# TLS Mastery

Michael W Lucas

# TLS Mastery

Michael W Lucas

## Copyright Information



Author: Michael W Lucas
Copyeditor: Amanda Robinson
Cover art: Eddie Sharam

ISBN (Beastie edition): 978-1-64235-052-4
ISBN (Tux edition): 978-1-64235-053-1
ISBN (hardcover): 978-1-64235-051-7



Tilted Windmill Press
https://www.tiltedwindmillpress.com

# TLS Mastery

Michael W Lucas

## More Tech Books from Michael W Lucas

Absolute BSD

Absolute OpenBSD (1st and 2nd edition)

Cisco Routers for the Desperate (1st and 2nd edition)

PGP and GPG

Absolute FreeBSD (2nd and 3rd edition)

Network Flow Analysis

## the IT Mastery Series

SSH Mastery (1st and 2nd edition)

DNSSEC Mastery

Sudo Mastery (1st and 2nd edition)

FreeBSD Mastery: Storage Essentials

Networking for Systems Administrators

Tarsnap Mastery

FreeBSD Mastery: ZFS

FreeBSD Mastery: Specialty Filesystems

FreeBSD Mastery: Advanced ZFS

PAM Mastery

Relayd and Httpd Mastery

Ed Mastery

FreeBSD Mastery: Jails

SNMP Mastery

TLS Mastery

The Networknomicon

## Other Nonfiction

Cash Flow For Creators

Only Footnotes

## Books and Novels (as Michael Warren Lucas)

Immortal Clay

Kipuka Blues

Butterfly Stomp Waltz

Terrapin Sky Tango

Forever Falls

Hydrogen Sleets

Drinking Heavy Water

Aidan Redding Against the Universes

git commit murder

git sync murder

See your local bookstore for more!

## Brief Contents

## Complete Contents

# Acknowledgements

TLS is perhaps the most complicated topic I've ever written about. Writing this book would have been impossible without outside help.

This book would not exist if the Internet Security Research Group hadn't deployed ACME and organized Let's Encrypt. TLS certificates are not only free for most people, their maintenance and renewal is highly automatable. They've changed the whole Internet, and deserve our thanks for that.

It doesn't matter how many RFCs I study and how many technical mailing list archives I read: I lack the expertise and context to best illuminate an arcane topic like TLS. The folks who read this manuscript's early stages and pointed out my innumerable errors deserve special thanks. James Allen, Xavier Belanger, Trix Farrar, Loganaden Velvindron, Jan-Piet Mens, Mike O'Connor, Fred Schlechter, Grant Taylor, Gordon Tetlow, and Fraser Tweedale, here's to you.

Lilith Saintcrow convinced me that *The Princess Bride* could be a useful motif for a serious technology book. This book was written during the 2020 pandemic, so I must also thank *The Princess Bride* for providing me a desperately needed sense of hope.

Dan Langille gracefully submitted to the pillaging of his blog for useful hints and guidance. I am grateful that JP Mens, Evan Hunt, and John-Mark Gurney provoked him into updating that blog and saving me a bunch of work.

I am unsure if I should profusely thank Bob Beck for his time and patience in revealing the innards of TLS, or profoundly curse him and his spawn unto the seventh generation. I must acknowledge the usefulness of "Happy Bob's Test CA," however, so I'll raise a glass to that while waffling over whether or not the bottle of fair-to-middlin' wine I owe him should be laced with iocane powder.

For Liz.

# Chapter 0: Introduction

Of the innumerable things I detest about information technology, first prize goes to the word "security." Not the concepts behind it, the actual word. The definition of "security" wobbles drunkenly all about the dictionary depending on who's speaking, who's listening, the context, and the distance to the nearest brute squad. It's a transcendental state where everyone is perfectly safe from everyone, but it's not inconvenient or intimidating or incomprehensible in the slightest. Security is Happy Fun Land, where everybody eats hot fudge sundaes all day every day without developing diabetes or gaining so much as a gram.

The only way to make this word even slightly meaningful is to tightly define the context.

That's one advantage Transport Layer Security (TLS) has. What it secures is right in the name. And even then, it's misunderstood. It doesn't make web servers secure. That little shield icon in the web browser's address bar doesn't mean your credit card information won't end up being used to purchase llama pornography. TLS encrypts a network connection during transit. That's it. It doesn't protect the client or the server from attackers. It doesn't keep scammers from tricking you out of your personal data. It doesn't even totally guarantee that you're at the site you think you're at. Protecting data in transit is vital. While it's best known for web sites, a TLS-aware application can apply TLS to any TCP or UDP network connection.

TLS is also poorly understood. Most sysadmins know that they get a certificate, slap it into place, and Magic Happens. Those certificates used to be expensive. Over the last twenty years the price dropped, and today you can get them for free. There are still times you want one of the expensive certificates, but most of us have no idea when or why that expense is warranted.

Even with free certificates, I'm still not fond of TLS. This certainly isn't one of those books where the author is so besotted by the technology that you wonder if it's going to turn into a kissing book. But TLS is pervasive, frustrating, and complex. Understanding is our only way to cope with it.

## *Who Should Read This Book?*

*TLS Mastery* is written for Unix system administrators who manage applications built with TLS, and anyone who uses the OpenSSL command on any platform. I assume you're comfortable with the command line, scripting, privilege management, and other standard Unix features.

My reference platforms are FreeBSD, OpenBSD, Debian, and CentOS. The closer your Unix resembles one of these, the easier time you'll have. If you run a less common Unix, presumably you're familiar with its idiosyncrasies. In particular, MacOS ships a stripped-down OpenSSL client lacking many of the functions discussed here. For real work on MacOS you probably need an add-on alternate OpenSSL.

Among the many ACME implementations, this book uses dehydrated (`https://dehydrated.io`). The principles demonstrated with dehydrated should apply to any other client. I use Apache 2.4 to show how certain dehydrated components work, but other web servers work just as well. For DNS-related examples I use BIND 9.16, but any name server that supports dynamic updates (RFC 2136) will also work.

My reference TLS toolkit is OpenSSL, version 1.1.1. I also use LibreSSL, OpenBSD's meticulously audited OpenSSL fork, but it retains compatibility with the OpenSSL command line. Anything referring to OpenSSL also applies to LibreSSL unless stated otherwise. The principles discussed are also applicable to other TLS toolkits like GnuTLS, but I don't demonstrate them. If you can build a functional OpenSSL or LibreSSL on your platform, it should work.

OpenSSL is not only for TLS; it is a general-purpose encryption suite. Its command line is convoluted and complex in part because encryption is convoluted and complex. It's also complex because it originated in 1995 and attempts to retain backwards compatibility. I can't make you comfortable with the OpenSSL command line, but I might be able to reduce the amount of vertigo you experience when interacting with it.

Might.

### *TLS, SSL, and Versions*

You hear about SSL connections and certificates, and TLS connections and certificates. What's the difference?

A *digital certificate* is a collection of carefully formatted information that identifies an entity, digitally signed by a Certificate Authority. A certificate signed by itself is called a *self-signed certificate*, and is the Internet equivalent of the handsome prince that smiles and says, "Trust me." Maybe you can trust him, or maybe you've already been betrayed. Servers, services, and users can have certificates. We go into certificates in depth in Chapter 3. Certificates are a key component of both SSL and TLS.

*Secure Sockets Layer*, or *SSL*, was an early transport layer security protocol. The Netscape Corporation wanted the ability to encrypt traffic between web servers and their spiffy new graphical browser, so in 1994 they created the primordial SSL and let a small group of people test it. It sort of worked and it let the testers experiment with ecommerce, but as with any protocol designed by a single institution it had numerous flaws. In 1995, Netscape hurriedly released the slightly more robust SSL version 2, followed by version 3 in 1996. Despite this quick succession of versions, SSL's core cryptographic design was intrinsically and irreparably flawed.

The IETF released version 1 of *Transport Layer Security*, or *TLS*, in 1999. It's a direct descendant of SSL version 3, but the name was changed for political reasons. TLS 1.1 escaped in 2006, 1.2 in 2008, and 1.3 in 2018.

SSL version 2 was completely obsoleted in 2011, and version 3 in 2015. No version of SSL should be used on today's Internet. Similarly, TLS 1.0 and 1.1 were increasingly discouraged starting around 2010, and nearly complete deprecation occurred in 2018. These protocol versions are actively dangerous and must not be used, as discussed in Chapter 1.

As of 2020, all Internet sites should use prefer TLS versions 1.3, falling back to 1.2 only if necessary. In early 2021, the NSA and the security bodies of several other governments strongly recommended abandoning TLS 1.2 as well.

If SSL is no longer a live protocol, and hasn't been in use for years, why do we keep hearing about it? Language moves more slowly than technology. Even sysadmins who only run TLS keep calling it SSL. Users have picked up that acronym, and once a user thinks they understand something they detest updating their knowledge. Those who point out that it's TLS, not SSL, get dismissed as pedants and lose friends. Plus, the most widely used TLS software toolkit includes SSL in the name. We're stuck with those letters, if not the technology.

I will not mention SSL again unless I'm specifically referring to the ancient, forbidden protocol.

### *Why TLS?*

The Internet has a whole market square of secure transport protocols. IPSec. Wireguard. OpenVPN. Some have been abandoned.[1] Protocols have had mergers, devolved into factions, and darn near fought with swords. What makes TLS special, and why has it survived?

TLS is a generic protocol for wrapping individual TCP/IP connections. Where solutions like IPSec can tunnel and encrypt all traffic between two IP addresses, TLS encrypts only a single connection.

1 I'm sorry, SKIP. The Internet has declared we shall not have a Happily Ever After.

TLS can be added to existing protocols comparatively simply. Netscape wanted existing web sites to be able to migrate to confidential and tamper-proof transport without reworking the HTTP protocol. They made SSL as unobtrusive as possible, and TLS still prioritizes that feature.

Finally, developers can manage TLS entirely inside their applications. There's no need to negotiate with the host's IPSec or OpenVPN features. Developers don't have to play games with routing or networking or any of that scary stuff. If you're porting software from Unix to Linux, or Windows, or MacOS, or whatever, transport security is not the hardest part of that effort.

Finally, TLS is somewhat opportunistic. A client can check a service for TLS, and use it if it's available. The client requires no special setup, unless you're using more complicated features like client certificate authentication.

### *Using openssl(1)*

Unix traditionally consists of a bunch of small programs, each of which handles a single simple task. OpenSSL (and forks like LibreSSL) take a different approach with openssl(1). The openssl(1) command is a general purpose tool for all things cryptographic and many things tangential. It can create keypairs, digitally sign files, parse ASN.1 and X.509, generate randomish numbers, and verify TLS certificates.

Cryptography is notoriously confusing, for the same reason that operator algebra, multivariable differential equations, and cosmology are notoriously confusing. Cryptography is legitimately difficult, and as a rule most sysadmins don't understand it.[2] Worse, many claim that they do. Mastering cryptography demands a lifetime. It's a comparatively inexpensive hobby that will stretch your brain, and takes up a lot less room than building model rockets every weekend.

2 Whenever I hear a group of sysadmins arguing cryptography, my first reaction is that they're like a group of paramedics debating the best methods for performing engineered-virus-assisted telemicroneurosurgery.

Aside from any design decisions on the part of the developers, OpenSSL inherits cryptography's complexity.

An `openssl` command takes the form:

```
$ openssl subcommand flags
```

The subcommand (often just called a command) defines which cryptographic functions you're working with. The subcommand `genrsa` creates RSA keys, while `x509` copes with X.509 certificates. Perusing the openssl(1) manual pages makes understanding all these operations seem like climbing the Cliffs of Insanity, but we'll make it as painless as possible.

Many subcommands share common flags for similar functions. You'll see `-in` and `-out` flags for input and output, `-text` for textual format, and so on. Where most Unix commands use single-character flags and allow stacking them right up against each other, such as `tar -czvf`, OpenSSL requires you enter each flag separately. I don't know that replacing `-in` and `-out` with, say, `-i` and `-o` would do much to improve the legibility of these commands. Cryptographic operations are inherently complex, and the added legibility is arguably a net win.

To see which version of OpenSSL you have, run `openssl version`.

```
$ openssl version
OpenSSL 1.1.1c FIPS  28 May 2019
```

To get much more detail about how your Unix packages and configures OpenSSL, use `openssl version -a`.

Many OpenSSL commands are designed to feed into one another, like so.

```
$ openssl s_client -showcerts -connect www.mwl.io:443 \
  </dev/null | openssl x509 -text -noout
```

The `openssl s_client` command serves as a TLS-aware netcat, negotiating a TLS connection with the host and port you specify.

This command normally waits for input, but we feed it `/dev/null` so that it doesn't wait. The `-showcerts` option displays the certificate information, and `-connect` lets you choose your target. We pipe this into `openssl x509`, the X.509 parser, specify that we want human-friendly output with `-text`, and skip showing the encoded certificate with `-noout`.

This combination grabs a web site's TLS certificate and displays the contents. It's the Unix equivalent of clicking on the lock icon in the browser's address bar and navigating a few layers of menu to find "Show Certificate."

### *The OpenSSL Manual*

OpenSSL is a giant. The manual is correspondingly gigantic. Older versions of OpenSSL put all the documentation for the command in a single giant manual page, openssl(1). Newer versions of OpenSSL split the manual into a few dozen smaller manual pages, one for each command. The documentation for `openssl version` appears in openssl-version(1), `openssl pkeyparam` is documented in openssl-pkeyparam(1), and so on. Many times, these smaller man pages are indexed by subcommand as well as the full name; you could type `man pkeyparam` instead of `man openssl-pkeyparam`. (Some Linux systems do not provide the openssl-prefixed version of the man pages; they're only available as `man x509`, `man pkeyparam`, and so on.)

Which does your system use? Run `man openssl-version` and see if the page exists.

Which is a better system? I have my bias, but any choice for arranging this much documentation will annoy someone. The ideal solution is to get promoted so you can delegate all TLS problems to someone else.

This book follows the newer OpenSSL standard of separate manual pages for each subcommand.

### *The United States and FIPS*

Many organizations in Canada and the United States must comply with the US *Federal Information Processing Standards* (FIPS), a set of cryptography requirements. The current standard is FIPS 140-3, but many organizations are still back on 140-2. It's often called "FIPS 140." FIPS dictates which cryptographic algorithms organizations may use, how data must be handled, how software is tested, and which implementations may be used.

Organizations bound by FIPS suffer severe penalties for violating that standard. If you work for such an organization, FIPS compliance overrides any of my advice. If you are the reason your employer loses all its government contracts, expect to be unceremoniously dragged before the Board of Directors.

