
Analog Telephone Adapter (ATA), 448
anchor points, 25
AND operation in netmasks, 257–258
anonymous FTP access, 395
anonymous-only FTP servers, 396–397
anycast addresses
    domains, 357
    IPv6, 269
AoR (Address of Record) section in Asterisk, 454
Apache web server, 403
    configuring, 409–416
    HTTP, 403–405
    installing, 406–409
    modules, 406–407
    process ownership, 405
    starting up and shutting down, 407–408
    troubleshooting, 416–417
AppArmor security model, 336
application programming interface (API), 42
application programming interface (API) file systems, 221
    cgroupsfs, 229–230
    /proc directory, 221–227
    SysFS, 227–228
    tmpfs file system, 231
applying patches, 216–217
APT (Advanced Packaging Tool)
    Apache installation, 406
    overview, 99
    Postfix server, 423
    Samba installation, 518
apt-cache tool, 99
apt-get tool, 99–101
aptitude tool, 102
archives, tape, 66–68
ARP (Address Resolution Protocol), 253–255
arptables tool, 300
associate-route-table command, 294
Asterisk software, 451
    configuration files and structure, 453–459
    firewalls, 459–460
    installing, 451–452
    maintaining and troubleshooting, 475–478
    starting and stopping, 452
asterisks (*) for filename expansion, 52
asynchronous replication, 536
ATA (Analog Telephone Adapter), 448
attach-internet-gateway command, 293
attacks, handling, 348–349
attributes
    groups, 134
    LDAP, 544
    users, 133–134
auth logging facility, 190
auth_mechanisms option in Dovecot, 435, 438
auth modules, 127
Auth section in Asterisk, 454–455

authentication
Dovecot, 432, 438
OpenLDAP, 555–558
Windows server, 526–529
authoritative parameter in DHCP, 577
authoritative servers in DNS, 361
authorize-security-group-ingress command, 295
authpriv logging facility, 190
authselect utility
OpenLDAP, 557–558
Samba, 528
autocompletion of commands, 276
autogen.sh script, 109
automated monitoring, 338
availability
mail services, 445
server design, 16–17
virtualization, 584
AWS (Amazon Web Services)
overview, 42–43
VPCs, 292–295
aws tool, 43
aws-cli tool, 43
az tool, 43
Azure component, 43

B

B-tree file system (Btrfs) file system, 158–159
backgrounding tasks, 48–49
backslash (\) characters for escaping, 52
backticks (`) for text execution, 53–54
BackupPC systems, 609
backups, 599
amount of data, 600
command-line tools, 605–609
data deduplication, 602
hardware and media, 600–601
miscellaneous solutions, 609
needs for, 599–604
network throughput, 601
speed and ease of data recovery, 601–602
tape management, 603–605
Bacula systems, 609
bandwidth, monitoring, 348
base chains in Netfilter, 304
Bash package, 87–88
Bash shell, 48
environment variables, 49–51
job control, 48–49
multiple commands, 53
pipes, 51
redirection, 52
text execution, 53–54
.bashrc file, 115
Basic Input/Output System (BIOS), UEFI
replacement for, 24
battery backups for servers, 16–17
bg command, 507
bidirectional TCP connections, 240
Big Iron file systems, 159
bind package, 364
BIND software
configuration file, 364–367
database files, 374–379
DNS servers, 362–363
bind-utils package, 364
binding interfaces, 344–345
BIOS (Basic Input/Output System), UEFI
replacement for, 24
blivet-gui tool, 172
blkid command, 143, 163
block devices, 57
blocks in file systems, 156–157
boot disks in GRUB 2, 143–144
/boot/efi partition, 24
/boot/grub/grub.conf file, 214
/boot/grub2/grub.cfg file, 214
boot loaders
bootstrapping, 144–145
description, 137
GRUB, 138–144
systemd scripts, 145–149
/boot partition
creating, 27
description, 24
booting
Apache web server startup, 408
kernel, 144–145, 214–215
NIC setup at, 278
recovery mode, 152
Bourne Again Shell (Bash). *See* Bash shell
bricks in DFS, 535
broken source code, 109
BSD license, 6
BugTraq mailing lists, 338
build problems in source packages, 108–109
bundles, 83
bzip2 tool, 65

C

caching servers in DNS, 361–362
caching zones in named.conf file, 369
Canonical Name (CNAME) DNS records, 373
capacity in partition information, 26
carbon footprint and virtualization, 583
carpald.sh script, 147–151
case-sensitivity, 58
cat program, 68–69
CD/DVD drives for Fedora installation, 19
CentOS
hostname configuration, 296
NFS, 500–501
NIC setup, 278–280
PAM modules, 126
Postfix server installation, 422–423
RPM, 84
XFS file system, 159
Ceph system, 535
certificates in SSL, 442
cfdisk utility, 168

cgroup-bin package, 229
cgroups (control groups)
resource management, 229, 331
systemd manager, 176–177
cgroupsfs, 229–230
chains for firewalls
example, 322–323
managing, 310–311
types, 303–306
channel request commands in Asterisk, 476
character devices, 57–58
checksums
Ethernet, 239
IP headers, 245
TCP headers, 247
UDP headers, 250
chgrp command, 59
chkconfig utility
Apache web server startup, 408
Postfix server, 422
service shutdown, 346–347
xinetd services, 187
chmod command, 59–61, 124
choke points in NAT, 301
chown command, 58–59
chroot environments, 333–335
CIDR (classless interdomain routing), 258
classes
CUPS, 560
IP addresses, 255–256
classless interdomain routing (CIDR), 258
cleaning up package installation, 107–108
client/server model
FTP, 389–390
LDAP, 543
clients
DHCP daemon, 581–582
DNS, 383–386
Docker, 595
NFS, 506–510
OpenLDAP, 552–554, 557–558
printing tools, 570–571
SSH, 482–483
TCP connections, 250–252
Closest Mirror option, 22
closing TCP connections, 252–253
cloud
network configuration, 288–289
virtualization, 584
cloud server deployment, 39
commercial cloud service providers, 41–43
free-to-run virtual Linux servers, 40–41
CNAME (Canonical Name) DNS records, 373
codecs for VoIP, 450
colons (:) in IPv6 addresses, 268
command line, 47
Asterisk, 475–476
backups, 605–609
Bash, 48–52
documentation tools, 54–56
editors, 70–72
file characteristics, 56–61
file manipulation, 61–69
miscellaneous tools, 72–76
printer tools, 567–568
shortcuts, 52–54
user management, 118–122
commands
autocompletion, 276
entering, 85–86
location, 69
multiple, 53
commas (,) in permissions, 60
comments in named.conf file, 365
commercial cloud service providers, 41–43
commercial distributions, 4
Common UNIX Printing System (CUPS), 559
administration, 568–570
configuring, 562–563
installing, 561
printers, adding, 563
printers, command-line tools, 567–568
printers, default, 568
printers, jobs, 568–569
printers, local and remote, 564–565
printers, Web interface, 565–567, 569–570
running, 560
compatibility of IPv6, 269
compiling
BIND software, 363
kernel, 211–212
OpenSSH, 485
source packages, 106–107
compressing files, 64–65
concatenating files, 68–69
.config file, 207
configuration information in Registry vs. text files, 12
configure scripts
missing, 109
packages, 105–106
Connection State Match option for kernel, 307
connections
NAT, 302–303
network device configuration, 273–277
TCP, closing, 252–253
TCP, data transfer, 251–252
TCP, opening, 250–251
connectivity in mail services, 443–444
containerization in virtualization, 585, 594–598
contexts in dialplans, 457
control_flag entry, 129–130
control groups (cgroups)
resource management, 229, 331
systemd manager, 176–177
control ports in FTP, 390
copying files, 62
core commands in Asterisk, 476
core system services, 175
cron program, 197–199
logging daemon, 187–197
systemd, 175–181
xinetd, 181–187

cost savings from virtualization, 583
cp command, 62
cps variable in /etc/xinetd.conf file, 185
create-internet-gateway command, 293
create-route command, 294
create-route-table command, 293
create-security-group command, 295
create-vpc command, 293
cron logging facility, 190
cron program, 197–199
crontab tool, 197–199
cross-mounting disks in NFS, 508
cross-platform support in virtualization, 584
CUPS. *See* Common UNIX Printing System (CUPS)
cupsd.conf file, 562–564
cupsdisable command, 568
cupsenable command, 568
Cygwin client, 483