The phrase "FIPS compliance" inspires vendors and organizations to new heights of weasel words. They might claim that their solution is FIPS compliant when they mean their developers read the standard and implemented those algorithms. The solution might look like FIPS, it might interoperate with FIPS, but it has not been audited and verified and approved. You can find a list of approved cryptographic engines online at **`https://csrc.nist.gov/projects/cryptographic-module-validation-program/validated-modules/search/all`**.

OpenSSL can be built to FIPS standards. Each individual software build must be tested for compliance. The operating system also requires specific FIPS configuration, but a FIPS-compliant software build clears one of the big hurdles. Red Hat ships a FIPS-approved OpenSSL, as well as a GnuTLS and kernel cryptographic engine.

FIPS restricts the algorithms your TLS-protected services can use. Yes, cryptography advances faster than government standards, and you might find yourself choosing between sub-optimal algorithms. Maybe the IETF or OpenSSL or whoever rolls out a snazzy new set of

algorithms, but if it's not on FIPS' approved list you cannot use it.

FIPS-compliant sites can choose to not use all permitted algorithms. As I write this, SHA-1 is on FIPS' permitted list. SHA-1 has been considered a dangerously feeble digital signature algorithm for years, and recent events have added it to the List of Things Sysadmins Need To Repeatedly Stab And Bury In A Ditch As Soon As Possible. (It's okay for certain uses, but if you are unsure which those are, drop it.) Disabling SHA-1 in your TLS configurations will not violate FIPS. Enabling stronger but unapproved algorithms will.

Always check with your organization's Cryptography Officer (the US government's mandated title for the FIPS compliance officer) before touching *anything*. They can provide you with their organization's configuration standards. Follow them. You're dealing with the FIPS standard, your interpretation of it, the cryptography officer's interpretation, and—worse—the auditor's interpretation. These are not the same. Make the best argument you can, but remember that you're not the one who must face the auditors.[3] In such environments, FIPS compliance is more important than system or connection integrity.

The NIST's document "Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations" is a useful and readable (for government standards) guide.

If you are not in the United States, don't dance away thinking you're free. Other countries have their own standards. Some are based on FIPS. Others, like GOST in Russia, are entirely different. The one thing they all have in common is that the governments involved take them very seriously.

## Applications and TLS

Every application server configures TLS differently. Any application I chose for a reference would be irrelevant to more than ninety percent

3 How do you know if you've pushed your argument too far? FIPS compliance officers always grow louder when they're about to feed on human flesh.

of my readers. I provide very few application-specific examples, except as necessary to demonstrate core functions.

What I will do is provide the roles various components play in TLS. If your Certificate Authority provides a certificate and an intermediate certificate bundle, you will understand how these items play into the whole TLS process. This lets you check your application documentation and see how to configure these components. You'll learn which versions of TLS you should support; your application manual will tell you how to disable the rest.

### *TLS versus DTLS*

The two primary transport protocols underlying the Internet are Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). I should explain—no, there is too much. Let me sum up.

In TCP, the host's network stack is aware of the connection's condition. It knows what packets have been sent and received, and requests or resends any missing data. This is why we call TCP a *connected* protocol. Applications that run over TCP include HTTP and HTTPS, SMTP, SSH, and FTP.

With UDP, the host's network stack flings packets across the Internet willy-nilly. Do they arrive? Do they disappear en route? Who cares? The application takes care of all that. Applications that at least partially rely on UDP include SNMP, DHCP, DNS, and file sharing. Many VPNs only use UDP. If you need more detail, check out a book on networking like my *Networking for Systems Administrators* (Tilted Windmill Press, 2015).

TLS expects the network to handle packet delivery, including coping with any retransmissions and reordering and so on. This means it runs only on TCP.

Many UDP applications also need transport protection, though. Rather than invent a whole new protocol, people took TLS and added a state tracking system to handle all the network accounting. The

result is the Datagram Transport Layer Security (DTLS) protocol.

DTLS was deliberately designed to work exactly like TLS. The code is different, and the packet structure is different, but to a sysadmin it's the same. A TLS certificate works just fine in DTLS. All of your TLS knowledge is fully applicable to DTLS. You still can't telnet to a DTLS-protected service. Even the protocol version numbers were deliberately synchronized: DTLS 1.3 uses the same mechanics as TLS 1.3.

Today, DTLS is mostly used for VPN products. We won't discuss it much, but don't let the acronym distress you.

### *Encryption and This Book*

TLS is built out of encryption. Encryption is built on top of math. Really complicated math developed by brilliant people, and continually inspected, validated, and attacked by folks who are merely terribly smart.

I am not any of these folks.

Chances are you're not, either.[4]

For us, cryptography can feel like magic. How can a 1024-bit ECDSA key be harder to break than a 1024-bit RSA key? Surely computing hash collisions isn't *that* hard? How is public key cryptography even a thing?

If you're seriously interested in the innards of cryptography, this book will not satisfy you. I'll show you how to configure and use TLS, but we both need to accept that cryptography's magic math works. Many people find the study of cryptography a challenging brain-stretching hobby, and if you're that interested I encourage you to dive into any number of books and web sites on the topic. The rest of us must accept that cryptography works, just as we accept that CPUs, memory, and network cards work even though we have zero true comprehension of what goes on inside them.

---

4 If you claim you're one of these brainy folks, I'm not going to argue with you. I won't *believe* you, but I'm not going to argue with you.

### What's in This Book

TLS has been around for nearly thirty years, and has gotten itself wedged into many corners of the Internet ecosystem. This book will not teach you everything there is to know about TLS. No single book can.

What this book will do is give you a solid foundation of what every sysadmin must know about TLS. You'll get in-depth treatment of the Automated Certificate Management Environment (ACME). You'll learn how to use the OpenSSL command line to perform typical TLS operations. More importantly, you'll learn when to use TLS and when it doesn't matter. This knowledge will equip you to safely traverse whatever TLS ledge you're standing on.

So what *will* we cover?

The first half of this book is about the principles of TLS. We'll discuss cryptography and certificates, protocols like STARTTLS, the Tree of Trust and TLS resumption. The second half takes us more deeply into dealing with Certificate Authorities and the Automated Certificate Management Environment.

Chapter 0 is this *Introduction*.

Chapter 1, *TLS Cryptography*, discusses how TLS uses cryptography, including specifics of the TLS protocols like Server Name Indication and the various sorts of resumption and renegotiation.

Chapter 2, *TLS Connections*, teaches you how to use OpenSSL to connect to TLS-protected services.

Chapter 3, *Certificates*, covers the X.509 certificates that form TLS' trust infrastructure. We'll delve into certificate components, encoding, transformations, and all the different sorts of certificates.

Chapter 4, *Revocation and Invalidation*, discusses when you should stop trusting your certificates, what you *should* do about it, and what you *can* do about it.

Chapter 5, *TLS Negotiation*, takes you through the mechanics of

typical TLS 1.2 and 1.3 connections. You can't know where something has gone wrong until you learn to recognize normality.

Chapter 6, *Certificate Signing Requests*, covers the various ways to generate modern certificate requests. Those tutorials you read on the Internet? Most of them have been wrong for at least two decades. You'll learn to get a certificate that fits your needs.

Chapter 7, *Automated Certificate Management Environment*, discusses the emerging ACME standard and how to automate maintenance of your certificates.

Chapter 8, *HSTS and CAA*, helps bring your TLS environment up to current standards.

Chapter 9, *TLS Testing and Certificate Analysis*, covers different ways to audit your SSL configuration. Many TLS misconfigurations are invisible without specific testing.

Finally Chapter 10, *Becoming a CA*, discusses building an internal CA and takes you through building a CA for educational purposes. We'll also touch on special-purpose name constraint certificates, and what it means to become a trusted public CA.

Then there's an *Afterword* and credits and index and stuff.

First, let's talk about codes and ciphers.

# Chapter 1: TLS Cryptography

Cryptography is a vast subject, but fortunately you only need understand a few select parts to properly manage TLS. You don't need to master the mathematics, only the cryptographic system components and how data flows between encrypted and unencrypted states. We'll go through these basics. Cryptography in general has three goals: integrity, confidentiality, and non-repudiation.

*Integrity* proves that the message has not been modified, either accidentally or deliberately.

*Confidentiality* makes the message unreadable by anyone except the intended recipient.

*Non-repudiation* proves that the message came from the declared sender; the sender cannot credibly declare that they didn't send the message.

Different cryptography tools support different aspects of these goals.

## *Hashes and Cryptographic Hashes*

A *hash* or *checksum* is a computation that computes a fixed-length string from any chunk of data. Different hashing algorithms produce hashes of different length, but the length of any hash produced by the algorithm never changes. A SHA256 hash is always 256 bits long, or 64 hexadecimal characters. Hashes can be *cryptographic* or not. The difference is that a cryptographic hash is irreversible in practical terms, as we'll see here. Hashing is an integrity verification tool.

A non-cryptographic hash is useful when you expect corruption rather than deliberate tampering. I could create a simple integrity-checking hash for this book by counting the number of words in the body of the text, and prepending zeroes to make it a fixed number

of digits. You could count the number of words in your copy and compare it to my count to see if it matched. TCP uses a similar sort of math to verify packets haven't been damaged in transit. These are valid hashes, but most hashing algorithms are far more complicated. They are also easily forged. You could replace two critical words from this book and the checksum would still match.

With a *cryptographic hash*, any change in a file changes the hash generated from that file. Suppose a text file contains the text *The wedding is at 10AM*. The SHA256 hash of this text is bdf34d7f51fbe672325a29b0afd7b871513591a0c6dc2c96cb529f6cb877-6070. If someone alters the message and changes the zero to a one, so that it claims that the wedding is at 11AM. With this one-character change, the SHA256 hash of the file becomes 6c717735c4d360d2ae5503f765a6a153c0317720ad17e2869888 e87b52f-25bf0. Human beings are terrible at noticing tiny details, so even someone familiar with the original message might skim over the tampered version and declare it correct. The shallowest examination of the hashes shows that they differ wildly, however.

Given infinite possible files, eventually two of them will have the same hash. This is called a *hash collision*. Resistance to computing such collisions is what makes a hash cryptographic.

Consider the hash of "how many words are in this book." Finding a document with the same hash is trivial. It wouldn't even notice the tampering in our wedding message. This algorithm has zero resistance to deliberate attack. Similarly, TCP checksums can be forged. These hashes are decidedly non-cryptographic.

If you're using a modern, robust cryptographic hash, finding a file that has a hash collision with your target file will take decades on an industrial-scale farm of high-end servers. You won't be mechanically computing such a file, either; rather, you'll try random files until you find a match. Finding a hash collision

is inevitable given infinite time, infinite computing power, and infinite budget. I might know that a message has a hash value of bdf34d7f51fbe672325a29b0afd7b871513591a0c6dc2c96cb529f6cb877-6070, but I have no realistic way to create a file with that same hash.

When the software finds a hash collision, the data would almost certainly bear no resemblance to the original message. A human being might not notice the wedding's changed hour in the message, but they'd certainly notice a message that contained only binary gibberish. Computing a file that has a matching hash, that's similar enough to the original to be useful, while still containing your alterations? This is not going to happen with current technology.

What makes a cryptographic hash successful? By the time a useful collision is found, the wedding is long past and everybody has already lived happily ever after.

### *Symmetric Encryption*

The common definition of "encryption" means scrambling a message so its contents can only be understood by the intended recipient. A villainous lackey encrypts a message so that only his prince can read it. A *cryptographic algorithm* is a method of encryption.

*Symmetric* algorithms use the same key to encrypt and decrypt text. If you have the key and know the algorithm, you can encrypt and decrypt messages. Symmetric algorithms rely on keeping the key secret. Most encryption algorithms used in the last ten thousand years are symmetric. People outside computing and cryptography might call a symmetric algorithm a *code*, but that's a heavily overused word in our profession so please avoid further burdening it. ("Encoding" is something entirely different, as Chapter 3 discusses.) The word "cipher" is often used outside computing to mean a symmetric algorithm, but in systems administration a cipher is most often shorthand for a cipher suite as discussed later this chapter.

A well-designed symmetric algorithm retains message confidentiality even if the algorithm is known. Consider the first code every kid learns: A=1, B=2, and so on. This algorithm is poor, because if you know the algorithm you can decrypt the message. You can slightly improve this algorithm by adding a secret key, a number to be added to each value. A key of 13, meaning A=14 and B=15 might look harder to crack, but an experienced cryptographer can decrypt a message with very little effort. Modern symmetric algorithms like CHACHA and AES retain confidentiality even when everyone knows how they function.

Computers have raised the standards for symmetric algorithms. Modern algorithms are complex, with long and cumbersome keys. Once two entities can exchange the secret key, they can communicate quickly and easily. The problem is getting the secret key from one to the other. That's where public key encryption comes in.

### *Public Key Encryption*

You've probably seen one of those old movies where a coin has been cut in a complicated jigsaw pattern. Two spies or criminals or other secretive sorts who have never met are each given half of the coin. When they do meet, each produces their half of the coin. If the halves fit together perfectly, each can be assured that the other person is who they're supposed to talk to. The split treasure map, medieval tally sticks, the two-sided coin that leads Indiana Jones to the Ark of the Covenant—our culture uses this idea over and over again.

That's sort of like *public key* encryption , also called *asymmetric* encryption.

The standard for public key encryption is the Rivest, Shamir, and Adelman (*RSA*) algorithm, named after its inventors. The Elliptic Curve Digital Signature Algorithm (*ECDSA*) is a newer method that works differently, but the sysadmin-level practice of using it is nearly identical.

You'll hear the phrase "public key encryption" tied with authentication, digital signatures, HMAC and hashes, key exchange, and all sorts of other terms. These systems are built *with* public key encryption, but they aren't public key encryption. Don't let sloppy language confuse you. Public key encryption encrypts. That's all it does. The way it encrypts and decrypts lets us build all this cool stuff. Understand how public key encryption works before even trying any of the other things.

Public key encryption is like using the two parts of that jigsaw coin as encryption keys. Each piece of coin can encrypt a message that can only be decrypted by the other piece of coin. If I have one of the halves, I can encrypt a message that can only the possessor of the other coin piece can read. My coin piece can't even decrypt a message that it encrypted; only the other piece of that same coin can decrypt it. (This is a major difference between public key encryption and symmetric encryption.) These two pieces of coin are paired; anyone who has one of the pieces can encrypt and decrypt messages.

Hang on to your piece of coin. You'll need it later.

Public key encryption is like that cut-up coin, but with numbers so big that working with them realistically requires computers. Such numbers behave weirdly when multiplied together.[5] You can easily and cheaply generate two related *keys*, called a *key pair*, that behave exactly like those two pieces of coin. You can encrypt a message with one key, and only the other key of the pair can decrypt it. This is the entire function of public key encryption.

Numbers differ from our coin pieces in a couple ways. If you have half of a jigsaw-split Canadian loonie, you can cut another coin to fit

5 One April night in 1977 Ron Rivest drank a "disproportionate" amount of wine and created the first one-way function for public key cryptography. People who understand the math assure me that, while the function certainly works, it is best studied while in a similar state.

it. Trying the same thing with half of a public key pair and a bunch of computers will take longer than our Sun's remaining lifetime. Having one key of the pair doesn't help an attacker figure out the other key.

Duplicating your half of a public key is trivial; you copy the file.

Let's say I create a public key pair. I keep one key of the pair. The other key I give to my chief goon, Vizzini, before I dispatch him out into the world. Nobody else has either of these keys. My goon and I can use these keys to exchange messages that can be read only with the other key in the pair. I use my half to encrypt my messages to my goon. Hopefully I remember what I said, because once the message is encrypted, my key cannot decrypt my message. Fortunately, "Start a war and frame Guilder for it" is short enough that even I can remember it. Only the other key in the pair can decrypt that message. I mail my message. Anyone who snoops on that message sees only indecipherable gibberish.

When my message reaches my goons, they use their key to decrypt it. They can then use their key to encrypt a response, like "How about we kidnap the princess?" and send the encrypted message back to me. Only my half of the key can decrypt this message.

So far, so good.

Where public key encryption gets wild is that we can give one key of a pair away. To the whole world. Put it on your web site. Or a billboard. Get a doomsday laser and engrave it on the near side of the Moon, large enough to be read by anyone. We call this key of the pair the *public key*.

The other key of the pair you keep utterly confidential. It is locked down to the best of your ability. It never leaves your control. We call this the *private key*. (The secret keys used in symmetric encryptions might also be called "private keys." Don't let that confuse you.)

When I encrypt a message with my private key, anyone can grab the public key and decrypt it. That doesn't exactly make my

messages confidential . But anyone in the world can use that key to encrypt a message that only I can read it. Anyone can confidentially communicate with me.

If you publish one of your keys, I can use that key to confidentially communicate with you.

If I am the only person in the world with my private key, anyone who receives a message encrypted with my private key can be pretty sure that the message came from me. I cannot credibly claim that I didn't send the message. We use this to meet cryptography's goal of *non-repudiation*.

To send a secret message to my goon, I encrypt it with my goon's public key. Only my goon can read it. My goon replies by encrypting his message with my public key.

Public key cryptography relies on each of us keeping our private keys truly private. If I am sloppy with my private key and it gets stolen, anyone in the world can pretend to be me. I can repudiate a message only by admitting to gross incompetence.

This technology underlies most of the Internet's encryption, including TLS. Any time you see the phrase "public key encryption," immediately consider who has which keys. If you become responsible for an unfamiliar system that uses public key encryption, your first and immediate responsibility is verify that the private keys are utterly locked down.

If public key encryption has these nifty features, why bother with symmetric encryption?

Public key cryptography takes thousands of times more computing power to encrypt, decrypt, and validate than symmetric algorithms. It is slow, expensive, and raises everyone's electric bill.

TLS, and most other applications that leverage public key cryptography, combine the two types of algorithms. We use slow and expensive public key algorithms to authenticate everyone, agree on

a symmetric algorithm, and exchange a random secret key for that algorithm. All further communications take place using the faster and cheaper symmetric algorithm.

Using public key cryptography is complicated. Public key cryptography users must agree on a whole series of algorithms, methods of distributing trust, and more. A collection of these is a *public key infrastructure* (*PKI*). If you've browsed the Internet, you're using the TLS PKI. Microsoft's Active Directory includes its own PKI, for use only within the domain. Enterprises can establish their own PKI. OpenPGP has yet another PKI. We'll spend most of our energy discussing the PKI used for globally valid TLS. Chapter 10 discusses building your own CA, a private PKI, for learning purposes.

### Message Authentication Codes

Hashes are great, so long as you take care to distribute the hash and the message over different channels. If I can intercept your message and change both the message and the accompanying hash, the hash is useless. That's where we need a *Message Authentication Code*, or *MAC*. A MAC is a hash encrypted with a symmetric encryption key known only to the sender and the recipient.

A *Hashed Message Authentication Code*, or *HMAC*, is specific method of using a particular cryptographic hash to create a MAC. You'll see HMAC names like HMAC-MD5 and HMAC-SHA256, built on those hash methods. An HMAC provides both integrity and authentication. Only someone with the symmetric key can encrypt or decrypt the hash.

The MACs used in TLS are generally HMACs.

### Digital Signatures

Cryptographically, a *digital signature* ensures that a message comes from the entity that claims to have signed it, and that nobody else

tampered with the message.[6] A digitally signed message is considered *authentic*. It has integrity, authentication, and non-repudiation.

Software digitally signs a message by generating a HMAC of the message, encrypting that hash with a private key, and attaching the encrypted file to the message. Anyone with the public key (that is, *everyone*) can decrypt the HMAC, independently compute the hash of the message, and compare the two. If the transmitted hash and the computed hash match, the message came unaltered from the private key holder.

Digital signatures are another reason you must protect your private keys. If someone has your private key, they can send messages you cannot repudiate. Losing exclusive access to your private keys is like handing the neighbor's teenage children your credit cards. You can clean up the mess, but it's going to hurt a whole bunch.

If the recipient calculates the message hash and it differs from the one in the digital signature, the digital signature is invalid. The message was altered. Do not trust it.

### Key Lengths

A key's *length* is the number of bits in the private key. A key length of 2048 means that the key contains 2048 random zeroes and ones. Both symmetric and public key encryption use key lengths.

If someone wants to forcibly decrypt your message, they could try every possible random key one after the other. This *brute force attack* eats a whole bunch of computing power, but can easily be divided between multiple machines. Each additional bit of length doubles the number of potential keys. If you want a key that's twice as hard to guess as a 2048-bit key, it's not a 4096-bit key. It's a 2049-bit key. A 4096-bit key has $2^{2048}$-1 *times* as many possible keys as a 2048-bit key. That's a key space about $3.2 \times 10^{616}$ times larger.

6 We are not discussing programs that let you "digitally sign" mortgages, employment contracts, or murder confessions. Such software is far less robust.

Symmetric and hashing algorithms usually have a fixed key length. Some symmetric algorithms let you choose between multiple key lengths. AES128 uses 128-bit keys, while the keys of AES256 are twice as long. The algorithm's strength is most often exponentially proportional to the key length.

Public key encryption algorithms let the software or the sysadmin choose a key length. Most software provides a default, which you can probably override. I recommend not doing so unless specifically instructed to do so by a reliable source. Reliable sources do not include emphatic forum posts, no matter how tightly they conform to your biases.

If a longer key is harder to decrypt, shouldn't you always use the longest possible key? Absolutely not. The computations for public key cryptography take a bunch of processor time, and longer keys demand more computing power. Your server doesn't exist for computing keys; it exists to serve web pages or email or some other minimally viable product, and the key computations are a necessary but incidental part of that. If your server spends all its time computing keys, it has nothing left for performing its function. The long key protects against a specific sort of attack while interfering with actual work.

If a longer key length is unwise, would you ever use a shorter key? Not in the real world.

Key length is not the only factor in how difficult it is to break a message. The algorithm also affects it. A message encrypted with 2048-bit RSA keys is about as difficult to break as one encrypted with a 224-bit ECDSA key, for reasons involving lots of horrible math. A longer key length does necessarily mean "harder to break." The algorithm matters.

If your organization's standards differ, follow those standards. Financial institutions and suchlike often use longer keys because they're high value targets. They also have money to spend on the hardware for these calculations.

If you're stuck using older software, you might need to increase default key lengths to comply with current standards. While 1024-bit RSA keys were perfectly suitable for the first twenty years of the commercial Internet, it became clear several years ago that brute forcing these keys would soon become practical. Experts advised creating all new keys with 2048-bit keys. Software defaults lagged behind the recommendation, sometimes for performance reasons on older hardware, so I had to override the defaults until everyone caught up.

Yes, one day we'll need 4096-bit RSA keys. Barring dramatic breakthroughs in mathematics or computing hardware, this will be well after the career of anyone reading this book. If you're looking at the future, the web site https://www.keylength.com/ extracts key length recommendations from several standards.

Eventually, everything ages out.

### *Breaking Algorithms*

What does "breaking" an algorithm mean? For public key encryption, it means you can compute a private key from a public key and an encrypted file. For hashes, it means you can computationally create a file with any given hash. Breaking the algorithm happens in one of two ways. Someone might figure out a flaw in the algorithm that allows one to shortcut decrypting a file or computing a collision. Alternately, an algorithm is considered broken when a single entity such as a government or corporation can amass enough processing capacity to break the algorithm by brute force. Faster computers make both easier.

While cryptography is an ancient craft with a long and glorious tradition, computer-based cryptography is only a few decades old. Before the Internet went commercial, processor time was so expensive each department got billed for its CPU usage. Unix still includes features to keep track of how much computation a user consumes, so finance departments can demand payment. Computers made practical many algorithms unworkable on pen and paper.

Testing and breaking these algorithms demanded a disproportionately larger amount of computing time. Consider the typewriter-sized Enigma machines, and Bletchley Park's massive but far more complex Enigma-breaking hardware. What one person could encrypt required entire teams to forcibly decrypt. This dynamic persisted throughout the early days of the commercial Internet.

The early computerized cryptographic algorithms, many of which were included in SSL and TLS, were rushed, insufficiently tested, and provided many opportunities to learn how computing-based cryptography really worked. Many of these algorithms cracked upon exposure to reality. We discarded them and learned from our mistakes.

As one example, MD5 and SHA-1 were the standard hashes in early versions of SSL. Computing an MD5 hash collision required buildings full of computers and would have still taken centuries. SHA-1 hashes took more computing power to calculate, so they were less frequently used even though they were harder to break.

Computing advanced. We learned more about cryptography. Breaking algorithms became a way to earn prestige. In a few years, computing an MD5 collision became a realistic possibility. MD5 was declared obsolete, and everyone was encouraged to stop using it. SHA-1 wasn't so much a deliberate mitigation of MD5's obsolescence so much as the last algorithm standing. It held up well through a decade or two of the same abuse, but today you can compute a SHA-1 hash collision with a few thousand bucks at a cloud provider. It's not yet trivial, but it's sure not impossibly expensive.

Newer algorithms, like ECDSA and the SHA-2 family, build on experience. These algorithms are expected to survive beyond the career of anyone reading this book. We need more algorithms that work differently, so when someone gets clever and breaks these we have replacements ready.

When cryptographers create new algorithms, they and their

colleagues attack them. They also calculate a pretty good estimate for how much computing power will be needed to brute-force it. Only cryptographic hash algorithms that are expected to remain unreversible for centuries, despite advancing hardware, reach broad acceptance. The cryptography community must broadly agree that the new algorithm is suitable and deploy it in applications that get rolled out to users. Development and deployment takes years.

A broken algorithm no longer guarantees confidentiality or integrity. Hopefully all your software has been updated to offer other algorithms by then. Design your applications so that algorithms can easily be added and removed.

Many Internet protocols operate by Jon Postel's robustness principle: "be conservative in what you do, be liberal in what you accept from others." Your applications transmit the best data they can, and compensate for other people's inadequacies. We're all on this Internet together. The fact that everyone can communicate is a miracle, and we all know what happens when you rush miracles.

Not only does this rule not apply to TLS, following it imperils both your server and your clients. Each TLS version specifies acceptable algorithms. Older versions of TLS use fragile or flat-out broken algorithms. Any application that supports TLS 1.1, TLS 1.0, or any version of SSL puts data at risk of destruction, alteration, and deletion. You *must* disable obsolete versions of TLS, or you're flinging your whole organization into the Pit of Despair.

Cryptographers constantly test algorithms. Finding flaws in cryptographic algorithms makes cryptographers happy. They get most happy when they discover a way to break a previously trusted algorithm. Even if you use only current TLS versions, you might need to disable certain otherwise-permitted algorithms. Historically, any TLS version outlasts some of the algorithms it supports. Assuming the current crop of algorithms is unbreakable is a great way to have a bad day.

## *Cipher Suites*

In TLS terms, a *cipher suite* is a specific combination of a particular symmetric, checksum, and public key algorithms, plus other details needed so that each party can encrypt and decrypt messages comprehensible to the other party. It's often referred to as a *cipher*.

Cipher suites are used between the TLS client and server. They have no impact on, and are not constrained by, the signature algorithm used on any certificates.

### Cipher Suite Names

Each cipher suite has an official name defined by the Internet Assigned Number Authority (IANA),
in their document "Transport Layer Security (TLS) Parameters." The ciphers have names like TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 and TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256. You do not need to know the innards of each cipher suite or understand the underlying algorithms, but you must notice mismatches. If one side of a TLS connection demands TLS_AES_128_CCM_SHA256 and the other insists on TLS_AES_128_CCM_8_SHA256, that extra _8_ buried in the middle will give you grief.

TLS 1.2 and 1.3 use different syntaxes for naming ciphers. TLS 1.2 cipher names can include up to six pieces of information.

`Protocol_Kx_Au_WITH_Enc_MAC`

The protocol is *TLS*.

*Kx* is the key exchange method, one of ECDHE, DHE, ECDH, DH, or RSA.

*Au* is the authentication method, either ECDSA or RSA. If both key exchange and authentication use the same method, such as RSA, it only appears once.

*Enc* gives the symmetric encryption plus the mode of operation, one of CBC, CCM, CCM_8, or GCM.

The *MAC*, or Message Authentication Code, is one of SHA, SHA256, or SHA384.

TLS 1.3 removed key exchange and authentication methods from the cipher name. The cipher suite no longer dictates them. It also dropped the WITH separator. The names are much shorter.

*Protocol_Enc_MAC*

The presence or absence of _WITH_ helps identify TLS 1.3 and 1.2 ciphers. Also, the TLS 1.3 standard includes only five ciphers. No TLS 1.2 ciphers survived.

```
TLS_CHACHA20_POLY1305_SHA256
TLS_AES_128_GCM_SHA256
TLS_AES_256_GCM_SHA384
TLS_AES_128_CCM_SHA256
TLS_AES_128_CCM_8_SHA256
```

When you're studying debugging data, the cipher names instantly identify which version of TLS you're using.