D

DAC (Discretionary Access Control) model, 335–336
Daemon entry in vsftpd, 393
daemon logging facility, 190
DATA command in SMTP, 421
data deduplication for backups, 602
data ports in FTP, 390
data recovery from backups, 601–602
data transfer in TCP connections, 251–252
database files in BIND, 374–379
date command, 117
date settings in Fedora installation, 21
dd command, 138
.deb files, 99
Debian distribution, 3–4
    GUI package managers, 102
    hostname configuration, 296
    KVM setup, 591–593
    NFS, 501
    NIC setup, 280–281
Debian Package Management System (DPMS)
    APT toolset, 99
    description, 83, 99
debugging PAMs, 131
declarations in DHCP, 576–577
deduplication for backups, 602
default-lease-time parameter in DHCP, 577
default printers, 568
default routes
    ARP, 254
    network hosts, 285
defragmentation for file systems, 158
deleting
    files and directories, 66
    IP addresses, 275, 277
    OpenLDAP entries, 545, 555
    print jobs, 571
    printers, 568
    routes, 286
    sed text, 72
    users and groups, 134–135
demilitarized zones (DMZs) in NAT, 301
depmod command, 214
describe-instances command, 295
describe-route-tables command, 294
describe-subnets command, 294
Desired Capacity field in partition information, 26
destination addresses
    Ethernet headers, 241–242
    IP headers, 245–246
Destination NAT (DNAT), 301
destination ports in TCP/IP, 342
/dev/null device, 68
/dev/scsi directory, 604
/dev/XYZ pattern, 26
Device Type field for partition information, 26
devices
    block, 57
    character, 57–58
    partition information, 26
    units, 178
df command, 73, 338
DFS (Distributed File Systems), 531
    implementations, 534–540
    overview, 531–534
    terminology, 535–536
dhclient daemon, 581–582
DHCP. *See* Dynamic Host Configuration Protocol (DHCP)
dhcpd.conf file, 575–581
dhcpd daemon, 574, 581
dhcpd.leases file, 581
dialplan show command, 476
dialplans
    PBX configuration, 463–464
    Twilio Elastic SIP Trunks, 473–474
    VoIP, 457
Differentiated Services Code Point (DSCP) field in IP headers, 244
dig (domain information gopher) tool, 380–381
directories
    creating, 65
    LDAP, 542, 553–555
    overview, 56
    removing, 66
    users, 114
    working, 66
Directory entry
    users, 114
    vsftpd, 394
directory information tree (DIT) in LDAP, 542
Dirvish systems, 609
disable_plaintext_auth option in Dovecot, 435
disable variable in /etc/xinetd.conf file, 184
disabling
    Apache modules, 412
    echo, 186–187
    Network Manager, 282
    printers, 568
    SELinux, 452
    services, 150–151, 346
disconnected mode for mail, 431
Discretionary Access Control (DAC) model, 335–336

disks
adding, 165–166
free space, 73
layout, 169–170
monitoring, 338
mounting and unmounting, 159–163
partitions, 165–166
utilization information, 72–73
displaying
files, 69
routes, 287
distinguished names (DNs) in LDAP, 542
Distributed File Systems (DFS), 531
implementations, 534–540
overview, 531–534
terminology, 535–536
distributions overview, 3–4
DIT (directory information tree) in LDAP, 542
dmesg command, 165
DMZs (demilitarized zones) in NAT, 301
DNAT (Destination NAT), 301
dnf package manager, 96–97
Apache web server, 406
Dovecot, 432–433
OpenLDAP, 545
Postfix server, 422–423
RPM installation, 89–91
DNS. *See* Domain Name System (DNS)
DNs (distinguished names) in LDAP, 542
docker command, 597
Docker tool
containers, 594–598
images, 596
installing, 595–596
virtualization, 585
dockerfile file, 595
documentation
command-line tools, 54–56
source packages, 104–105
DocumentRoot option for Apache web server, 411
domain information gopher (dig) tool, 380–381
Domain Name System (DNS), 355
BIND database files, 374–379
client configuration, 383–386
description, 12
domain and host naming conventions, 356–358
hosts file, 356
in-addr.arpa domain, 360
record types, 369–373
server configuration, 367–369
server installation, 362–367
server setup, 374–379
server types, 361–362
subdomains, 358–359
tools, 379–383
troubleshooting, 266–267
domains
description, 12–13
naming conventions, 356
root, 357
second-level, 358
subdomains, 358–359
third-level, 358
top-level, 357
Xen, 586
doveadm tool, 439
doveconf utility, 434–438
dovecot.conf file, 434
Dovecot software
configuring, 434–438
installing, 432–433
overview, 431–432
running, 439
services and modules, 441
downloading
BIND software, 363
Fedora, 19
kernel source code, 203–204
OpenSSH, 484–485
patches, 216–218
software, 89–92
Ubuntu, 32
virtual machines, 40–41
downstream components, 6–7
dpkg application
DHCP installation, 575
overview, 99–100
DPMS (Debian Package Management System)
APT toolset, 99
description, 83, 99
drivers
Dovecot, 436
kernel modules, 210
DSCP (Differentiated Services Code Point) field in IP headers, 244
dselect tool, 102
du command, 72–73
dump tool for backups, 605–607
dumpfiles, reading and writing, 265
dynamic-bootp keyword in DHCP, 577
Dynamic Host Configuration Protocol (DHCP), 573
client daemon, 581–582
configuring, 575–582
Fedora installation, 30
installing, 574–575
ISC, 362
mechanics, 573–574
server installation, 18
servers, 574–581
starting and stopping, 581
dynamic routing IP addresses, 260–264

E

E.164 telecommunication numbering plan, 473
e-mail
IMAP. *See* Internet Message Access Protocol (IMAP)
POP. *See* Post Office Protocol (POP)
SMTP. *See* Simple Mail Transfer Protocol (SMTP)
ebtables tool, 300
EC2 (Elastic Compute Cloud), 42–43
Echo-Request packets, 241

echo service, 71
    appending text with, 174
    enabling and disabling, 186–187
    example entry, 186
ECN (Explicit Congestion Notification) field
  in IP headers, 244
editors, 70–71
EFI system partition (ESP), 24
Elastic Compute Cloud (EC2), 42–43
emacs editor, 70–71
emergency power for servers, 16–17
enabling
    Apache modules, 412
    echo service, 186–187
    Netfilter, 306–307
    NFS, 500–501
    printers, 568
    services, 150
encryption
    passwords, 113
    public key cryptography, 479–481
    Samba, 516–517
Endpoint section in Asterisk, 455–456
entities, 111
enumerated /proc entries, 223–225
environment variables
    Bash shell, 49–51
    as parameters, 53
environmental considerations in server installation, 15–16
equal-cost multipath in OSPF, 264
equal signs (=) in permissions, 60
ErrorLog option in Apache web server, 414
ESP (EFI system partition), 24
/etc/aliases file, 427
/etc/apache2/apache2.conf file, 410
/etc/apache2/sites-available directory, 416
/etc/asterisk/extensions.conf file, 457, 463, 473–474
/etc/asterisk/pjsip.conf file, 461–463, 471–473
/etc/asterisk/rtp.conf file, 459
/etc/cgconfig.conf file, 230
/etc/cron.allow file, 197
/etc/cron.deny file, 197
/etc/cups directory, 562
/etc/dhcp directory, 575
/etc directory, 12
/etc/dovecot/conf.d/10-mail.conf file, 437
/etc/dovecot/conf.d directory, 441
/etc/dumpdates file, 605
/etc/exports file, 503–506
/etc/firewalld file, 317
/etc/fstab file, 151, 162–163, 173, 231
/etc/group file, 117–118, 121
/etc/grub.d directory, 140
/etc/HOSTNAME file, 296
/etc/hosts file, 356
/etc/httpd/conf.d/welcome.conf file, 409
/etc/httpd/conf directory, 410
/etc/init.d directory, 346–347
/etc/inittab file, 145, 180
/etc/ld.so.conf.d directory, 109
/etc/ld.so.conf file, 109
/etc/modprobe.d directory, 272
/etc/named.rfc1912.zones file, 369
/etc/network/interfaces file, 280–281
/etc/nslcd.conf file, 547
/etc/nsswitch.conf file, 385–386, 527–528
/etc/openldap/ldap.conf file, 552
/etc/openldap/slapd.conf file, 548–551
/etc/pam.d directory, 127
/etc/pam.d/other file, 131
/etc/passwd file, 112
    Directory field, 114
    GECOS field, 114
    GID field, 114
    Password field, 113
    Shell field, 116
    UID field, 113–114
    Username field, 113
/etc/pki/dovecot/ dovecot-openssl.cnf file, 438
/etc/postfix directory, 423
/etc/resolv.conf file, 384–385
/etc/rndc.conf file, 383
/etc/rsyslog.conf file, 189–190
    format, 190–191
    mail logs, 427
    rules, 192–194
    sample, 194–195
/etc/samba/smb.conf file, 519–520, 527
/etc/security directory, 127
/etc/security/limits.conf file, 330–332
/etc/services file, 344
/etc/shadow file, 116–117
/etc/shells file, 116
/etc/ssh/ssh_config file, 487–488, 493
/etc/sssd/sssd.conf file, 547
/etc/sysconfig/docker file, 595
/etc/sysconfig/network file, 285, 296
/etc/sysconfig/network-scripts directory, 275, 278–279
/etc/systemd/network directory, 281
/etc/systemd/resolved.conf file, 386
/etc/vsftpd.conf file, 392
/etc/vsftpd/vsftpd.conf file, 392, 396–397, 399
/etc/xinetd.conf file
    attributes, 183–185
    format, 182–183
/etc/xinetd.d directory, 346
/etc/xinetd.d/echo file, 187
Ethernet cards in Fedora installation, 30
Ethernet headers, 241–243
Ethernet layer in TCP/IP, 239
events in upstart daemon, 180
everyone permissions, 123
Execute permission, 123
executing kernel, 144–145
exit command, 80
expansion of filenames, 52–53
Explicit Congestion Notification (ECN) field
  in IP headers, 244
export command, 50–51
exportfs command, 504–505

ext4 (fourth extended file system), 157
extended instructions in virtualization, 584
Extensible Markup Language (XML), 240–241
Extensible Messaging and Presence Protocol (XMPP), 449
extensions, PBX, 461
extensions.conf file, 457, 463–464
extents
    file systems, 158
    volumes, 166
external IP addresses, 289
EXTRAVERSION variable for kernel, 211