While cipher suites set a whole bunch of algorithms, they have some flexibility. For example, the cipher TLS_AES_128_GCM_SHA256 doesn't mention the public key algorithm, so the client and server can negotiate that.

Never hand-pick ciphers. This month's best cipher is next month's nightmare. We'll see how to use the best available ciphers in "Cipher Lists" later this chapter.

#### Alternate Cipher Names

TLS software developers keep falling victim to classic programmer blunders—the best known of which is thinking, "I can write something better than this widely used schlock," but almost as well known is "renaming things that already have a standard name." If you understand the algorithms well enough to write useful code for them, and understand how the cipher will be deployed, the impulse to simplify TLS_RSA_WITH_AES_256_CBC_SHA to a shorter but still unique identifier like AES256-SHA is understandable.

OpenSSL gave ciphers their own names up through TLS 1.2. They were not alone in this. The TLS 1.2 cipher officially known as TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 is called ECDHE-ECDSA-AES128-GCM-SHA256 in OpenSSL land, while GnuTLS calls it TLS_ECDHE_ECDSA_AES_128_GCM_SHA256. I readily concede that hyphens are easier to type than underscores. In many cipher names the word _WITH_ doesn't clarify anything, but in others it's a vital separator. When an algorithm mismatch causes mayhem, however, using different names for the same cipher suite leads to inevitable bewilderment.

I find **`https://ciphersuite.info`** highly useful for translating between the different names and identifying the components of the algorithm. These sites also provide hints about the algorithm's current strength.

### Included Cipher Suites

Use `openssl ciphers` to view the cipher suites your OpenSSL build is aware of, listing both the standard and OpenSSL names. Use `-v` to list each cipher on its own line, with details. Add `-s` to include only supported ciphers. The `-stdname` flag displays the standard name. The table's too wide to show comfortably in any version of this book, so I'll use awk(1) to print only the columns showing the standard and OpenSSL names for TLS 1.2. I find `column -t` invaluable for shaping such output.

```
$ openssl ciphers -v -stdname -s | awk '{print $1, $3}' | column -t
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384       ECDHE-ECDSA-AES256-GCM-SHA384
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384         ECDHE-RSA-AES256-GCM-SHA384
TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 ECDHE-ECDSA-CHACHA20-POLY1305
…
```

Alternately, you could use `-V` instead of `-v` and get the cipher suite's hex values and the official name at the beginning of the table, but that alters your awk(1) command.

To get more detail on the cipher in a terminal of modest width, don't use `-stdname`. You'll get six columns. Use `column -t` to get more legible output.

```
$ openssl ciphers -v -s | column -t
TLS_AES_256_GCM_SHA384  TLSv1.3  Kx=any   Au=any    Enc=AESGCM(256)  Mac=AEAD
TLS_AES_128_GCM_SHA256  TLSv1.3  Kx=any   Au=any    Enc=AESGCM(128)  Mac=AEAD
AES256-GCM-SHA384       TLSv1.2  Kx=RSA   Au=RSA    Enc=AESGCM(256)  Mac=AEAD
ECDHE-ECDSA-AES256-CCM  TLSv1.2  Kx=ECDH  Au=ECDSA  Enc=AESCCM(256)  Mac=AEAD
AES256-SHA256           TLSv1.2  Kx=RSA   Au=RSA    Enc=AES(256)     Mac=SHA256
…
```

The first column gives the OpenSSL name for this cipher.

The second column shows the TLS version that introduced this cipher. You'll see ciphers that appear to support obsolete versions of TLS, or even SSL. While SSLv3 is obsolete, ciphers introduced in that version are still in use in TLS 1.2. You can add the `-tls1_2` or `-tls1_3` flags to list only the ciphers used by a specific version of TLS.

The *Kx* column shows the cipher's key exchange algorithm. The first two ciphers listed could use any key exchange algorithm, the third must use RSA, and the fourth ECDH.

The *Au* column gives the key's authentication algorithm. While the first two can, again, use any authentication algorithm, the third uses RSA and the fourth ECDSA.

TLS 1.3 changed how key exchange and authentication work. They're no longer part of the cipher suite. The *any* shown for these ciphers is legacy formatting.

Under *Enc*, we see the symmetric encryption algorithm. Most modern ciphers use some variant of AES.

The *Mac*, or Message Authentication Code, gives the algorithm used to authenticate symmetric data. Many of these show AEAD, or Authenticated Encryption with Associated Data. The MAC is incorporated into the encryption method. GCM is the most widely deployed AEAD algorithm, and it uses fixed MAC algorithms. AES_128_GCM always uses a SHA256 MAC, while AES_256_GCM uses SHA384. Our last example is a non-AEAD cipher, and it uses SHA256.

### Cipher Lists and Cipher Ordering

OpenSSL supports *cipher lists*, letting you select exactly which ciphers you will permit an application to use. It includes a variety of built-in cipher lists with names like HIGH, MEDIUM, RSA, ECDHE, and so on. The ciphers(1) or openssl(1) man page contains a list of the built-in lists. Perhaps best known is the HIGH list, which contains the strongest cipher suites.

A cipher list restricts what ciphers a command can use. Many applications accept cipher lists somewhere in their configuration. If an application is negotiating a TLS connection and should use only RSA ciphers, setting the cipher list RSA in the application accomplishes that.

You can view the cipher suites in a list by giving the list name as the final argument in an `openssl ciphers` command. This restricts the command to only ciphers in that list.

```
$ openssl ciphers -v -s HIGH
```

Suppose you have a customer whose IT staff insists that the application you use to exchange data only offer ciphers that use AES in Galois Counter Mode . Such policies are doomed to obsolescence, exactly like FIPS, but they're giving you money so you provision a dedicated virtual machine for them and roll with it. Checking the manual page reveals the AESGCM cipher list. Use it to show the algorithms your customer will find acceptable.

```
$ openssl ciphers -v -s AESGCM
```

If you wander around the Internet's fiery swamp of HOWTOs, you'll see recommendations to use only ciphers in the HIGH cipher list. This list was vital in the early days of SSL, when constantly escalating computing power in the hands of avid algorithm-breaking cryptographers meant sysadmins needed to constantly update their cipher lists. Decades ago, HIGH was a convenient shortcut that

updated algorithms every time the sysadmin updated OpenSSL, without the sysadmin needing to know the details of which algorithms had been broken this week.

Today's HIGH list contains all TLS 1.2 and 1.3 ciphers. If you only run modern TLS versions, you might think you don't need to set a cipher list in your application. New ciphers will appear, though. And some clever cryptographer might break an algorithm we think reliable. OpenSSL updates its cipher lists when that happens. Using the HIGH cipher list in your configurations makes no difference at this moment, but at each OpenSSL upgrade automatically propagates changes through your application stack. Specifying HIGH is a proactive defense.

If a tutorial recommends altering the cipher list, check the date. It's probably far obsolete.

The list of permitted algorithms isn't the only factor, though. Even in the HIGH cipher list, some algorithms are better than others. OpenSSL's *cipher order* ranks algorithms by strength. TLS protocol negotiation includes agreeing on a cipher. The client declares every algorithm it supports. The server picks the one it likes best. If the server only supports algorithms it considers strong, it could allow the client to pick the algorithm. This makes sense when clients might have a hardware cryptographic accelerator. If the server supports a variety of strong and weak algorithms, the server should use the strongest algorithm possible.

Unless you have reason to do otherwise, configure your applications to support both the HIGH cipher list and cipher ordering.

#### When HIGH Isn't Enough

Some people think that the HIGH cipher list isn't enough. Perhaps your application is a special delicate flower and you must restrict your cipher list to only the very best algorithms. My first question would be: are you going to schedule time to monitor the state of cryptographic

algorithms? When the job gets busy and you're coping with outages and managers and customers, will you drop everything to investigate the current state of different algorithms? Can you easily test changes and push them out to all of your servers? Are you willing to deal with problems caused by old clients trying to access a tightly restricted cipher list?

Now that you've said yes: are you *sure*? Really super sure?

If so, Mozilla provides configuration guidelines for many popular application servers at `https://ssl-config.mozilla.org/` that include only very strong ciphers. Weekly checks for changes in these configurations provides good guidance on the current state of cryptographic algorithms.

Long term, I have never seen an organization successfully manage tightly restricted cipher lists like this. I am perfectly okay if you prove me wrong. Personally, I'm sticking with HIGH.

### *Trust Models and Certificate Authorities*

The big problem of public key encryption is determining an authoritative source of trustworthy public keys. You'll find two general models for trusting public keys, the Web of Trust and Certificate Authorities. They differ by who performs the labor of deciding who to trust.

The *Web of Trust* model popularized by OpenPGP requires the user to make the initial trust decisions, and then assesses the world with transitive trust. It boils down to "I chose to trust key A, and key A trusts key B who trusts C who trusts D, so I will trust D." It's wholly unsuitable for end users who merely want to purchase a custom-fitted six-fingered glove without a packet sniffer stealing their credit card details.

The *Certificate Authority* (CA) model escapes making the user do any trust work by choosing to trust organizations that have badgered, bribed, belabored, or blackmailed their way into being considered trustworthy. These broadly trusted *Certificate Authorities* can sign

certificates for the rest of us, and are subject to audits to ensure they only sign valid certificate requests. They are the root of the public PKI.

Both models have problems. The Web of Trust is confusing, difficult to use, and easily gamed. Any system that relies on every individual user's ability to make good decisions is inherently flawed, as any democracy demonstrates. The authoritarian CA model was expensive until recently, and relied upon CAs perfectly verifying all information. Any system that relies on elites to make the important decisions is also inherently flawed, as any technocracy proves. The only sensible thing to do is to pretend safety exists and take refuge in comforting 1980s fantasy flicks.

SSL and TLS were designed to separate people from their money, and "user ease" is the deciding factor in such protocols. User ease requires trust. The Certificate Authority model became the standard in SSLv2.

Even under the CA model, not all devices and applications trust the same Certificate Authorities. We discuss trust bundles in Chapter 3.

The advent of the Automated Certificate Management Environment (ACME) meant routine X.509 certificates can be obtained for free.

### *Private Key Protection*

Anyone who can read the private key of a pair can use that key's certificate to masquerade as the key owner. If I have a TLS certificate authoritatively identifying my web site as **`https://mwl.io`**, anyone who gets the private key can set up a rogue web server pretending to be my site. (Anyone who can convince a CA to sign a certificate for that domain can also do so, but that's a different attack.) Certificate files must be readable only by **`root`** and the application using the certificate. They should not be writable by anyone except **`root`**.

If anyone does connive access to the private key, the entire certificate is compromised and can no longer be trusted. You must revoke the certificate and generate a new one.

By default, private key files are encrypted with a passphrase. The key cannot be used unless a human being types the passphrase, which makes the private key file useless to intruders who don't have the passphrase. The downside is, the key won't activate until a human being types the passphrase. Server reboot? TLS services are down until a human being types the passphrase. Does your server have operators standing by all day and night? Do you have staff ready all hours of the day? Do you trust those staffers with the passphrase? Most small companies with limited staff (and many large companies) decide that passphrases are not viable.

If your environment truly needs to protect its private key, consider putting the private key on a hardware security module (HSM). Accessing the private key means having the physical device, which can fail. As an intermediary step between an expensive HSM and basic Unix, you might use applications and operating systems that support extensive privilege separation, like OpenBSD.

Encrypting your private key is not irreversible (unless you lose the passphrase). If you try to work with an encrypted private key and discover that your organization can't handle it, create an unencrypted private key file as seen in Chapter 3 or install an HSM.

### *TLS Resumption*

Cryptography is not only complicated, the math is tedious, odious, and expensive. The initial public key computations at the start of every connection burn CPU cycles. It'd sure be nice if we could reuse that initial certificate validation and protocol negotiation when the client clicks on the next link of the web page or new mail check.

*Resumption* allows a TLS client to pick up where it left off. The first time you click on a web site, the client and server perform the full TLS validation and negotiation. The second click, though, the client resumes the previous TLS session and saves a round of network requests and public key computations. Resumption improves how

quickly TLS-wrapped protocols respond to subsequent requests, especially useful for interactive applications where one "session" involves many connections, like the web.

TLS 1.2 and earlier based resumption on client-side *session tickets* and server-side *session caches*. Both had similar problems. A TLS 1.2 session ticket is a blob of data that uniquely identifies a particular client. The ticket contained enough information about the cipher's pre-shared key (PSK) to identify the session, derive the key, and start a new connection with the same settings. This form of resumption turned out to be easily broken, and should not be used.

TLS 1.3 discarded the earlier resumption method in favor of a pre-shared key (*PSK*) system. It still uses session tickets, but in a more restricted manner. It remains unbroken, so far.

No matter the TLS version, resumption poses privacy risks to the client. Servers can uniquely identify and track browsers by their tickets and PSKs. This isn't a concern for server operators; we use TLS certificates precisely so clients can uniquely identify us! Privacy-sensitive users might want to disable TLS resumption in their client, however. Load balancers and other network devices further complicate resumption. It might or might not work in any given environment.

### TLS Secure Renegotiation

In TLS 1.2 and earlier, the server and client could change the terms of the session through *Secure Renegotiation*. A web site might require medium-strength encryption for most of the site, but require strong encryption on login pages. In the earlier days of the public Internet, when SSL sessions were so compute-heavy that many operators added hardware SSL accelerators to cope with the load, this flexibility was important.

Like resumption, it turned out Secure Renegotiation was not so secure. It got broken, patched, hacked again, repatched, and eventually put out with the trash. If your application supports Secure

Renegotiation, a properly placed malefactor might be able to snoop on your traffic.

TLS 1.3 does not allow renegotiation of a TLS session; once everyone agrees on the connection's terms, those terms remain unless you create a new TLS session. Combined with today's much faster and more parallel processing, cryptographic load is less of a concern.

### *Perfect Forward Secrecy*

When all of your data is encrypted with your private key, anyone who has the key can capture and decrypt your data. That's a TLS nightmare. But what if the intruder captures and saves your data in transit, then steals your private key? It's happened.

Many TLS 1.2 and earlier ciphers used the certificate's RSA key for authentication in both key exchange and authentication. If you had the private key and a saved packet capture, you could easily decrypt the session. Popular packet sniffers include a "point me at your private key" file for exactly such purposes.

The Diffie-Hellman Ephemeral Key Exchange (DHE) algorithm let us avoid using the certificate's key. Elliptic curve (ECDHE) versions followed. These ciphers do not use the certificate's RSA key to agree upon a symmetric key. A captured packet trace cannot be decrypted. This is called *Perfect Forward Secrecy* or *PFS*. Despite the name, Perfect Forward Secrecy is not perfect. No cryptographic solution ever is.

In all TLS 1.3 ciphers and some TLS 1.2 ones, the key exchange now uses an ephemeral key exchange algorithm like ECDHE or DHE.

You can decrypt TLS 1.3 sessions live if you already have access to the client or the server and can configure a temporary key log file to capture the master key. TLS is not intended to protect against such debugging.

### Server Name Indication

One IP address can host many web sites. When a client connects to a TLS-wrapped web site, the web server needs to know which site the client is trying to connect to. The HTTP 1.1 protocol gives the site name after the connection is established. How do you tell the web server which certificate to use to encrypt the connection when the target site hasn't been identified yet?

The answer is *Server Name Indication*, or *SNI*. The client includes the destination site in the initial TLS negotiation. The server uses the hint from SNI to establish the connection, and then lets HTTP make its separate request. Servers that don't support SNI can support only one TLS site per IP address.

Even in TLS 1.3, SNI is one of the few parts of the connection still sent in cleartext. Eavesdroppers can tell which site you're accessing. People are working on an encrypted SNI, ESNI, but nothing is yet finalized.

With this foundation, let's use TLS across the network.

# Chapter 2: TLS Connections

In the early days of the Internet, we were happy to get data across the network at all. We didn't have the resources to encrypt it, and we weren't daft enough to think that anyone should trust the network with vital data. Then we let the rest of you on the Internet (for some reason that still escapes me), and transport encryption became vital. SSL, the predecessor to TLS, got wedged into existing network services in whatever manner seemed most sensible at the time.[7] TLS inherited these methods, and you must be able to cope with them.

One key tool for debugging network daemons is eliminating the user-friendly client and connecting to the daemon interactively, using a tool like netcat(1). If I want to know if a SSH or SMTP daemon is reachable, I might use netcat to connect to port 22 or 25. Do I get a protocol banner back? Or does something accept the connection but not answer?

```
$ nc mail.mwl.io 25
220 mail.mwl.io ESMTP Sendmail 8.15.2/8.15.2; Fri, 28
Aug 2020 14:23:09 -0400 (EDT)
^C
```

I'm talking to a mail server. The network works. If I'm conversant with the modern SMTP protocol, I can even chat with the server. You can do the same for any text-based protocol.

TLS complicates this debugging. If I use netcat to connect to a TLS-wrapped port, I'll get binary TLS in my text-only client. This would not make me happy. I need my client to handle TLS for me. OpenSSL includes the `s_client` subcommand for creating and debugging TLS-wrapped network connections. We'll demonstrate connecting to daemons using `s_client` so you can manually interrogate your own daemons.

7 Trying all these different approaches taught us a great deal, mostly about what not to do.

Many netcat variants support TLS, and even have TLS debugging features. Every netcat fork has picked different features and unique arguments, however, so I can't document them here. For that reason, the examples all use `s_client` to build your understanding. Once you know how a TLS connection works and what "normal" looks like, use whatever tool you prefer.

## *Connecting to Ports*

Clients connect to network ports via direct TLS connections, TLS connections with CR/LF handling, STARTTLS, or methods like DTLS.

### Connecting versus Debugging

The `s_client` command was written for debugging TLS connections. When it discovers invalid certificates, it defaults to accepting them and continuing on. If you're relying on OpenSSL to expose TLS problems, add the `-verify_return_error` flag to all of your `s_client` commands. If you're investigating a daemon or application protocol problem within the TLS wrapping, you could skip this flag.

The `s_client` command includes a variety of specific debugging options, as well as flags for exotic edge cases and truly bizarre situations. If you have a problem that seems not only strange but downright perverse, read the manual page to see if anything there seems appropriate.

### Line Feeds, Carriage Returns, and Newlines

Anyone reading this book has tripped over the way different operating systems treat the "go to the next line" characters. Unix treats a line feed (LF) as a new line. Microsoft operating systems treat a carriage return with a line feed (CRLF) as a new line. Early MacOS used a carriage return (CR) as a newline, and who knows what other operating systems use? FTP's binary and ASCII modes exist to cope with newline handling.

This difference extends into plain text network protocols. Some protocols, like HTTP, expect to be able to send a carriage return without getting a new line. OpenSSL needs to cope with both.

Modern OpenSSL uses either the `-connect` or the `-crlf` options to connect to network ports. The `-connect` command treats `ENTER` as a carriage return with a line feed, while `-crlf` treats `ENTER` as a carriage return.

With LibreSSL and older OpenSSLs, you must always specify the target host and port with `-connect`. Add `-crlf` if you need its newline handling.

Also, some Unixes ship modified OpenSSLs that behave differently. If `-connect` or `-crlf` doesn't work well, try the other one.

#### TLS-Dedicated TCP Ports

Common network services like web, email, and FTP attach to dedicated TCP/IP ports. The simplest way to wrap a daemon's connections in TLS is to assign a different port for the TLS-protected version. That's why HTTP runs on TCP port 80, while HTTPS uses port 443. The application developers had to do comparatively minor changes to their applications, and didn't have to worry too much about TLS's innards.

Here my mail client is annoying me and I'm not sure if it's the network or my client or me, so I use `s_client` to talk directly to the TCP port dedicated to TLS-wrapped POP3. The `-connect` argument tells `s_client` to create a TLS tunnel to the given host and port. I'm adding `-verify_return_error` because if it's a TLS error I want to know about it.

```
$ openssl s_client -verify_return_error \
  -connect imap.gmail.com:995
```

I'll see a whole spew of TLS information, dissected in Chapter 5. A double dashed line appears, declaring that `s_client` has finished its work, and the daemon responds to the connection.

```
+OK Gpop ready for requests from
   2001:db8::bad:c0de:cafe g12mb780908504jaq
```

I can use the raw POP3 protocol to see if the server is working correctly.

```
USER notmyrealaccount
+OK send PASS
PASS notmyrealpassword
-ERR [AUTH] Application-specific password required:
   https://support.google.com/accounts/answer/185833
```

The results transform my question to: why didn't my mail client report this error?

Web traffic is a little different. My web sites all redirect traffic from HTTP to HTTPS, and from the **`www`** version of the hostname to the bare domain—that is, **`http://www.mwl.io`** redirects to **`https://mwl.io`**. This enables a variety of attacks, but my site is neither confidential nor important. (I also enabled HSTS, discussed in Chapter 8.) After I reconfigure the server, I want to verify that those redirects are still functioning. Browsers cache all sorts of detritus, so I don't trust them. Instead, I want to interrogate the server directly. HTTP separates CR and LF, so I must use the `-crlf` option to `s_client`.

```
$ openssl s_client -verify_return_error -crlf www.mwl.io:443
```

If you're running an older OpenSSL or LibreTLS, use both `-crlf` and `-connect`.

```
$ openssl s_client -crlf -verify_return_error \
   -connect www.mwl.io:443
```

I get all the TLS information, and at the end there's a double dashed line.

```
---
```

Web servers do not offer protocol banners upon connection. I enter my HTTP commands.

```
GET / HTTP/1.1
Host: www.mwl.io
```

I hit ENTER once between the lines. HTTP requests end with two ENTERs, so when I've finished my commands I hit ENTER ENTER. It responds.

```
HTTP/1.1 301 Moved Permanently
Date: Wed, 09 Sep 2020 16:12:18 GMT
Location: https://mwl.io/
…
```

My redirect is intact.

The s_client command takes the SNI server name from the name of the host you connect to. If you must specify a different SNI server name, use the -servername option. Here, I want to be sure that my installation of my web site on a new server works, even though it's on a temporary hostname.

```
$ openssl s_client -servername www.mwl.io \
   -crlf newwww.mwl.io:443
```

Using separate port numbers for TLS versions of services works, but while network ports are numerous they are not infinite. Plus, the technique increases network traffic and slows down services. We need better.

### Opportunistic TLS

In some protocols, a client can request that a server use TLS and servers can request clients upgrade to TLS. The idea is to use TLS when it's available, but permit continuing if TLS isn't available. This *opportunistic TLS* was ideal for universal protocols like email, where everyone had a server and they couldn't all simultaneously upgrade. The expectation was that eventually, all applications would support TLS. Opportunistic TLS is often called *STARTTLS* after email's in-protocol command to request TLS, but not all opportunistic TLS implementations use the literal command STARTTLS.

"Opportunistic" sounds flexible, but a server using opportunistic TLS can inflexibly require TLS. When a client connects, the server

might request TLS. If the client refuses, the server can declare "Sorry, TLS or nothing" and disconnect.

Every protocol looks different, so each must implement opportunistic TLS differently. They're similar, but the SMTP protocol used for email doesn't resemble LDAP or XMPP or anything else. Before trying to use a server's opportunistic TLS with `s_client`, check the `-starttls` option in the man page to see if your implementation supports that protocol. Note the name `s_client` uses for your protocol.

We'll use email as an example.[8] The history of email and TLS has been complicated by email's evolution, the rapid development of desktop mail clients, web-based email, and decades of workarounds and lingering migration paths from carrier pigeon, UUCP, and worse. Email has selfishly claimed TCP ports 25, 465, and 587, and shows no willingness to surrender any of them. Here I use `s_client` to connect to a mail server on port 25, using STARTTLS.

```
$ openssl s_client -connect mail.mwl.io:25 \
   -starttls smtp
```

We use the familiar `openssl s_client` command, and the same `-connect` keyword with a host and port. The new bit is at the end. The `-starttls` option tells `s_client` to immediately negotiate a TLS session. We then give the protocol name, `smtp` in this case.

You'll get the usual spew of TLS information, followed by:

```
220 mail.mwl.io if you must
```

I can now engage in my usual SMTP-by-hand shenanigans.

```
EHLO mail.mwl.io
250 mail.mwl.io Hello vermin.isp
   [2001:db8::bad:c0de:cafe], pleased to meet you
MAIL FROM: mwlucas@michaelwlucas.com
550 5.7.1 Mail from 2001:db8::bad:c0de:cafe refused -
   see http://www.spamhaus.org/zen/
```

8 Email is a great example of a terrible example.

My RBL subscription works? That's good. Unfortunately, I was testing my RBL override for my home address.

While desktop mail applications care very much about the validity of server certificates, TLS connections between mail servers are perfectly content with self-signed certificates. In server-to-server mail, there's no human being involved to decide if the warning should be overridden or not, and email specifically doesn't care about the sending host's identity. That's okay. Server-to-server SMTP exists mostly to prevent large scale email capture by systems like the United States government's Carnivore. SMTP's TLS accepts self-signed and expired certificates on servers, because self-signed certificates suffice to prevent passive snooping on the wire. An attacker could still inflict a man-in-the-middle attack, which is much more challenging at scale.

Privacy-hostile networks, which include the entire Internet of certain countries, often block STARTTLS. They also block dedicated TLS ports, however.

### Connection Commands

The `s_client` session remains open until you terminate it. While `CTRL-C` is the time-honored method of breaking a connection, you do have other options.

Entering a `Q` cleanly closes the TLS connection.

The tricky ones are the commands that alter a live TLS session. Entering a `k` sends a TLS 1.3 key update, while a `K` both sends an update and requests one in return. If this connection uses an older TLS version, it will break the connection. Entering an `R` renegotiates a TLS 1.2 session.

Most people won't need these, but their mere existence can cause problems. What if the protocol you're using has commands that begin with these letters? I don't know of a protocol with a command that starts with `Q`, but every time you try to debug it with `s_client` OpenSSL will hang up on you.

The `-ign_eof` flag is intended to keep a TLS connection alive even after the end of any input. It has the side effect of disabling these commands.

If you use `s_client` and `s_server` to fling files around the network netcat-style, the `-ign_eof` flag is pretty much mandatory.

#### DTLS

Testing DTLS by hand resembles using other UDP-based protocols with netcat. It demands extensive knowledge of the underlying protocol. It's not something you do casually. SCTP is actively hostile to this sort of interactive use. If you're interested in playing with TLS over UDP or SCTP, many folks have written simple responders for them.

Also take a look at OpenSSL's `s_server` subcommand, which lets you create the equivalent of a netcat listener wrapped in DTLS. Most netcat variants have similar functions.

### *Silencing s_client*

Perhaps you know that the TLS is working and you just want to poke the daemon underneath. All of this low-level TLS information is only a distraction. Add the `-quiet` flag to silence everything except a summary of the certificate chain.

```
$ openssl s_client -quiet -verify_return_error \
  -crlf www.mwl.io:443
depth=2 O = Digital Signature Trust Co.,
  CN = DST Root CA X3
verify return:1
depth=1 C = US, O = Let's Encrypt,
  CN = Let's Encrypt Authority X3
verify return:1
depth=0 CN = mwl.io
verify return:1
GET / HTTP/1.1
…
```

To see a summary of the negotiated TLS characteristics, use `-brief` instead.

```
$ openssl s_client -brief -verify_return_error \
   -crlf www.mwl.io:443
CONNECTION ESTABLISHED
Protocol version: TLSv1.2
Ciphersuite: ECDHE-RSA-AES256-GCM-SHA384
Peer certificate: CN = blather.michaelwlucas.com
Hash used: SHA256
Signature type: RSA-PSS
Verification: OK
Supported Elliptic Curve Point Formats: uncompressed:
   ansiX962_compressed_prime:ansiX962_compressed_char2
Server Temp Key: X25519, 253 bits
GET / HTTP/1.1
…
```

I can now enter my HTTP commands and be told that this site has moved, without wading through all of the TLS detail.

### *Specific TLS Versions*

When a TLS client like `s_client` first connects to a server, the client lists the TLS versions it supports. The server picks the highest version it supports. For debugging you might need to force use of a particular TLS version, or disallow certain TLS versions for this connection. OpenSSL has command-line options for both. The `-tls1_3` flag means to only accept TLS 1.3, while `-tls1_2` forces TLS 1.2. If you're checking obsolete versions of TLS, there's also `-tls1_1`, `-tls1`, and `-ssl3`.

Suppose I configured my web server to only accept TLS 1.2 and 1.3. Web servers are complex. It's entirely possible that I missed something. I must verify my work.

```
$ openssl s_client -brief -ssl3 -crlf www.mwl.io:443
$ openssl s_client -brief -tls1 -crlf www.mwl.io:443
$ openssl s_client -brief -tls1_1 -crlf www.mwl.io:443
```

If all of these return errors, the connections were unsuccessful. These TLS versions don't work on my server. I should probably verify that TLS 1.2 and 1.3 work as well, however.

```
$ openssl s_client -brief -tls1_2 -crlf www.mwl.io:443
$ openssl s_client -brief -tls1_3 -crlf www.mwl.io:443
```

These flags also come on "no" versions that forbid a particular protocol. By adding `-no_ssl3`, `-no_tls1`, `-no_tls1_1`, `-no_tls1_2`, or `-no_tls1_3`, you can forbid `s_client` from using SSLv3, or TLS 1, 1.1, 1.2, and 1.3 respectively. Checking to see if a web site supports TLS versions other than 1.2 and 1.3 could be run in a single command, like so.

```
$ openssl s_client -brief -no_tls1_3 -no_tls1_2 \
  -crlf www.mwl.io:443
```

If this server supports any TLS version other than 1.3 and 1.2, `s_client` will find it.