F

F Virtual Window Manager (FVWM), 11
facilities in log messages, 188–190
fault tolerance in DFS, 535–536
fdisk utility, 168
Fedora, 6
    GUI package manager, 97–98
    hostname configuration, 296
    NFS, 500–501
    NIC setup, 278–280
    PAM modules, 126
    Postfix server installation, 422–423
Fedora installation, 18
    localization settings, 21
    process, 31–32
    project prerequisites, 19
    Settings section, 30–31
    software settings, 21–22
    starting, 20
    system settings, 22–30
fg command, 49
FHS (Filesystem Hierarchy Standard), 6–7
file handles in NFS, 509
file systems, 155
    adding disks, 165–166
    components, 155–159
    creating, 172–174
    diagnosing and repairing, 163
    GRUB 2, 139
    layouts, 24–25
    managing, 159–165
    partition information, 26
    Ubuntu installation, 34–36
    volume management, 166–172
File Transfer Protocol (FTP), 389
    mechanics, 389–390
    server installation, 17
    sftp program, 493
    vsftpd. *See* Very Secure FTP Daemon (vsftpd) program and package
filename expansion, 52–53
filename parameter in DHCP, 577
files
    characteristics, 56–61
    command-line manipulation, 61–69
    compressing, 64–65
    concatenating, 68–69
    copying, 62
    displaying, 69
    finding, 63–64
    groups, 59
    linking, 63
    listing, 58
    modes, 59–61
    moving, 62–63
    ownership, 58–59
    PAMs, 126–127
    redirection, 52
    removing, 66
    size, 158
Filesystem Hierarchy Standard (FHS), 6–7
Filetransfer entry in vsftpd, 394
FileZilla client, 483
filter tables in Netfilter, 300
filters, printing, 560
FIN packets in TCP connections, 252
find command, 63–64
finding files, 63–64
firewall-cmd command
    Asterisk, 459
    NFS, 506
firewalld manager, 317–320
firewalls, 299
    Asterisk, 459–460
    cloud, 289
    iptables command, 322–324
    Netfilter. *See* Netfilter firewall
fixed-address parameter in DHCP, 577
flags field
    IP headers, 244
    TCP headers, 247
flows in NAT, 301
fork bombs, 331–332
FORWARD chains in Netfilter, 305
forward references in zone files, 368
forward resolution in DNS, 360
FOSS (Free and Open Source Software), 3–4
    mail servers, 431
    SMTP, 420
    standards, 7
fourth extended file system (ext4), 157
FQDNs (fully qualified domain names), 356–357
fragment offset field in IP headers, 244
frames in TCP/IP, 236
Free and Open Source Software (FOSS), 3–4
    mail servers, 431
    SMTP, 420
    standards, 7
free space on disks, 73
free-to-run virtual Linux servers in server cloud deployment, 40–41
freemium model, 4
fsck tool, 151, 163–164
FTP. *See* File Transfer Protocol (FTP)
Full NAT option in kernel, 307
full virtualization, 584
fully qualified domain names (FQDNs), 356–357
fuser program, 161
FVWM (F Virtual Window Manager), 11

G

G.711 codec, 450
G.722 codec, 450
G.729A codec, 450
Gates, Bill, 9
GCC (GNU Compiler Collection), 86
gcc package installation, 91–92
GCE (Google Compute Engine), 43
gcloud tool, 43
GCP (Google Cloud Platform), 43
GECOS field for users, 114
GET command in HTTP, 404
get-lease-hostnames parameter in DHCP, 578
Ghostscript interpreter, 560
GID field in users, 114
GIDs (group IDs), 111, 117, 510
Git system, 219
gluster command, 539
glusterd daemon, 537–538
GlusterFS system, 534, 536–540
gnome-control-center system applet, 122
gnome-disks command, 172
gnome-terminal emulator, 86
GNOME window manager
    description, 11
    Fedora, 85, 97
    terminal emulators, 86
    virtual terminals, 47
GNU Compiler Collection (GCC), 86
GNU Public License (GPL), 6
GNU systems, 5–6, 103
    encryption, 113
    PAM modules, 131
    source packages, 103–108
    tape archive, 66
    texinfo system, 56
    vi editor, 70
Google Cloud Platform (GCP), 43
Google Compute Engine (GCE), 43
GPL (GNU Public License), 6
graphical.target target, 330
graphical user interface (GUI)
    entering commands in, 85–86
    limitations, 47
    package managers, 97–98, 102
    performance, 16
    separation from kernel, 10–11
    user managers, 122–123
greater-than symbols (>) for redirection, 52
grep utility
    pipes, 51
    redirection, 52
group declarations in DHCP, 576
group IDs (GIDs), 111, 117, 510
Group option in Apache web server, 413
group variable in /etc/xinetd.conf file, 184
groupadd command, 121, 133
groupdel command, 121, 134–135
groupmod command, 122, 134
groups
    adding, 121, 133
    control, 229–231
    files, 59
    identifiers, 111
    modifying, 122
    permissions, 59–60, 123
    removing, 121, 134–135
    security, 294–295
    systemd manager, 176–177
    users, 114
    volumes, 170
GRUB boot loaders, 138
    conventions, 138
    GRUB 2, 139–144
grub.cfg file, 139–141
grub-install script, 138
grub2-install utility, 144
grub2-mkconfig command, 140
GSM codec, 450
guest OS in virtualization, 584
GUI. *See* graphical user interface (GUI)
gutenprintcups RPM package, 566
gzip utility, 64–65

H

H.323 protocol, 449
hard command in NFS, 507
hard links, 57
hard mounts in NFS, 508
hardphones, 448
hardware
    backups, 600–601
    server considerations, 15–16
hardware compatibility lists (HCLs), 15
hardware parameter in DHCP, 578
hardware virtual machines (HVMs), 584
HCLs (hardware compatibility lists), 15
head command, 73
headers
    HTTP, 404
    processes, 74–75
    TCP/IP, 236, 241
    TCP/IP, Ethernet, 241–243
    TCP/IP, IP, 243–246
    TCP/IP, TCP, 246–249
    TCP/IP, UDP, 249–250
hello package, 103–108
HELO command in SMTP, 420
high-volume server issues, 227
home directories
    moving, 78–80
    NFS, 512
    users, 114
/home partition, 25, 115
host declarations in DHCP, 576
host OS in virtualization, 584
host tool for DNS, 379–380
hostnamectl tool, 296–297

hostnames, configuring, 29–30, 295–297
hosts
    Docker, 595
    IP addresses, 255–256
    naming conventions, 356
hosts.txt file, 355
httpd-*.rpm package, 406
HVMs (hardware virtual machines), 584
Hyper-V virtualization, 585
Hypertext Transfer Protocol (HTTP), 403
    headers, 404
    ports, 405
    process ownership, 405
    server installation, 17
    TCP/IP layer, 240–241
hypervisors, 584
hyphens (-) in permissions, 59

I

i-nodes
    file systems, 155–156
    hard links, 57
ICMP (Internet Control Message Protocol)
    description, 241
    Netfilter firewall, 318
ICMP Time Exceeded message, 244
id variable in /etc/xinetd.conf file, 184
identification field in IP headers, 244
Identify section in Asterisk, 455
ifconfig program, 273–275
ifup command, 281
IGs (internet gateways), 293–294
IHL (Internet Header Length) field in IP headers, 244
iLBC (Internet Low Bitrate Codec), 450
images
    Docker, 595–596
    GRUB 2, 139
    server installation, 18–19
    Ubuntu installation, 32–33
IMAP. *See* Internet Message Access Protocol (IMAP)
imap-login service section, 441
in-addr.arpa domain, 360
Include option for Apache web server, 413
include statement in named.conf file, 365
indirection, i-nodes for, 156
inet_protocols option for Postfix, 425
inetd program, 346
info command, 56
information queries in RPM, 86–88
init command
    kernel execution, 144–145
    legacy notes, 179–180
initctl command, 181
initial RAM disk in GRUB 2, 141
initramfs file in kernel installation, 215
initramfs option in GRUB, 141
initrd option, 141
INPUT chains in Netfilter, 305
INSTALL files, 104–105
Installation Destination option in Fedora installation, 23
installing
    Apache web server, 406–409
    Asterisk, 451–452
    BIND software, 363
    CUPS, 561
    DHCP, 574–575
    DNS servers, 362–367
    Docker, 595–596
    Dovecot, 432–433
    GlusterFS, 537
    kernel, 212–214
    Netfilter firewall, 306–308
    OpenLDAP, 545–546
    OpenSSH, 483–486
    Postfix server, 422–423
    Samba, 518
    server. *See* server installation
    snap apps, 108
    software, 89–92, 100–101
    source packages, 106–107
    X-Lite softphones, 464
instances variable in /etc/xinetd.conf file, 184
interface variable in /etc/xinetd.conf file, 185
interfaces
    binding, 344–345
    iptables command, 313
Internet Control Message Protocol (ICMP)
    description, 241
    Netfilter firewall, 318
internet gateways (IGs), 293–294
Internet Header Length (IHL) field in IP headers, 244
Internet Low Bitrate Codec (iLBC), 450
Internet Message Access Protocol (IMAP), 429–431
    basics, 431
    checking, 440–441
    connectivity, 443–444
    Dovecot, 431–439
Internet Printing Protocol (IPP), 564
Internet Systems Consortium (ISC), 362–363
Internet telephony service provider (ITSP), 467
intr command, 507–508
IP addresses. *See also* TCP/IP (Transmission Control Protocol/Internet Protocol)
    aliasing, 277–278
    cloud, 288–289
    configuring, 274–276
    DHCP. *See* Dynamic Host Configuration Protocol (DHCP)
    DNS. *See* Domain Name System (DNS)
    dynamic routing, 260–264
    headers, 243–246
    hosts and networks, 255–256
    iptables command, 312
    IPv6, 268–270
    NAT, 301–303
    netmasks, 257–258
    network configuration, 28–30
    port numbers, 341–342
    static routing, 258–260
    subnetting, 256–257
    Twilio Elastic SIP Trunks, 471

IP layer in TCP/IP, 239
IP phones, 448
ip utility for network device configuration, 272–277
ip6tables tool, 300
IPMI protocol, 18
IPP (Internet Printing Protocol), 564
iproute package, 272, 278, 288
iproute2 package, 272
iptables
    chains, 303, 310–311
    description, 310
    firewalls, 322–324
    NAT, 320–321
    Netfilter, 300
    Netfilter firewall, 306
    rule-specification, 312–317
iptables-save command, 320
IPv6 addresses, 268–270, 276–277
ISC (Internet Systems Consortium), 362–363
ISO images
    Fedora installation, 19
    Ubuntu installation, 32–33
ITSP (Internet telephony service provider), 467

J

jitter in VoIP, 477
jobs
    Bash shell, 48–49
    printer, 559, 568–569
    upstart daemon, 180–181
joe editor, 71
journalctl tool
    Apache web server, 417
    logging data, 195–197
journald daemon, 195
journaling
    examples, 195–197
    file systems, 157–158

K

kern logging facility, 190
kernel
    booting, 214–215
    building, 204–205
    compiling, 211–212
    configuring, 205–210
    description, 3
    differences, 4–5
    executing, 144–145
    forms, 10
    GUI separation from, 10–11
    installing, 212–214
    loading, 144
    modules, 210
    Netfilter, 307–308
    NFS support, 502–503
    overview, 201–203
    patching, 216–219
    source code, 203–204
    upgrades, 205
    versions, 203–204
kernel-based virtual machine (KVM)
    example, 587–590
    managing, 590–591
    overview, 585–587
    setting up, 591–593
kernel-install command, 214
keyboard settings in Fedora installation, 21
keys
    public key cryptography, 479–481
    SSL, 442
killing processes, 75–76
konsole emulator, 86
Kubuntu hostname configuration, 296
KVM. *See* kernel-based virtual machine (KVM)
kvm-amd module, 592
kvm command, 592–593
kvm-intel module, 592

L

languages in Fedora installation, 20–21
latency in VoIP, 477
layers in TCP/IP, 235
    Ethernet, 239
    HTTP, SSL, and XML, 240–241
    IP, 239
    packets, 236–238
    Physical, 238
    TCP and UDP, 239–240
LD_LIBRARY_PATH environment variable, 109
LDAP Data Interchange Format (LDIF), 544, 553–554
ldapadd utility, 554
ldapdelete utility, 555
ldapsearch utility, 554–555
ldconfig command, 109
ldd command, 333–335
LDIF (LDAP Data Interchange Format), 544, 553–554
leases in DHCP, 574
length field in TCP headers, 249
less command, 69
less-than signs (<) for redirection, 52
/lib/security directory, 126
/lib/systemd/network directory, 281
/lib64/security directory, 126
libcgroup package, 229
libcgroup-tools package, 229
libdb-utils package, 398
libnss_ldap library, 547
libnss_winbind library, 527
library problems, 108–109
libvirtd service, 588
Lightweight Directory Access Protocol (LDAP), 541
    basics, 541–542
    client/server model, 543
    directories, 542
    OpenLDAP, 544–554

terminology, 543–544
uses, 543
Lightweight X11 Desktop Environment (LXDE), 11
limited resources, 330–332
limits module, 315
Line Printer Daemon (LPD), 565
link-local routes, 285
links
files, 63
hard, 57
symbolic, 57
Linode, 42
Linux Security Modules (LSM) framework, 336
Linux Standard Base (LSB) specification, 6–7
Listen option
Apache web server, 410
Dovecot, 435–436
listing
files, 58
logged in users, 77
processes, 73–75
live operating system environments for server installation, 18
ln command, 63
loading kernel, 144
LoadModule option in Apache web server, 412
local disks, mounting and unmounting, 159–163
local extensions in PBX, 461
local logging facilities, 190
local printers, adding, 564–565
local security, 325–326
limited resources, 330–332
nonhuman user accounts, 330
risk mitigation, 332–336
risk sources, 326–329
runlevels, 330
system monitoring, 337–338
local user logins in vsftpd, 395
localization settings in Fedora installation, 21
locating commands, 69
log_on_success variable in /etc/xinetd.conf file, 185
LOG Target Support option in kernel, 307
log_type_from variable in /etc/xinetd.conf file, 185
logged in users, listing, 77
logger command, 188
logging daemon, 187–197
Logging entry in vsftpd, 394
logging statement in named.conf file, 365–366
Logical Volume Management (LVM)
description, 166
partitions, 27–28
logical volumes (LVs)
creating, 168, 171–172
description, 166–167
login command in IMAP, 440
.login file, 115
login processes
Dovecot, 432
Fedora, 31–32
LogLevel option in Apache web server, 414
logout command in IMAP, 441
logrotate program, 337
logs
local security, 337
mail services, 445
parsing, 347
Postfix server, 427–428
storing entries, 347–348
logs directory for error files, 416
Long Term Support (LTS) release in Ubuntu, 32
loopback addresses, 256
loopback routes, 285
lost+found directory, 164–165
lpadmin command, 567–569
LPD (Line Printer Daemon), 565
lpinfo command, 567
lpq command, 571
lpr command, 570
lpr logging facility, 190
lprm command, 571
lpstat utility, 567, 569
ls command, 58
LSB (Linux Standard Base) specification, 6–7
lsb_release command, 77
lshw command, 205
lsinitrd command, 215
LSM (Linux Security Modules) framework, 336
lsof program, 161
lspci command, 205
lssubsys utility, 229
LTS (Long Term Support) release in Ubuntu, 32
Lustre system, 534
lvcreate command, 169
lvdisplay command, 169
LVM (Logical Volume Management)
description, 166
partitions, 27–28
LVs (logical volumes)
creating, 168, 171–172
description, 166–167
LXDE (Lightweight X11 Desktop Environment), 11
lynx application
installing, 89–90
uninstalling, 93

M

MAC addresses. *See* Media Access Control (MAC) addresses
MAC (Mandatory Access Control) model, 336
mail_access_groups option in Dovecot, 437–438
mail access protocols, 429
mail delivery agents (MDAs), 422
Mail Exchanger (MX) DNS records, 372–373
MAIL FROM: command in SMTP, 420
mail_location option in Dovecot, 435, 437
mail logging facility, 190
mail processes in Dovecot, 432
mail queues in Postfix server, 427