Don't mix `-tls` and `-no_tls` flags. Either declare a protocol version, or ban protocol versions.

The `s_client` subcommand has many other options. Get the full list with the `-help` argument, or read the manual page.

### Choosing Ciphers

Maybe you're looking to emulate a specific client, or want to test if certain ciphers are available on a server. If you check your web browser's TLS connection details, or the log of the client program, you'll see the cipher name along with the TLS version. That cipher name might be the cipher's OpenSSL name, the IANA name, the GnuTLS name, or some other name. Use **`https://ciphersuite.info`** to find the OpenSSL cipher name, so you can specify that cipher in your `s_client` command. The `-cipher` option lets you specify TLS 1.2 options, while `-ciphersuite` lets you specify TLS 1.3 ciphers.

Setting a cipher list for a particular TLS version does not mean that `s_client` automatically uses that TLS version, however. It might. It might not. If you're testing a particular cipher, specify the TLS version as well.

Now let's consider certificates themselves.

# Chapter 3: Certificates

TLS is built on *digital certificates*. A digital certificate is a collection of carefully formatted information that identifies an entity, digitally signed by a Certificate Authority. Servers, services, and users can have certificates.

A *server certificate* is intended for an application server, and is used to authoritatively identify the server. This is the kind of certificate you'd install in your web server. Most sysadmins focus on server certificates.

A *client certificate* or *user certificate* identifies someone or something that authenticates to a server. A person might use a client certificate to authenticate themselves to a VPN, mail, or web server, using the certificate's passphrase much like a password. An internal mail server might have a client certificate that authenticates it to its public-facing outbound mail server—the internal mail server is a client of the outbound server.

A *trust anchor*, sometimes called a *root certificate*, is ultimately trusted to sign other certificates, as discussed later this chapter. Trust anchors might be globally trusted public CAs, or they might be private certificates like those used in Active Directory.

*Issuing certificates* can sign other certificates. An issuing certificate might be issued by a certificate authority, or they might be part of a private CA. Chapter 10 discusses name constrained issuing certificates.

Most of the examples in this book use server certificates. We'll also discuss client certificates, but most everything applicable to server certificates applies to user certificates. Chapter 10 covers creating CA certificates, issuing certificates, and more. If you encounter something special, like an OCSP validation certificate, either the tools you use to scrutinize server certificates work perfectly well on them or you'll be

able to leverage your knowledge to find the right command to crack it open. After all, all these certificates are built on the same standards.

### Certificate Standards

TLS does a decent job of concealing the standards that certificates are rooted in, but occasionally these roots poke up through the forest floor. If you don't recognize them, eventually they'll trip you.

TLS certificate information is organized as per X.509, the International Telecommunications Union's (ITU) standard for digital certificates. X.509 is used in many applications that demand precise formatting of data. It defines how certificates will be arranged, which features are used, how they're validated and revoked, and more.

X.509, in turn, is built on Abstract Syntax Notation One (ASN.1), another ITU standard. ASN.1 is a method for defining cross-platform data structures and providing information in a globally recognized manner. You'll see ASN.1 in protocols like LDAP and SNMP. TLS tools like OpenSSL automatically convert those numbers to human-friendly words. Fortunately, you don't need to understand the innards of ASN.1. Accept that they have been unfavorably compared to the Cliffs of Insanity by more than one developer and move on.

ASN.1 is built by arranging objects into a tree. Each branch and leaf on the tree is identified by a numerical *Object Identifier*, or *OID*. X.509 shares a tree with protocols like SNMP, but is on a different branch. New objects are slowly but constantly added to TLS certificates. Each CA has its own chunk of the object tree. Your software might not have all of the OIDs in a certificate. If a command produces output like 1.3.6.1.4.1.44947.1.1.1 followed by a lump of indigestible gibberish, you should think "Aha! This is a raw OID that my tools don't know how to process!" Check for software updates. CAs can also use private OIDs, such as those from 1.3.9900 to 1.3.9999. Much like 192.168.0.0/16 IP addresses, these aren't *supposed* to be visible beyond the organization, but occasionally escape.

If this wasn't enough, X.509 also pillages the X.500 directory standard. X.500 defines one- or two-letter labels for different types of information, such as `OU =`, `O =`, and `CN =`. O gives the *Organization* name, such as a company or other entity, while OU represents *Organizational Unit*, a division of that entity.

The CN gives the *Common Name*, which was historically a hostname in TLS. Using CN to store hostnames has been deprecated since 2000, as CN cannot support hostnames longer than 63 characters. The replacement for hostnames in CN is called Server Alternative Names (SAN), discussed later this chapter. Many CAs transparently copy hostnames from the CN to SANs. You'll occasionally see other labels that the CA felt obliged to stick in there.

The Common Name might be a uid, email address, first and last name, or some combination thereof. It could even be a serial number, identifying a device.

Each directory item has an OID. If you poke around, you'll see that the omnipresent CN is 2.5.4.3. You don't need to know the numbers, but don't be puzzled when they appear.

### *Trust Anchors*

A *trust anchor*, often called a *root certificate* is included on a list of ultimately trusted certificates. There's nothing special about any of these certificates. Your system or application has been told to trust them, so it does. Yes, it's highly arbitrary.

Many trust anchors are self-signed certificates from big organizations in the business of signing certificates. Some also bear signatures from other trust anchors, as discussed later this chapter. The only difference between a trust anchor and any other self-signed certificate is that applications have been told to trust it. Many trust anchors also have very long lifetimes, decades or more.

Every operating system or application platform provides a *trust bundle*, a collection of self-signed certificates trusted to

sign other certificates. It might be called a *trust anchor bundle*, *root certificate bundle*, or sometimes even *certificate bundle* or plan *bundle*. Vendors like Microsoft, Apple, and Google maintain their own bundles. The Mozilla Foundation maintains the bundle used in Firefox, but most versions of Unix also use the Mozilla bundle or a derivative thereof. Oracle and Adobe also have major trust bundles.

Not all of these organizations trust the same trust anchors, however. Different CAs have applied to different bundles, and not all CAs qualify to be in all roots. Projects like the Trust Stores Observatory monitor which trust anchors are in which bundles.

An organization can run its own internal certificate authority, as discussed in Chapter 10. Such CAs are trusted within the organization, but not by the outside world. It's easy to start and annoying to maintain. For many organizations it's a sensible solution, however.

Many experts, and a whole bunch of amateurs, object to the inclusion of specific entities in these trust anchor bundles. Governments and giant corporations need us to automatically trust them, but do *we* need to extend that trust? These root certificate bundles are used globally. Should a US citizen trust a CA run by the government of Hong Kong? Perhaps folks in the Netherlands should trust their government, but should they trust the United States government? If you trust them today, should you trust them tomorrow? Who the heck is HARICA or the Shanghai Electronic Certification Authority? You can find out. You can curate your own bundle of trust anchors that you consider trustworthy. More than one person maintains a "trust anchor bundle for the commoners," but how do you decide to trust the curator? What will you do when an application breaks because it can't validate a certificate? Will you deal with the political or technical fallout of not trusting that certificate, or would you re-add the trust anchor to your bundle and go back to watching right-handed swordsmen fight left-handed?

This problem underlies all discussions of certificate authorities. I won't belabor it any further. Make your decisions and live with them.

### Making Your Own Trust Bundle

Many huge enterprises maintain their own trust bundle. The business has made a decision that they don't trust specific governments or organizations, and/or they do want to trust their internal-only certificate authority. The trick is where they deploy that bundle.

Maintaining your own trust anchor bundle for web browsers is a vexing hobby. You have no way of knowing which CAs your external partners use, let alone random web sites out in the wider Internet. Outside entities have no obligation to warn you when they change CAs. Operating system updates can overwrite your carefully selected bundle. A curated browser trust anchor bundle is an exercise in frustration.

Other applications can absolutely use a curated trust anchor bundle, however.

Suppose your organization made an internal decision to only get certificates from "Miracle Max's CA." Max signs your web site certificates, but also your mail and LDAP servers, as well as any other TLS applications. Rather than having those clients read the web browser's trust bundle, configure them to use a trust bundle that contains only Miracle Max's CA certificates. This makes it much more difficult for an intruder to hijack your internal services. Perhaps the intruder can fool Guilder into issuing a bogus certificate for your domain—but if your applications don't trust Guilder, certificate errors will warn your users that something is wrong.

Or perhaps you build an embedded device that you install in all of your offices. This device should only communicate with your own infrastructure. Its trust bundle should include your private CA, but no other certificates.

### The OpenSSL Trust Bundle

OpenSSL must have a trust bundle to validate certificates. Most OpenSSL installs use the Mozilla trust bundle, distributed as a single file containing all the CA certificates, but each breed of Unix has chosen their own way to manage these certificates.

Most software expects to find certificates in `/etc/ssl/certs`. OpenSSL expects to find them in a `certs` subdirectory of the system OpenSSL directory (available by running `openssl version -a`). Some Unixes break Mozilla's big file into individual certificates. Others don't. Every Unix resolves these conflicts by applying symlinks until the complaints stop.

Each Unix also has their own way to manage these certificates. Maybe it's `certctl`, or `add-trusted-cert`, or `update-ca-certificates`, or a tangle of OpenSSL commands. If you need to add or remove certificates from the OpenSSL trust store, check your operating system documentation for the correct way to do so. Don't just fling files into what looks like the correct directory; a system update might well overwrite them.

If you want to use a specific CA certificate in an OpenSSL command—say, for testing a private CA—you can use the `-CAfile` option to point at the certificate file.

## *Certificate Components*

A certificate contains two primary pieces: the information about the entity being certified, and a digital signature of that information.

The information about the entity being certified is created by the entity requesting the certificate. That entity might be a sysadmin, or it could be an automated process like ACME. It might include details about host names, physical location, and the responsible organization. It might be only a hostname. The certificate also includes a public key. This organization information and the public key is gathered together into an X.509-formatted file called a *certificate signing request*

or *CSR*. You submit the certificate signing request to your Certificate Authority. You hide the private key like the treasure it is.

The CA verifies the information in the certificate signing request, as discussed in *Certificate Types* later this chapter. Some CAs offer extensive verification, ensuring that the request comes from the right organization and that it was submitted through approved channels. Others, like free ACME CAs, verify only that the entity that submitted the request controls the host and/or its domain. Once the CA is satisfied of the request's legitimacy it attaches any delegated permissions and digitally signs the whole thing. That's the certificate.

Certificates have validity dates, or at least an expiration date. Clients may reject expired certificates. Manually managed CAs generally offer one-year certificates, while ACME CAs most often use three months to encourage automation. Certificate safety is inversely proportional to certificate lifetime. I've seen multiple financial institutions use certificates that expire in twenty-four hours. Generally, use the fastest expiring certificates you can reliably renew. Additionally, the latest browsers reject certificates older than 398 days.

Using a certificate requires the certificate file returned from the CA, plus the private key created when you made the certificate signing request. The certificate is useless without both components.

### *Extensions and Constraints*

While the certificate's X.509 format rigidly defines data fields, it is also extensible. A certificate issuer can add their own rigidly defined data fields to certificates. Most of these extensions are used as constraints or policy statements.

A *constraint* dictates how the certificate can be used. One common constraint is a name constraint, where the certificate can sign other certificates within a certain domain. I might spend a bunch of cash on my own intermediate CA that could sign certificates, but only for my domain `mwl.io`. We discuss these certificates in Chapter

10. Some certificates might have constraints that they can only sign client certificates, or certificate revocation lists, or must use certain cryptographic algorithms.

A *policy* most often applies to aspects of the TLS connection. Most policies are built with X.509 extensions. A certificate could have a policy that it is valid only on certain domain names, or that it can only be used for client authentication.

Certificates mark each extension as either *critical* or *non-critical*. The client must process and validate all critical extensions. If it cannot validate a critical extension, the connection fails. Clients are expected to respect non-critical extensions, but if the client software doesn't recognize the extension, they have permission to skip them. That's no guarantee that the connection will work, however. Server Alternative Names (SAN) are non-critical, but any client that wants to operate on the modern Internet handles them.

Suppose a certificate has a constraint that it can only be used to sign certificates in the `mwl.io` domain. Name constraints are always critical extensions. I use it to sign a certificate for `microsoft.com`. The certificate violates the critical constraint. If the client does not know how to interpret that constraint, it will reject the certificate. If the client does know how to interpret the constraint, it will realize that the certificate violates the constraint and reject it. Either way, my limited signing certificate can't create a usable certificate outside my domain.

### *Validation Levels*

For most folks, the entire purpose of a certificate is to shut up a program's warnings about an unsafe application. Certificates give you the little lock or shield in your browser. Certain environments might require server certificates with greater level of trust than usual, however, as expressed by how well the owner is validated.

A domain validated (DV) certificate means that the CA verified that the requesting entity owns and controls the domain the certificate

is for. If I want a DV certificate for my domain `mwl.io`, I need to demonstrate that I own and control the host that the name `mwl.io` points at, or that I can add entries to the DNS. All free ACME and most inexpensive certs are DV.

An organization validated (OV) certificate includes everything in a DV certificate, but also checks that the requesting organization exists and is at the address claimed. Most commercial CAs offer OV certificates.

An extended validated (EV) certificate digs into the entity requesting the certificate. It verifies the organization's business registration and jurisdiction. This sort of certificate includes everything you need to track down the organization. These certificates can run thousands of dollars.

Each certificate type contains the information validated. A DV certificate contains only the domain name. An OV certificate includes basic organizational information, while an EV certificate gives you all sorts of data.

Most of us, in most cases, want our applications to quit complaining. A DV certificate suffices. Financial institutions often use certificates with OV validation. An application that receives an EV certificate might flag the user somehow; some browsers turn the address bar green on sites using an EV cert. In the real world, users almost never notice this—and when they do, they get alarmed. EV certificates are entirely about regulatory compliance, as they offer no technical benefits.

### *Trust and Your Certificate*

The whole point of a certificate is for an entity to prove its identity. When a server or a client receives a certificate from the other side of the connection, it must validate that certificate. Historically, certificates were validated using a Chain of Trust. Today, it's more like a Tree of Trust.