mail services
availability, 445
IMAP. *See* Internet Message Access Protocol (IMAP)
logs, 445
POP. *See* Post Office Protocol (POP)
SMTP. *See* Simple Mail Transfer Protocol (SMTP)
SSL/TLS security, 442–444
mail_spool_directory option in Postfix, 424–425
mail spools in NFS, 512
mail transport agents (MTAs), 421
mail user agents (MUAs), 421
mailing lists in local security, 338
mailq command, 427
mailx mail user agent, 427
main.cf file, 423–425
mainline kernels, 203
major kernel versions for patches, 216
major numbers for block devices, 57
make tool
kernel compilation, 211–212
package compilation, 106–107
makefiles
kernel, 205
targets, 205–206
man command, 55–56
Mandatory Access Control (MAC) model, 336
mangle tables in Netfilter, 300, 304, 306
manifests in netplan, 283
MANs (metro area networks), 239
Manual Partitioning screen in Fedora installation, 23–24
manual storage configuration overview, 24
mapped addresses in IPv6, 269
mark logging facility, 190
MASQUERADE Target Support option for kernel, 307
Masquerading in NAT, 301
Master Boot Record (MBR), 137–138
master.cf file, 423
master processes in Dovecot, 432
Match feature in iptables command, 314–317
max-lease-time parameter in DHCP, 578
Maximum Segment Size (MSS) value in TCP connections, 250
maximum transmission unit (MTU) in IP headers, 244
MaxRequestWorkers option in Apache web server, 412
mbox format, 429
MBR (Master Boot Record), 137–138
MDAs (mail delivery agents), 422
Media Access Control (MAC) addresses
ARP, 254
DHCP declarations, 578
Ethernet headers, 241–243
static routing, 258
media for backups, 600–601
Media Gateway Control Protocol (MGCP), 449
media images in server installation, 18
media protocols for VoIP, 450
media verification step in Fedora installation, 20
members in groups, 117
meta-packages, 101
metadata in DFS, 535
methods in HTTP, 404
metro area networks (MANs), 239
MGCP (Media Gateway Control Protocol), 449
micro-kernels, 10
Microsoft DFS (MS-DFS) system, 534
migration scripts in OpenLDAP, 556
MindTerm client for SSH, 483
minicom package, 348
minor numbers for block devices, 57
minus signs (–) in permissions, 60
mirrors
kernel, 203
server installation, 19
Ubuntu installation, 34
missing configure scripts, 109
mkcert.sh script, 438
mkdir command, 65
mkfs.xfs tool, 172–173
mkinitramfs command, 214
mknod command, 604
mkswap command, 163
mod_ Apache modules, 406–407
modes, file, 59–61
modifying
groups, 122, 134
users, 120, 133–134
modinfo command, 272
modprobe command, 320, 322
module_path entry, 128
module_type entry, 127–130
modules
Apache web server, 406–407
kernel, 210
network configuration, 271
VoIP, 458
modules.conf file, 458–459
Mondo Rescue systems, 609
monitoring
bandwidth, 348
system, 337–338, 347–348
monolithic kernels, 10
more command, 69
mount command
NFS, 506–508
options, 160–161
risk source, 327–328
mount points in partitions, 25–26
mount units in systemd manager, 178
mounting
description, 11
local disks, 159–163
partitions, 499–500
remote Samba shares, 524
moving
files, 62–63
users, 78–80
MRTG (Multi-Router Traffic Grapher), 348
MS-DFS (Microsoft DFS) system, 534
MSS (Maximum Segment Size) value in TCP connections, 250
mt utility, 604, 607
MTAs (mail transport agents), 421

MTU (maximum transmission unit) in IP headers, 244
MUAs (mail user agents), 421
Multi-Router Traffic Grapher (MRTG), 348
multi-user.target, 330
multicast addresses in IPv6, 269
multiple commands, 53
multiple user systems, 9–10
mutt mail user agent, 427
mv command, 62–63
MX (Mail Exchanger) DNS records, 372–373
My Network Places, 11–12
mydestination option in Postfix, 424
mydomain option in Postfix, 424
myhostname option in Postfix, 424
mynetworks option in Postfix, 425
mynetworks_style option in Postfix, 425
myorigin option in Postfix, 424

N

name resolution in GlusterFS, 537
name server database files, 369–373
Name Server (NS) DNS records, 371
named.conf file
    statements in, 364–367
    zones, 367–369
names
    domain and host, 356
    groups, 117
    partitions, 164–165
NAT. *See* network address translation (NAT)
NAT of Local Connections option for kernel, 308
National Security Agency (NSA), 336
ncurses-devel package, 207
Nessus application, 350
net command in Samba, 528
Netfilter firewall, 299
    chains, 303–304
    configuring, 308–319
    cookbook solutions, 320–324
    firewalld manager, 317–320
    installing, 306–308
    iptables command, 310–317
    NAT, 301–303
    operation, 300
    services, 318
    zones, 317–318
netmasks for IP addresses, 257–258
netplan command, 282–284
netstat command
    example output, 287–288
    security implications, 343–344
    service tracking, 342–344
    system monitoring, 337
    working with, 343
network address translation (NAT), 301
    connection tracking, 302–303
    examples, 301–302
    iptables command, 320–321
    nftables command, 321
network configuration, 271
    AWS, 293–295
    Fedora installation, 28–30
    hostnames, 295–297
    modules and network interfaces, 271
    routers, 290–292
    routes, 284–289
    Ubuntu installation, 34
    utilities, 272–278
    VPCs, 293
Network File System (NFS), 497
    client configuration, 506–510
    components, 501–502
    enabling, 500–501
    mechanics, 497–498
    partitions, 499–500
    sample configuration files, 510–512
    security, 499
    server configuration, 503–506
    server installation, 17
    uses, 512
    versions, 498–499
network interface cards (NICs)
    configuring, 271–278
    Debian-like systems setup, 280–281
    Fedora, CentOS, and RHEL setup, 278–280
    netplan setup, 282–284
    systemd-networkd setup, 281–282
network security, 341
    attack handling, 348–349
    interface binding, 344–345
    service shutdown, 345–347
    service tracking, 342–344
    system monitoring, 347–348
    TCP/IP, 341–342
    tools, 349–351
network throughput in backups, 601
network user systems, 9–10
networkctl command, 282
NetworkManager (NM) package, 273
networks
    DFS, 532–534
    IP addresses, 255–256
    naming conventions, 356
    printers, 565
new-kernel-pkg script, 214
newaliases command, 427
news logging facility, 190
next-server parameter in DHCP, 578
NFS. *See* Network File System (NFS)
nfsvers command, 507
nft utility, 321
nftables tool, 300, 304, 307, 321
NICs. *See* network interface cards (NICs)
NM (NetworkManager) package, 273
nmap program, 350
nmbd daemon, 517
nmcli utility, 273–277, 279
no_access variable in /etc/xinetd.conf file, 185
noncommercial distributions, 4

nonhuman user accounts, 330
normal files, 56
normal users, 111
[NOTFOUND=action] entry, 385
NS (Name Server) DNS records, 371
NSA (National Security Agency), 336
nslcd daemon, 547
nslookup utility, 382
nss-pam-ldapd package, 547
nsupdate utility, 382
null passwords in Samba, 525–526

O

octal permissions for files, 60
octets in IP addresses, 255–256, 258
offline mode for mail, 431
online defragmentation in file systems, 158
online mode for mail, 431
only_from variable in /etc/xinetd.conf file, 184
oops messages, 213
Open Shortest Path First (OSPF) protocol, 260, 264
open source software
    advantages, 8–9
    disadvantages, 9
    overview, 5–7
Open Systems Interconnection (OSI) model, 238–241
OpenAFS system, 535
OpenBSD project, 482
opening TCP connections, 250–251
OpenLDAP, 544
    clients, 552–554, 557–558
    configuring, 546–554
    directories, 553–555
    installing, 545–546
    server-side daemons, 544
    user authentication, 555–558
    utilities, 544–545
OpenSSH, 482
    Apple systems, 483
    files used by, 493
    installing, 483–486
    scp program, 492–493
    secure tunnels, 489–491
    server startup and shutdown, 486–487
    sftp program, 493
    shell tricks, 492
    ssh client, 488–489
    Ubuntu installation, 36
openssl command, 443–444, 550
OpenStack project, 41–42
openSUSE
    GUI package manager, 98
    hostname configuration, 296
    RPM, 84
OpenVAS application, 350
OSI (Open Systems Interconnection) model, 238–241
OSPF (Open Shortest Path First) protocol, 260, 264
other configuration file for PAM modules, 131
out-of-order delivery in VoIP, 477
OUTPUT chains in Netfilter, 305–306
outside connections for PBX, 467
ownership
    files, 58–59
    HTTP processes, 405
    permissions, 59–60, 123