Our examples assume that a TLS client, like a web browser or email client, is validating a server's certificate. Client certificates work the same way.

### The Chain of Trust

Back in the day, a sysadmin issued a Certificate Signing Request containing their organization and server information. A Certificate Signing Request was basically a certificate without the digital signature. The sysadmin sent the request to a Certificate Authority, along with a wad of cash. The CA signed the Certificate Signing Request, creating a complete certificate, and returned it to the sysadmin, who installed the certificate file on their server.

In those years, the Chain of Trust looked something like this.

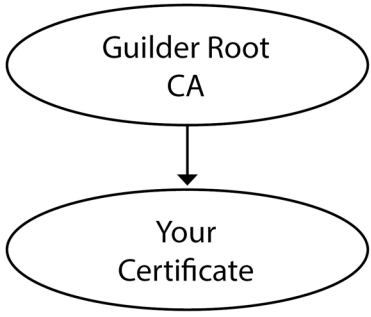


*Figure 1: a primordial Chain of Trust*

A single link. Easily validated. Very simple.

When a client visited the web site, they would first validate the certificate. The certificate claimed to be signed by a particular root CA. The client trusts that root CA. The client would validate that signature and proceed—or reject the signature and stop.

### Intermediate CAs

Root CAs operate under very strict requirements. The private key is often kept tightly locked up and can only be accessed by select corporate officers. Select corporate officers dislike interrupting

their pleasures to carry out their corporate responsibilities, when they should rightfully delegate all the routine labor. That's where *intermediate CAs* come in.

An intermediate CA has a certificate signed by a trusted CA, so clients will trust it in turn. It has a shorter lifetime than the trust anchor—if nothing else, you don't want the trusted CA certificate to expire while the intermediate CA or any certificates signed by it are still in use. The intermediate CA certificate has been delegated the privilege of signing further certificates. You now have a Chain of Trust with multiple links, like a real chain.

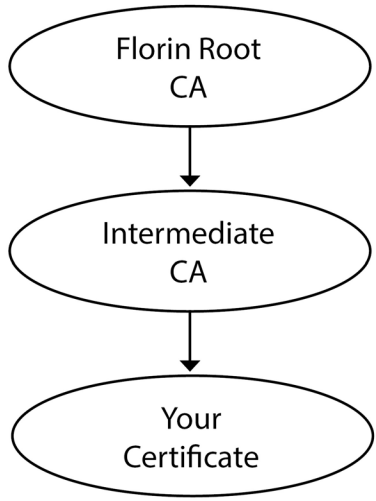


*Figure 2: Chain of Trust with an intermediate CA*

Clients visiting your web site validate your certificate by evaluating the signature on your certificate, and then the signature on the certificate used to sign your cert, then the CA's certificate. If everything matched, your certificate was valid.

Intermediate certificates can have additional constraints put on them. Humperdinck Unlimited might buy their own intermediate CA certificate that can sign any certificates in their domain. They might create further intermediate CAs for, say, the North American, European, and Asian subdomains, and delegate signing authority to them.

One complication with Chains of Trust is that the client only has the trust anchor. It does not have the intermediate certificates, nor any way of collecting them. Without those, validation fails. The server must offer the client the certificate chain. A *chain file*, sometimes called a *CA bundle*, includes any and all intermediate certificates. They sometimes include the trust anchor, although that's unnecessary. A *full chain* file also includes the end host's certificate. CAs that use intermediate certificates offer chain files.

If your CA has an intermediate certificate and your server does not offer clients a chain file, clients cannot validate your certificate. Period.

In those good old days, clients might have to validate a Chain of Trust seven or eight links long. It's not so easy now.

### The Tree of Trust

The thing about trust? Not everybody has it.

While most of users are content to trust whatever root CAs ship with their software, some organizations distrust certain CAs. A government entity might refuse to trust certain CAs from other nations. Some companies might distrust their competitor's CA.[9]

After years or decades, trust anchors expire. A trust anchor included with SSL 1.0, using algorithms considered unbreakable in 1995, should certainly not be trusted today. Software that receives regular updates will get newer certificates, but that's not always realistic. Even sysadmins fiercely adamant that all user software must be updated often bear responsibility for decrepit mission-critical systems accessible only via Internet Exploder 6.

But what happens if a trust anchor is compromised? A trust anchor cannot be revoked, as we'll discuss later. That certificate would be removed from the list of trusted certificates, however. After updating, clients would stop trusting that certificate.

---

9 Windows repeatedly asks if I'll always trust software from Oracle. I laugh derisively. Every. Single. Time.

*Cross-signing* permits certificates to carry more than one signature. So long as the client can find one chain of valid signatures leading to a trusted root, the client trusts the certificate. Cross-signing creates something like Figure 3.

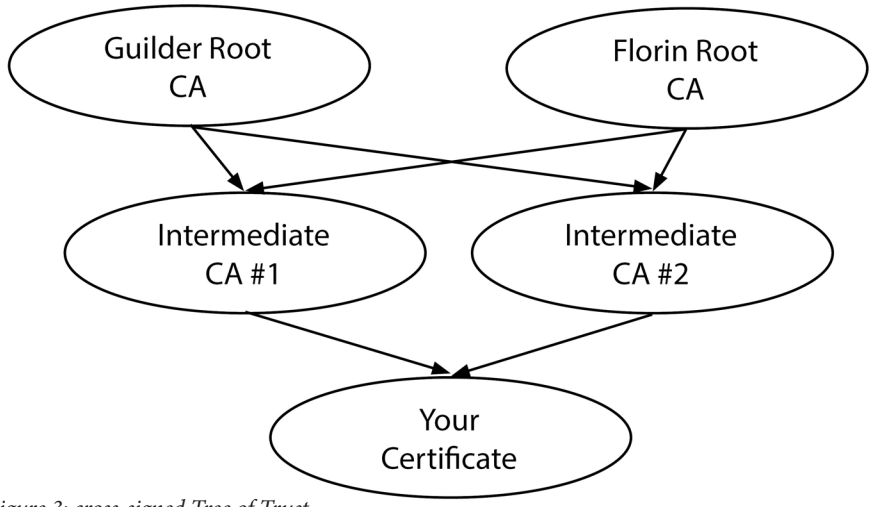


*Figure 3: cross-signed Tree of Trust*

The whole goal of this is to make sure that your certificate is valid, no matter what happens further up the tree. If Guilder's trust anchor gets ripped from the list of trusted certificates, the client can validate the certificate against Florin's trust anchor. If an intermediate CA's certificate is revoked, the other CA provides valid links to a root CA. You could lose all but one trusted entity on each level and still validate the certificate.

This is a very simple cross-signed certificate setup. Several roots might sign any of a combination of intermediate CAs, which in turn all sign your certificate. You might have countless layers and innumerable roots all funneling incredible torrents of trust down to your hapless certificate. You might discover an intermediary certificate in your trust bundle, but the certificate it was signed with has expired. The Tree of Trust has so many bizarre twists and cross-connections it could more reasonably be called the Twisted Tangle of Trust, but

that would reduce confidence in the protocol underlying all modern industry and commerce so we won't do that.

While a Tree of Trust is cryptographically reliable, not all TLS clients can successfully validate one. Many developers only paid full attention to the first Chain of Trust found, neglecting the whole rest of the Tree. These applications fared badly as the Tree of Trust grew ever more complex. Once trust anchors widely used in cross-signed certificates began expiring in the late 2010s, application developers hurried to catch up. Intermediate certificates became trusted trust anchors, but validation software expected the root to be at the top of the tree, not in the middle. When the original trust anchor expired, certificates signed by the new root failed validation. Many of the necessary software improvements are still in progress, but the Tree of Trust constantly evolves towards a Tangle of Terror.

This race will never end.

Even if the client software can validate a complex Tree of Trust, it will not have the intermediate certificates. Your application must provide complete chains. Most servers can accept a complete chain file as the certificate.

If your software tells you that the Chain of Trust isn't trusted, but examination of the certificate shows multiple intermediaries and roots, your software needs an update or perhaps flat-out repairing. If your vendor has no update, you need a new vendor.

### *Certificate Validation*

Clients must validate server certificates. The full certificate validation process is baroque and cumbersome, with a whole bunch of exceptions and loops and dead ends. If you require details, RFC 5280 contains enough of them to destroy any hope that true love exists. We'll cover the common case.

This discussion assumes that an application server such as a web site or mail host provides the certificate, and a client like a browser or

desktop mail program must validate it. Some applications require the client provide a certificate to the server. Perhaps both the client and the server offer certificates. While the protocol internals differ, both sides follow highly similar procedures. We discuss the "TLS client" and "TLS server" separately from the application client and server.

Any time a TLS connection fails, the client presents the user with an error message describing why. In most applications, the user can choose to accept the certificate anyway and proceed. You've seen these messages in your web browser, along with cryptic notices that declare you still have much to learn about TLS. Your users might override the error and continue, or call the helpdesk and complain that the Internet is broken.

Upon making the TLS connection, the server presents its certificate and any intermediate certificates to the client. The client inspects those certificates, checking both the validity dates and the digital signatures. If the signatures do not validate, the certificate was either tampered with or damaged in transit. If the certificate isn't yet valid, or has expired, it's invalid. Either way, the TLS connection fails.

If the certificate is intact and has good dates, the client attempts to find a path between the server's certificate and the client's trust anchors. If the client can't find a path, as discussed in "Trust and Your Certificate," the certificate is rejected and the TLS connection fails.

The client then checks to see if the certificate has been revoked. We discuss revocation and all its variants, like OCSP, in Chapter 4.

If the certificate looks good so far, the client then checks the constraints and extensions within the certificate. The critical extensions must validate, or the client rejects the certificate.

All of this looks straightforward enough. And it is, so long as everything works. What does your particular client do if the Certificate Revocation List (as discussed in Chapter 4) is unavailable, either from an outage or because the CA screwed up or because

the CA didn't bother to update the list this week? Each application developer made choices on how to handle failure. If a web site throws a confusing error message in the usual browser, but another browser shows it just fine, the user won't care that the real problem is that the OCSP staple is stale and that the web site is untrustworthy. They only care that their old browser is a complete failure and that their new favorite browser works.

### Encoding

While certificates all use the X.509 format, a system can *encode* and store those certificates in many ways. Some software accepts only one specific encoding. You must be able to identify different encodings and convert between them as needed.

An *encoding* is a method of arranging data. It has nothing to do with encryption, ciphers, or secret codes and everything to do with serializing objects for transmission and storage. ASCII is an encoding. Microsoft Word files have an encoding. Both the paper and electronic versions of this book are encoded. Very few of us can read binary or base64[10] encoding by eye, but to software all of the encodings it understands are interchangeable. You don't have a PEM TLS certificate, you have a TLS certificate encoded in PEM format. If the encoding doesn't fit your needs, change the encoding.

While Unix doesn't consider filename extensions meaningful, suffixes are useful for the human beings managing the systems. You'll see extensions like `.pem`, `.der`, `.crt`, and more. Unfortunately, TLS seems to be the area where sysadmins treat such extensions with even more disdain than usual. More than once I have been asked to troubleshoot servers where the sysadmin gave all TLS-related files names ending in `.crt`, regardless of the file's contents or the encoding used. You must be able to differentiate between encodings using more than just the filename extension.

10 Remember, base64 is not a password hashing method.

### Distinguished Encoding Rules (DER)

The *Distinguished Encoding Rules* (*DER*) is one of the oldest methods of encoding X.509 certificates. DER is a subset of the *Basic Encoding Rules for ASN.1*, or *BER*. DER gives a unique and unambiguous encoding of any ASN.1 object. Every system that supports DER will interpret this data in precisely the same way.

DER is a binary format. Everything is encoded with a tag, a length, and then the data. A purist might note that everything in a computer is zeroes and ones, but DER plunges to that level faster than most. Commands like file(1) often identify DER-encoded files as "data." You can't copy and paste a DER certificate or send it in the body of an email. To treat a certificate as text, you must convert it to PEM.

A filename ending in `.der` hints that the contents should be DER-encoded. The only way to be sure is to use OpenSSL's `x509` subcommand to view the certificate.

```
$ openssl x509 -in file.der -inform der -text -noout
```

The `-in` option lets you specify a file to be read. Here we're reading `file.der`.

The `-inform` option doesn't sent your subversive activity to the authorities. It's short for "incoming format." Software often can't easily identify pure data formats like DER, so you must tell OpenSSL how to interpret it.

The `-text` flag tells OpenSSL to provide the output in human-readable text form. Readability does not imply comprehension.

Finally, `-noout` prevents display of the encoded certificate.

DER-encoded files are small, and are often used in space-sensitive applications like distributing Certificate Revocation Lists (Chapter 10).

### Privacy-Enhanced Mail (PEM)

The IETF created a standard for sending encrypted email, Privacy-Enhanced Mail (PEM), back in 1993. PGP defeated PEM in the Email Cryptography Swordfight, but PEM encoding proved suitable for keys and certificates and survived.

PEM is mostly base64-encoded DER, plus headers and footers to make it more human-friendly. You've probably seen certificates or keys encoded in PEM format; they start with a line of dashes and the words BEGIN CERTIFICATE or BEGIN RSA PRIVATE KEY. You then have a screen full of characters and symbols, followed by a line declaring END CERTIFICATE or KEY. PEM encoding permits including multiple items in a single file. I'm biased towards PEM over DER, if only because I can eyeball the contents and catch egregious mistakes, like trying to use a key as a certificate.

Files containing PEM-encoded items often end in `.pem`, but you'll also see `.crt` for files containing a certificate, or `.key` for public or private keys. The nice thing about PEM encoding is that you can read the headers and see what's in it.

View a PEM-encoded certificate with `openssl x509`.

```
$ openssl x509 -in file.crt -noout -text
```

The `-in` option lets us specify the file to read. The `-noout` flags blocks displaying the certificate, while `-text` says to produce human-readable text output. OpenSSL defaults to PEM, so there's no need to manually set `-inform`.

Key files are mostly random data. If you try to read a PEM-encoded key with `openssl x509`, OpenSSL will tell you it's not a certificate.

### Converting Between Encodings

Some software only accepts one encoding, and some CAs only provide certificates in the wrong format. You must be able to convert between encodings.

Re-encoding uses the `-in` option we used to view certificates. It also needs `-out` to give a destination filename. We'll also use `-inform` and `-outform` to tell OpenSSL how to read DER files and what encodings to use.