P

packages
    querying for, 87–88
    validating, 94–95
    verifying, 93–94
Packet Filtering option for kernel, 307
packet loss in VoIP, 477
Packet Mangling option for kernel, 308
packets
    headers, 241–250
    TCP/IP, 236–238
page description language (PDL), 560
pam_ldap library, 547
pam_nologin.so module, 130
pam_winbind module, 517, 527
PAMs. *See* Pluggable Authentication Modules (PAMs)
parameters, environment variables as, 53
paravirtualization, 585
parsing logs, 347
parted utility, 168–170
partitions
    creating, 26–27, 168
    GRUB 2, 139
    layout, 169–170
    names, 164–165
    NFS, 499–500
    overview, 165
    schemes, 24–25
passive FTP mode, 390
passwd command
    risk source, 327–328
    users, 132–133
passwords
    changing, 349
    Dovecot databases, 436
    encryption, 113
    groups, 117
    PAM modules, 127
    POP, 440
    root, 30
    Samba, 516–517, 525–526
    users, 113, 116, 132–133
patch program, 217
patches
    kernel, 4–5, 216
    kernel, downloading and applying, 216–217
    kernel, errors, 219
    kernel, release candidates, 218–219
    SELinux, 336
PBX. *See* Private Branch Exchange (PBX) configuration

pdbedit command, 525
PDL (page description language), 560
performance
    NFS, 508–509
    server design, 16–17
    tcpdump tool, 265–266
permissions
    files, 59
    NFS, 504, 509–510
    processes, 405
    SetUID programs, 326–328
    users and access, 123–125
phone sets in PBX, 464–466
Physical layer in TCP/IP, 238
physical volumes (PVs), 166–167
pico editor, 71
pine editor, 71, 427
ping program
    ICMP, 241
    risk source, 326
pipes, 51
pjsip commands in Asterisk, 476
pjsip.conf file, 453–456
Pluggable Authentication Modules (PAMs), 12–13, 125–126
    configuring, 127–128
    debugging, 131
    files, 126–127
    fixing configuration files, 131
    operation, 126
    Samba, 516
    sample configuration file, 128–130
plus signs (+) for permissions, 60
Pointer (PTR) DNS records, 372
POP. *See* Post Office Protocol (POP)
port variable in /etc/xinetd.conf file, 185
Portable Operating System Interface for UNIX (POSIX) compliance, 536
portmap manager, 498, 501–502, 506
ports
    FTP, 390
    HTTP, 405
    TCP/IP, 341–342
POSIX (Portable Operating System Interface for UNIX) compliance, 536
Post Office Protocol (POP), 429–431
    basics, 431
    checking, 439–440
    connectivity, 443–444
    Dovecot, 431–439
postfix check command, 426
Postfix server
    configuring, 423–426
    installing, 422–423
    running, 426–428
POSTROUTING chains in Netfilter, 306
PostScript files, 560
PowerShell client in SSH, 483
PREROUTING chains in Netfilter, 304–305
pretty hostnames, 296
primary servers in DNS, 361
primary zones in named.conf file, 367–368
printenv command, 50
printing, 559
    client-side tools, 570–571
    CUPS system. *See* Common UNIX Printing System (CUPS)
    environment variables, 50
    terminology, 559–560
priorities in log messages, 188–190
Private Branch Exchange (PBX) configuration, 460–461
    dialplans, 463–464
    local extensions, 461
    outside connections, 467
    phone sets, 464
    test scenario, 466
    trunking, 467–475
private IP addresses, 256
problems in source package builds, 108–109
/proc directory
    contents, 221–222
    editing files in, 222–223
    entries, 223–225
    settings and reports, 225–227
/proc/kcore file, 222
/proc/sys/fs/file-max entry, 227
/proc/sys/net/ipv4 file, 222
processes
    killing, 75–76
    listing, 73–75
    ownership in HTTP, 405
    unnecessary, 329
Procmail program, 429
.profile file, 115
profiles
    NetworkManager package, 273
    Ubuntu installation, 36
programs, scheduling, 197–199
project prerequisites in Fedora installation, 19
proto command, 507
protocol entry
    /etc/xinetd.conf file, 184
    Ethernet headers, 241–242
    IP headers, 245
    iptables command, 312
protocols option in Dovecot, 435
proxy configuration in Ubuntu installation, 34
ps command, 73–74, 328
ps auxww command, 337
pseudo-tty interface, 48
PTR (Pointer) DNS records, 372
pty interface, 48
public key cryptography, 479–481
pure acknowledgments in TCP connections, 252
PuTTY client in SSH, 482
pvcreate command, 169
pvdisplay command, 169–171
PVs (physical volumes), 166–167
pwd command, 335
PXE protocol in server installation, 18

Q

QEMU, 585–586
Quality of Service (QoS) in VoIP, 477
question marks (?) in filename expansion, 52
queues
    Postfix server, 427
    TCP/IP packets, 237
QUIT command in SMTP, 421

R

range declarations in DHCP, 577
RARP (Reverse ARP), 254
raw tables in Netfilter, 300
rawhide Fedora versions, 18
rccups command, 561
RCPT TO: command in SMTP, 420
Read permission for users, 123
reading dumpfiles, 265
README files, 104–105
Real-time Transport Control Protocol (RTCP), 450
Real-time Transport Protocol (RTP), 450
reboots with fsck tool, 151
record types in DNS, 369
    A and AAAA, 371
    CNAME, 373
    MX, 372–373
    NS, 371
    PTR, 372
    RP and TXT, 373
    SOA, 370–371
recovery mode in booting, 152
Red Hat Enterprise Linux (RHEL) series
    commercial release, 6
    hostname configuration, 296
    NFS, 500–501
    NIC setup, 278–280
    PAM modules, 126
    Postfix server installation, 422–423
    XFS file system, 159
Red Hat Package Manager (RPM)
    description, 83
    DHCP installation, 574
    GUI package managers, 97–98
    information queries, 86–88
    OpenSSH installation, 483–484
    overview, 84
    package validation, 94–95
    package verification, 93–94
    Samba installation, 518
    software installation, 89–92
    software uninstallation, 92–93
REDIRECT Target Support option for kernel, 307–308
redirection in Bash shell, 52
Reformat field in partition information, 26
Registration section in Asterisk, 456
Registry, Windows, 12
registry for Docker, 595
reject command for printer jobs, 569
REJECT Target Support option for kernel, 307
release candidates for kernel, 218–219
remote file access in Samba, 522–524
remote name daemon control (rndc) utility, 383
remote printers, adding, 564–565
remote procedure calls (RPCs), 498
remote shares in Samba, 524
removing
    files and directories, 66
    groups, 121, 134–135
    software, 92–93, 101–102
    users, 121, 134–135
renderers in netplan, 283
reparse points, 11
replication in DFS, 536
reports in proc directory, 225–227
Representational State Transfer (REST) interface, 42
request headers in HTTP, 404
reset (RST) flag in TCP connections, 252
resolvconf program, 384, 386
resolvectl tool, 381
resolver in DNS, 383–385
resources, limited, 330–332
REST (Representational State Transfer) interface, 42
restore tool, 605, 607–608
retrans command, 507–508
return on investment (ROI) in virtualization, 583
Reverse ARP (RARP), 254
reverse resolution in DNS, 360
RHEL. *See* Red Hat Enterprise Linux (RHEL) series
Ring 0, 10
RIP (Routing Information Protocol), 260–264
risks in local security
    mitigating, 332–336
    sources, 326–329
rm command
    files and directories, 66
    package installation, 107–108
rndc-confgen tool, 383
rndc (remote name daemon control) utility, 383
ROI (return on investment) in virtualization, 583
root directory, 24
root domains, 357
root elements in LDAP, 542
root file system (/), 160
root-level pages in Apache web server, 409
root name servers, 357, 361
root user passwords, 30
route command, 285–288
router configuration, 290–292
routes
    ARP, 254
    configuring, 285–286
    displaying, 287
    network configuration, 284–285
Routing Information Protocol (RIP), 260–264
routing IP addresses, 258–264
routing tables, 259
RP DNS records, 373
rpc commands, 502
rpcbind service, 500–501
rpcinfo command, 500–501

RPCs (remote procedure calls), 498
RPM. *See* Red Hat Package Manager (RPM)
rsize command in NFS, 507, 509
RST (reset) flag in TCP connections, 252
rsync utility, 609
rsyslog package, 187–188
    /etc/rsyslog.conf file, 189–191
    log message classifications, 188–189
    rsyslog daemon, 188
    systemd-journald component, 195–197
    templates, 191–194
RTCP (Real-time Transport Control Protocol), 450
RTP (Real-time Transport Protocol), 450
rules
    /etc/rsyslog.conf file, 192–194
    iptables command, 312–317
run-instances command, 295
/run/systemd/network directory, 281
runlevels, 145, 330

S

safety in servers, 16–17
Samba, 515
    daemons, 517
    installing, 518
    passwords, 516–517, 525–526
    remote file access, 522–524
    remote shares, 524
    servers, 522
    shares, 519–524
    starting and stopping, 519
    troubleshooting, 529
    usernames and passwords, 516–517
    users, 524–525
    Windows server authentication, 526–529
saving Netfilter configuration, 308–309
SCCP (Skinny Client Control Protocol), 449
scheduling programs, 197–199
schemas in LDAP, 544
SCM (Source Code Management) system, 219
scp (Secure Copy) program, 492–493
script kiddies, 325
ScriptAlias option in Apache web server, 415
scripts
    Samba, 516
    startup, 114–115
    systemd, 145–149
scsidev program, 604
searching LDAP directories, 554–555
sec command, 507
second-level domains, 358
secondary DNS servers, 361
secondary zones in named.conf file, 368
sections in Asterisk, 453–456
Secure Copy (scp) program, 492–493
Secure FTP (sftp) program, 493
Secure Shell (SSH), 479
    clients, 482–483
    OpenSSH installation, 483–486
    OpenSSH operation, 488–493
    public key cryptography, 479–481
    SSHD configuration file, 487–488
    tunneling, 345
    Ubuntu installation, 36
    versions, 481–483
Secure Sockets Layer (SSL)
    Dovecot parameters, 438
    mail services, 442–444
    TCP/IP layer, 240–241
secure tunnels in OpenSSH, 489–491
security
    groups, 294–295
    HTTP, 405
    kernel, 4–5, 502–503
    local. *See* local security
    mail services, 442–444
    network. *See* network security
    NFS, 499
    OpenLDAP, 558
    processes, 76
    servers, 16–17
    SMTP, 421
Security entry in vsftpd, 393
security logging facility, 190
security tables in Netfilter, 300
sed editor, 71–72
selector field in rsyslog.conf, 193
SELinux
    Asterisk, 478
    security, 335–336
semicolons (;) in multiple commands, 53
server_args variable in /etc/xinetd.conf file, 184
server cloud deployment, 39
    commercial cloud service providers, 41–43
    free-to-run virtual Linux servers, 40–41
server-identifier parameter in DHCP, 578
server installation
    DNS, 362–367
    Fedora. *See* Fedora installation
    hardware and environmental considerations, 15–16
    methods, 17–18
    overview, 15
    server design, 16–17
    summary, 37
    Ubuntu. *See* Ubuntu installation
Server Message Block/Common Internet File System (SMB/CIFS) protocol, 515
Server Message Block (SMB)
    description, 565
    server installation, 18
server-name parameter in DHCP, 578
server-side daemons in OpenLDAP, 544
server statement
    /etc/xinetd.conf file, 184
    named.conf file, 366
ServerAdmin option in Apache web server, 411
ServerName option in Apache web server, 411
ServerRoot option in Apache web server, 410
servers
    DHCP, 574–581
    DNS, configuring, 367–369

servers (*cont.*)
DNS, installing, 362–367
DNS, setup, 374–379
DNS, types, 361–362
high-volume, 227
NFS, 503–506
OpenLDAP, 555–556
Samba, 522
unnecessary processes on, 329
VoIP, 447–448
service command, 347
service option in Dovecot, 435
service units in systemd manager, 178
service utility in Apache web server, 407
services
disabling, 150–151
enabling, 150
Netfilter firewall, 318
shutting down, 345–347
tracking, 342–344
Session Initiation Protocol (SIP)
Asterisk, 453–456
description, 449
VoIP, 460–461
session modules, 127
SetGID program
finding, 328
user permissions, 123–124
setting environment variables, 50
SetUID program
finding and creating, 327–328
overview, 326–327
user permissions, 123–124
sfdisk utility, 168
sftp (Secure FTP) program, 493
shared mail spools in NFS, 512
shared-network declarations in DHCP, 576
shares in Samba, 519–524
Shell field for users, 116
shell tricks in OpenSSH, 492
shortcuts, command line, 52–54
showmount command, 505
shutdown
Apache web server, 407–408
OpenSSH, 487
services, 345–347
signaling protocols in VoIP, 449
signals for processes, 75–76
SIGTERM signal, 76
Simple Mail Transfer Protocol (SMTP), 419–420
e-mail components, 421–422
overview, 420–421
Postfix server, configuring, 423–426
Postfix server, installing, 422–423
Postfix server, running, 426–428
security, 421
troubleshooting, 428
single-user mode, booting into, 152
single-user systems, 9–10
SIP (Session Initiation Protocol)
Asterisk, 453–456
description, 449
VoIP, 460–461
size of files, 158
Skinny Client Control Protocol (SCCP), 449
slapd.conf file, 548–551
slapd daemon
configuring, 548–552
OpenLDAP, 544, 546
slaptest command, 551
slash (/) partition, 160
SLE (SUSE Linux Enterprise)
GUI package manager, 98
hostname configuration, 296
sliding windows in TCP headers, 248
slurpd daemon, 544
SMB/CIFS (Server Message Block/Common Internet File System) protocol, 515
SMB (Server Message Block)
description, 565
server installation, 18
smbclient program, 521–523, 525
smbd daemon, 517
smbpasswd program, 525–526
SMTP. *See* Simple Mail Transfer Protocol (SMTP)
smtpd_banner option in Postfix, 425
snap apps
installing, 108
Ubuntu installation, 37
snapd program, 108
snapshot units, 178
SNAT (Source NAT), 301
Snort program, 350
SOA (Start of Authority) DNS records, 370–371
Socket entry in vsftpd, 393
socket_type variable in /etc/xinetd.conf file, 184
sockets in systemd manager, 177–178
soft mounts in NFS, 507–508
softphones
description, 448
PBX, 464–466
software
DPMS, 99
Fedora installation settings, 21–22
GNU, 103–108
installing, 89–92, 100–101
testing, 107
Ubuntu, 99–102
uninstalling, 92–93, 101–102
software management, 83
DNF, 96–97
RPM. *See* Red Hat Package Manager (RPM)
Yum, 96
sort command, 73
source addresses
Ethernet headers, 241–242
IP headers, 245–246
source code
broken, 109
kernel, 203–204
managing, 83
Source Code Management (SCM) system, 219

Source NAT (SNAT), 301
source packages
    build problems, 108–109
    cleaning up, 107–108
    compiling, 106
    configuring, 106
    documentation, 104–105
    Fedora installation, 22
    getting and unpacking, 103–104
    installing, 106–107
source ports in TCP/IP, 342
sources.list file, 99, 101
speed of data recovery in backups, 601–602
spins, Fedora, 22
spoolers, printing, 560
SSH. *See* Secure Shell (SSH)
ssh client, 488–489
sshd_config file, 487–488, 493
sshd service, 486–487
ssl option in Dovecot, 435, 438
ssl_cert option in Dovecot, 435, 438
ssl_key option in Dovecot, 435, 438
SSL (Secure Sockets Layer)
    Dovecot parameters, 438
    mail services, 442–444
    TCP/IP layer, 240–241
sssd daemon, 547
stability in server design, 16–17
stable point kernel releases, 203
stacks in TCP/IP, 235
stale file handles in NFS, 509
Stallman, Richard Matthew, 5–7
standards, 6–7
Start of Authority (SOA) DNS records, 370–371
starting up Apache web server, 407–408
startup scripts, 114–115
stat utility, 327
state module for iptables, 315
stateful connection tracking in NAT, 303
statement keywords in named.conf file, 365
static hostnames, 296
static routes
    IP addresses, 258–260
    routing with, 290–292
status command, 181
sticky bits in permissions, 125
storage pools in GlusterFS, 538
storing log entries, 347–348
su command, 77–78
subdomains, 358–359
subnets
    DHCP declarations, 576–577
    IP addresses, 256–257
    VPCs, 293, 295
sudo command, 78
superblocks in file systems, 157
superusers, 111
SUSE Linux Enterprise (SLE)
    GUI package manager, 98
    hostname configuration, 296
swap space, 25
switching users, 77–78
symbolic links, 57
SYN flags in TCP connections, 250–252
SYN flood protection, 226
synaptic tool, 102
synchronous replication, 536
syncookies, 226
/sys/devices directory, 228
sysctl tool
    hostname configuration, 296
    Netfilter, 309
    /proc directory, 225–226
    SYN flood protection, 226
SysFS file system, 227–228
sysklogd daemon, 187
syslog logging facility, 190, 347–348
syslogd daemon, 187–188
system calls, 221
system-config-users utility, 122
system console, entering commands at, 85
System.map file, 213
system settings in Fedora installation, 22–30
system users in Dovecot, 436
systemctl command
    Apache web server, 407–408
    Asterisk, 452
    CUPS, 561
    DHCP, 581
    Dovecot, 439
    graphical environments, 85
    KVM, 588
    LDAP, 551–552
    NFS, 500–501
    OpenSSH, 486–487
    Postfix server, 422, 426
    runlevels, 330
    Samba, 528
    service shutdown, 346
systemd-journald component, 188, 195–197
systemd manager
    aggressive parallelization, 176
    control groups, 176–177
    cron program, 199
    human digestive system comparison, 177
    legacy notes, 179–181
    overview, 175–176
    role, 176
    scripts, 145–149
    socket activation, 177
    /tmp file system, 231
    units, 178–179
systemd-networkd command, 278, 281–283
systemd-resolved manager, 384
systemd.timer object, 199
systems
    details, 77
    monitoring, 337–338, 347–348
sysv-rc-conf command, 346

T

tac command, 69
tail command, 416–417
tape archives, 66–68
tape backups, 603–605
tar tool
    archives, 66–68
    backups, 608
    source packages, 103–104
tarballs, 103
targets
    chains, 310, 313
    makefiles, 205–206
    systemd manager, 178
tasks, backgrounding, 48–49
TCP headers, 246–249
TCP/IP (Transmission Control Protocol/Internet Protocol), 235. *See also* IP addresses
    ARP, 253–255
    connections, 250–253
    dynamic routing, 260–264
    headers, 241–250
    hosts and networks, 255–256
    layers, 235–241
    netmasks, 257–258
    network security, 341–342
    OSI model, 238–241
    packets, 236–238
    port numbers, 341–342
    static routing, 258–260
    subnetting, 256–257
    tcpdump tool, 265–267
TCP layer in TCP/IP, 239–240
tcp module for iptables, 316
tcpdump tool, 241–243
    ARP, 254
    DNS issues, 266–267
    dumpfiles, 265
    IP headers, 245–246
    network security, 350–351
    own network traffic, 266
    packet capturing, 265
    performance impact, 265–266
    TCP connections, 250–252
    TCP headers, 246–248
    UDP headers, 249
.tcshrc file, 115
tdbsam database, 524–525
tee command, 174
telinit command, 180
Telnet program
    IMAP, 440
    POP, 439
    security, 442
    SMTP, 420
templates in rsyslogd, 191–194
temporary files in package installation, 107–108
terminal interfaces in job control, 48
testing
    Apache web server, 408–409
    PBX, 466
    PSTN, 474–475
    software, 107
texinfo system, 56
text execution with backticks, 53–54
text files vs. Registry, 12
TFTP (Trivial File Transfer Protocol) service, 18
third-level domains, 358
three-way handshakes in TCP connections, 250
time settings
    Fedora installation, 21
    UNIX epoch, 117
time-to-live (TTL) field in IP headers, 244–245
timer object, 199
timer units in systemd manager, 178
TLDs (top-level domains), 357
TLS (Transport Layer Security)
    mail services, 442–444
    TCP/IP layer, 240–241
/tmp file system, 231
/tmp partition, 25
tmpfs file system, 231
top command, 75
top-level domains (TLDs), 357
Torvalds, Linus, 8
total length field in IP headers, 244
touch command, 62, 66
tracking services, 342–344
transient hostnames, 296
Transmission Control Protocol/Internet Protocol. *See* TCP/IP (Transmission Control Protocol/Internet Protocol)
Transport Layer Security (TLS)
    mail services, 442–444
    TCP/IP layer, 240–241
Transport section in Asterisk, 454
Trivial File Transfer Protocol (TFTP) service, 18
troubleshooting
    Apache web server, 416–417
    Asterisk, 475–478
    DNS, 266–267
    NFS, 506, 509–510
    Samba, 529
    SMTP, 428
trunking in PBX, 467–475
trust in network security, 349
trusted storage pools in GlusterFS, 538
$TTL entry in BIND database files, 374
TTL (time-to-live) field in IP headers, 244–245
tunnels
    IPv6, 270
    OpenSSH, 489–491
    SSH, 345
Twilio Elastic SIP Trunks, 467–475
TXT DNS records, 373
type variable in /etc/xinetd.conf file, 184

U

Ubuntu, 99
    DHCP installation, 575
    GUI package manager, 97–98, 102

hostname configuration, 296
information queries, 100
KVM setup, 591–593
NFS, 501
OpenLDAP installation, 546
OpenSSH installation, 484–485
Postfix server installation, 423
software installation, 100–101
software uninstallation, 101–102
Ubuntu installation, 32–33
file system setup, 34–36
network configuration, 34
profile setup, 36
proxy configuration, 34
snaps, 37
SSH setup, 36
starting, 33–34
udev system, 227
udp module for iptables, 316–317
UDP (User Datagram Protocol)
description, 240
headers, 249–250
UDP (User Datagram Protocol) layer in TCP/IP, 239–240
UEFI (Unified Extensible Firmware Interface), 24
UIDs (user IDs), 111
NFS, 510
users, 113–114
ulimit facility, 330–332
umount command, 161–162
uname command, 77, 207, 214
unicast addresses in IPv6, 269
Unified Extensible Firmware Interface (UEFI), 24
Uniform Resource Information (URI) for printers, 564
uninstalling software, 92–93, 101–102
units in systemd manager, 178–179
UNIX epoch, 117
unmounting local disks, 159–163
unnecessary processes, 329
unpacking
kernel source code, 204
source packages, 103–104
unset command, 51
unsetting environment variables, 51
upgrades, kernel, 205
upstart daemon, 179–181
upstream components, 6–7
uptime
kernel, 202
servers, 17
urgent pointers in TCP headers, 249
URI (Uniform Resource Information) for printers, 564
USB boot disks in GRUB 2, 143–144
use-lease-addr-for-default-route parameter in DHCP, 579
user accounts
creating, 30–31
nonhuman, 330
user authentication in OpenLDAP, 555–558
User Datagram Protocol (UDP)
description, 240
headers, 249–250
User Datagram Protocol (UDP) layer in TCP/IP, 239–240
user IDs (UIDs), 111
NFS, 510
users, 113–114
user logging facility, 190
User option in Apache web server, 412
user variable in /etc/xinetd.conf file, 184
useradd command, 118–120, 132–133
userdel command, 121, 134–135
UserDir option in Apache web server, 413–414
usermod command, 120, 133–134
Username field, 113
usernames in Samba, 516
users
adding, 118–120, 132–133
characteristics, 111–112
command-line management, 118–122
Dovecot, 436
/etc/group file, 117–118
/etc/passwd file, 112–116
/etc/shadow file, 116–117
GUI managers, 122–123
information on, 112–118
modifying, 120, 133–134
moving, 78–80
PAM modules, 125–131
passwords, 132–133
permissions, 123–125
removing, 121, 134–135
Samba, 524–525
shells, 116
switching, 77–78
/usr/bin/dig tool, 364
/usr/bin/host tool, 364
/usr/lib/apache2/modules directory, 412
/usr/lib/firewalld file, 317
/usr/lib/systemd/system file, 145–146, 178
/usr/lib64/asterisk/modules directory, 458
/usr/local/src directory, 103
/usr partition, 25
/usr/sbin/named tool, 364
/usr/sbin/rndc tool, 364
/usr/share/doc/initscripts/sysconfig.txt file, 279
/usr/share/httpd/noindex/index.html file, 409
/usr/src directory, 204
utilization information for disks, 72–73
uucp logging facility, 190

V

validating packages, 94–95
vanilla kernels, 203
/var/lib/samba/private/passdb.tdb file, 525
/var/log directory, 337
/var/log/apache2 directory, 416
/var/log/httpd directory, 416
/var/log/mail file, 428
/var/log/maillog file, 427–428
/var/log/messages file, 366
/var/named/example.org.db file, 367
/var partition, 25
/var/spool/mail directory, 425

verbs in HTTP, 404
verifying packages, 93–94
versions
    IP headers, 243
    kernel, 203–204
vertical bar (|) character for pipes, 51
Very Secure FTP Daemon (vsftpd) program and package, 389
    configuring, 391–392
    FTP server, customizing, 395–401
    FTP server, starting and testing, 392–395
    FTP server, virtual user setup, 397–401
    obtaining, 391
vgcreate command, 169
vgdisplay command, 169, 171
vgextend command, 169
VGs (volume groups), 166–167
vi editor, 70
vim editor, 70
virsh utility, 588, 590
virt-builder utility, 40–41
virt-install utility, 587–590
virt-manager application, 587
Virtual Machine Manager, 587
virtual machines (VMs), 39
virtual private clouds (VPCs), 289, 293
virtual users in FTP server, 397–401
VirtualBox platform, 586
VirtualHost option in Apache web server, 415–416
virtualization, 583
    concepts, 584–585
    containers, 594–598
    implementations, 585–586
    KVM, 586–593
    purpose, 583–584
VMs (virtual machines), 39
VMware, 586
Voice over Internet Protocol (VoIP)
    Asterisk, 451–459
    ATA, 448
    dialplans, 457
    implementations, 450–451
    IP phones, 448
    modules, 458
    overview, 447
    PBX. *See* Private Branch Exchange (PBX) configuration
    protocols, 448–450
    servers, 447–448
Volume Group field in partition information, 26
volume groups (VGs), 166–167
volumes
    creating, 171–172
    exploring, 169
    groups, 170
    overview, 166–167
    partitions and logical volumes, 168–172
VPCs (virtual private clouds), 289, 293
vsftpd. *See* Very Secure FTP Daemon (vsftpd) program and package

W

w command, 77
wait variable in /etc/xinetd.conf file, 184
WANs (wide area networks), 239
wbinfo utility, 529
Web interface
    printer installation, 565–567
    printer management, 569–570
well-known TCP/IP ports, 342
wget utility, 103, 203
whereis command, 69
which command, 69
whitelists for Twilio Elastic SIP Trunks, 471
who command, 77
whois command, 382
wide area networks (WANs), 239
wildcards in filename expansion, 52–53
winbind package, 527
winbindd daemon, 517, 526–527
Windows Registry system, 12
Windows server, Samba authentication for, 526–529
Windows vs. Linux, 9–13
Wireshark tool, 350–351
working directories, 66
world permissions, 59–60
Write permission, 123
writing dumpfiles, 265
wsize command, 507, 509

X

X-Lite softphones, 464–466
X Window System, 10
.Xdefaults file, 115
Xen virtualization, 586
XFCE window managers, 11
XFS file system, 159
xinetd program, 181–182
    echo service example, 186–187
    /etc/xinetd.conf file, 182–185
    shutting down, 346
.xinitrc file, 115
XML (Extensible Markup Language), 240–241
XMPP (Extensible Messaging and Presence Protocol), 449
xterm emulator, 86
xz tool, 65

Y

YaST (Yet Another Setup Tool), 98
yast2 command, 172
Yum tool, 96

Z

zombie reapers, 180
zones
    named.conf file, 367–369
    Netfilter firewall, 317–318