Here I convert the DER-encoded certificate *`www.der`* to PEM-encoded *`www.pem`*. The `-inform` flag tells `openssl x509` that the source file is in DER format. The `-outform` flag lets us choose PEM for our output.

```
$ openssl x509 -in www.der -inform der -outform pem \
   -out www.pem
```

Converting PEM certificates to DER is slightly simpler. OpenSSL can automatically identify incoming PEM-encoded certificates, so we don't need the `-inform` option. We still need `-outform der` to give a target encoding.

```
$ openssl x509 -in www.crt -outform der -out www.der
```

You can now feed your narrowminded software whatever encoding it requires.

#### OpenSSL Without Input Files

If you don't specify an input source with `-in` or `|`, OpenSSL takes input from standard input. You can run a command and paste input into it.

Suppose I want to convert a PEM certificate into DER format. I could run OpenSSL like so.

```
$ openssl x509 -outform der -out cert.der
```

The command would hang, awaiting input.

I then go to another terminal and copy my PEM certificate, including the ---BEGIN CERTIFICATE—header and the footer. I paste that into my command window and hit `ENTER`. The command takes that input and produces my DER file.

## *PKCS #12*

The Public Key Cryptography Standards (PKCS) from RSA Laboratories defines methods for storing, arranging, and using cryptographic information. These standards were first created in the early 1990s, and most of them relate to the inner workings of public key encryption and key exchange. Many of them were later republished as other standards: PKCS #10 defines CSRs, and is also available as RFC 2986. PKCS #12, however, defines how to store multiple related encryption files in a single archive file. This archive can be encrypted and digitally signed. It's most commonly used to store a certificate chain and its private key. Each file goes in a separate container in the archive, called a *SafeBag*. You might use PKCS #12 to email a private key and its certificate to a colleague in your organization. So long as you use a good password and send that password out of band[11], it's fairly safe.

Java uses PKCS #12 heavily. It's the default key storage format as of Java 8, which has added several features that make it not entirely compatible with other PKCS software. If you need a program to interoperate with Java, carefully test all operations.

PKCS #12 archive files usually have a file suffix of `.p12` or, sometimes, `pfx`. (The PFX file format was a precursor to PKCS #12.)

You need to be able to stuff assorted files in a PKCS #12 archive, view the contents, and extract those same files. Use `openssl pkcs12` for all these functions.

### Creating a PKCS #12 File

I want to store the private key `privkey.pem`, the certificate `cert.pem`, and the chain file `chain.pem` in a single encrypted PKCS #12 archive. Use `openssl pkcs12 -export` to accomplish this. The `-out` flag

11 "Out of band" means "pick up the phone and call them." Yes, the world contains worse things than OpenSSL.

lets you set the name of the PKCS #12 archive file. Use `-inkey` to give the private key, `-in` for the certificate file, and `-certfile` for any additional certificates.

```
$ openssl pkcs12 -export -out site.p12 \
   -inkey privkey.pem -in cert.pem -certfile chain.pem
```

The file needs a password. You must enter it twice, for verification.

### Viewing a PKCS #12 File

Before shipping the encrypted PKCS #12 file to your colleague, double-check that it contains your key. The `-info` argument displays the contents on your screen.

```
$ openssl pkcs12 -info -in site.p12
Enter Import Password:
```

Enter the password, and you'll see information about each SafeBag and its contents.

```
Bag Attributes
   localKeyID: 79 0D 40 5E A0 CC 45 DD D0 0E 03 C4 05 DC
   F9 FF 11 9E 4B 03
subject=CN = mwl.io

issuer=C = US, O = Let's Encrypt, CN = R3

-----BEGIN CERTIFICATE-----
MIIGwzCCBaugAwIBAgISBNkUzXsbmLOfPbh9m63mEqG6MA0G...
…
```

This archive contains multiple certificates. Each went in its own SafeBag. You'll see metadata for each SafeBag, then the certificate in the bag.

When a PKCS file contains a private key, OpenSSL defaults to showing that key in encrypted form. You'll be asked to enter a private key encryption key before displaying the key. If you want to see the key unencrypted, add the `-nodes` flag to your command line.

### Exporting From PKCS#12 Files

As you might expect from all the other OpenSSL commands we've used, exporting to a file works much like viewing on the screen. Give the archive file name with `-in`, and the output filename with `-out`. You'll need the PKCS password to open the archive. As with viewing the file, `openssl pkcs` will ask you for a passphrase to encrypt the private key. Add `-nodes` to leave the private key unencrypted.

```
$ openssl pkcs12 -in site.p12 -out all.crt -nodes
```

The file *`all.crt`* contains the extracted version of everything in the PKCS archive.

Realistically, you don't want the certificate and the key in a single file. You probably want the certificate in one file and the private key in another. This file will contain the SafeBag metadata for each certificate. You'll probably need to remove those lines before deploying the certificate.

```
$ openssl pkcs12 -in site.p12 -out certs.crt -nokeys
```

Use `-nocerts` to extract only the private key, or `-nokeys` to extract only the certificates. Again, add `-nodes` to leave the private key unencrypted.

```
$ openssl pkcs12 -in site.p12 -nodes \
   -out private.key -nocerts
```

Looking at the exported key, you'll see identifying information much like an exported certificate. But there's another, more subtle difference.

```
Bag Attributes
   localKeyID: 79 0D 40 5E A0 CC 45 DD D0 0E 03 C4 05 DC
   F9 FF 11 9E 4B 03
Key Attributes: <No Attributes>
-----BEGIN PRIVATE KEY-----
MIIJQQIBADANBgkqhkiG9w0BAQEFAASCCSswggknAgEAAoICAQDW…
```

This looks perfectly fine at first glance. You chop off the bag details and have a key file.

This PEM-encoded certificate starts with BEGIN PRIVATE KEY. This is technically known as the PKCS #8 format. Most private keys use the PKCS #1 format, where the PEM-encoded file begins with a line that identifies the encryption algorithm, such as BEGIN RSA PRIVATE KEY. The example above happens to be an RSA key, but software that explicitly checks for "RSA" or "ECDSA" in the header will choke. You can transform the key format during export by piping it through `openssl rsa` (for RSA keys) or `openssl ec` (for ECDSA keys). Use the `-out` argument to give a filename.

```
$ openssl pkcs12 -in site.p12 -nodes -nocerts \
  | openssl rsa -out rsakey.pem
```

OpenSSL supports several other rarely-used PKCS options. Check the man page for details.

### *Certificate Contents*

Using `openssl x509` to view a certificate is easy, but what's all the crud *in* the certificate? Grab a certificate file and view it with the `-in` flag, like I do here with the PEM certificate *`www.crt`*. Add `-text` to show the output in text, and `-noout` to not show the encoded certificate.

```
$ openssl x509 -in file.crt -text -noout
```

This spills out the certificate contents. The first line always says *Certificate:* and the second *Data*.

```
Certificate:
  Data:
```

The meaningful stuff comes next.

```
Version: 3 (0x2)
Serial Number:
   03:c9:0d:76:bc:e1:a5:da:a4:70:3a:7d:ab:39:70:11:48:e0
Signature Algorithm: sha256WithRSAEncryption
Issuer: C = US, O = Let's Encrypt, CN = Let's Encrypt
Authority X3
```

*Version* gives the X.509 certificate versions of this certificate. Version 3 is the most common for TLS certificates, because it supports extensions that many Internet applications rely on. The rarely used version 2 is for "attribute certificates" and irrelevant to TLS. If a certificate does not have a Version field, it's a version 1 certificate. Version 1 mostly gets used in non-TLS applications.

The *serial number* uniquely identifies this certificate among all certificates signed by this CA. You could purchase multiple certificates for the same host, but they will have different serial numbers if they all come from the same CA. If you need to revoke a certificate, you might need this number.

*Signature Algorithm* tells how the CA signed this certificate. Each legitimate combination of hashing and encryption algorithms has its own code, but they're mostly self-evident. This certificate was signed using *sha256WithRSAEncryption*, or SHA256 hashing protected with RSA encryption.

The Issuer identifies the CA. The information given here is in X.509 format. `C` gives the country, `O` the organization. The `CN` label gives the Common Name for this entity. This certificate was signed by the US-based company Let's Encrypt, running the CA "Let's Encrypt Authority X3."

```
Validity
   Not Before: Jun  7 03:41:14 2020 GMT
   Not After : Sep  5 03:41:14 2020 GMT
Subject: CN=mwl.io
Subject Public Key Info:
   Public Key Algorithm: rsaEncryption
     RSA Public-Key: (2048 bit)
   Modulus:
     00:c8:88:be:30:04:f1:ad:3f:c3:5d:2e:fc:3b:c3:
…
   Exponent: 65537 (0x10001)
```

*Validity* gives timestamps for when this certificate becomes valid and when it expires. This certificate is good for 90 days. When something

or someone reports a problem with certificate validity, check the clocks of everyone involved.

The *Subject* is the primary identity of the entity being certified. It is often called the *Distinguished Name*. OV and EV certificates include country, city, and organizational information as well as the CN, while DV certificates offer only CN. This certificate still offers the hostname in CN, to support older client software. This certificate is for my web site, **`mwl.io`**. Remember, a certificate does not necessarily need a hostname in its CN; it could be for a UID, email address, or something else.

The *Subject Public Key Info* provides the details of my host's public key. This is not the CA's key, but the public key generated on my host when I requested a certificate. *Public Key Algorithm* identifies the algorithm used and the bit size. The *modulus* and the *exponent* are the public key's numerical components.

### Certificate Extensions

A certificate's *X509v3 extensions* lists all of the extensions included in the certificate. We discussed extensions in "Extensions and Constraints" earlier this chapter. Extensions not explicitly declared critical are non-critical. Remember, software must comply with critical extensions or it rejects the connection. Programs are expected to comply with all non-critical extensions they understand, but if they don't understand a non-critical extension they can try to continue.

Not all certificates offer all extensions, and not all certificates will have all of the extensions shown in this example.

```
X509v3 extensions:
  X509v3 Key Usage: critical
    Digital Signature, Key Encipherment
  X509v3 Extended Key Usage:
    TLS Web Server Authentication, TLS Web Client Au-
    thentication
  X509v3 Basic Constraints: critical
    CA:FALSE
```

We discussed extensions in "Extensions and Constraints" earlier this chapter. The *X509v3 extensions* section lists these extensions.

The *X509v3 Key Usage* extension declares how this certificate can be used for low-level TLS and encryption options. This certificate can be used to create digital signatures and exchange short-lived encryption keys, both critical components of TLS.

The *X509v3 Extended Key Usage* extension gives a higher-level view of how the certificate can be used. The server can use this certificate to identify itself as either a client or a server. While the description declares it to be for web servers alone, this certificate would work for SMTP or IMAP or any other TCP/IP service you might wrap in TLS. Note that this isn't marked critical, so applications are free to ignore it if they wish.

The *x509v3 Basic Constraints* extension identifies if this certificate is a Certificate Authority root certificate. If CA is FALSE, this certificate can't sign other certificates. If it's TRUE, it's a CA certificate. CA certificates also have a `pathlen` parameter, which shows the number of CA certificates signed by this cert that could appear beneath this one in a chain. Certificate Authorities use `pathlen` to create their intermediate CAs. This extension is critical, and must be obeyed.

```
X509v3 Subject Key Identifier:
   A2:37:C8:8D:78:75…
X509v3 Authority Key Identifier:
   keyid:A8:4A:6A:63…
```

The X509v3 key identifiers are public keys. The X509v3 Subject Key Identifier is the public key on this particular certificate. The X509v3 Authority Key Identifier is the signer's public key. While these are non-critical extensions, they're vital in finding a path to a trust anchor.

```
Authority Information Access:
   OCSP - URI:http://ocsp.int-x3.letsencrypt.org
   CA Issuers - URI:http://cert.int-x3.letsencrypt.org/
```

The *Authority Information* extension declares how the CA offers further information about this certificate. This Certificate Authority offers *OCSP* (Chapter 4). The CA Issuers field points to a collection of all the CA's issuing certificates.

```
X509v3 Subject Alternative Name:
   DNS:cdn.mwl.io, DNS:mwl.io, DNS:www.mwl.io
```

A certificate can be valid for multiple hosts. The *X509v3 Subject Alternative Name* field lists all valid host names. This is the modern standard way to get the certificate's names, replacing the Common Name. Note that this extension is not flagged as critical; clients can ignore it if they wish. This certificate represents three host names: **`cdn.mwl.io`**, **`mwl.io`**, and **`www.mwl.io`**.

```
X509v3 Certificate Policies:
   Policy: 2.23.140.1.2.1
   Policy: 1.3.6.1.4.1.44947.1.1.1
     CPS: http://cps.letsencrypt.org
```

The *X509v3 Certificate Policies* extension describes the CA's organization, operational controls, and procedures. You can get more information at the Certification Practice Statement (CPS) site shown.

If you're interested in a particular extension, you can extract it from a certificate using the `-ext` option to `openssl x509`. You must provide the extensions of interest in a comma-delimited list.

```
$ openssl x509 -in mwlio.cer -noout \
   -ext keyUsage,extendedKeyUsage
```

Get the complete list of extensions from x509v3_config(5).

### Certificate Transparency

Certificates are public information. Thanks to *certificate transparency* (Chapter 9), you can look up all the certificates issued for a domain. The Signed Certificate Timestamp (SCT) provides cryptographic proof that the certificate was submitted to a certificate log.

```
CT Precertificate SCTs:
   Signed Certificate Timestamp:
     Version   : v1 (0x0)
     Log ID    : 07:B7:5C:1B:E5:7D:68:FF:F1:B0…
     Timestamp : Aug  6 04:41:10.980 2020 GMT
     Extensions: none
     Signature : ecdsa-with-SHA256
     30:44:02:20:5C:B4:6C:B9:1B:0E:77:80:12:D8:…
   …
```

The timestamp tells you when the log signed and returned the certificate to the CA. This is your guarantee that the CA offers certificate transparency.

Eventually, TLS clients will reject certificates that were not submitted to a public log.

#### Digital Signature

At the very end of the certificate, you'll see the digital signature.

```
Signature Algorithm: sha256WithRSAEncryption
   05:a1:e2:2b:49:44:af…
```

This certificate was signed using the SHA-256 algorithm and encrypted with RSA.

#### Incomprehensible Certificate Information

If you poke at enough certificates, you'll eventually see something like this amidst a certificate.

```
1.3.6.1.4.1.11129.2.4.2:
   ......v....Y....@.-/....K..G…
```

The 1.3.6.1.4.1.11129.2.4.2 is a raw ASN.1 Object Identifier. It's followed by its value. Your OpenSSL tool doesn't know how to interpret this OID or its value. It's probably a new X.509v3 extension, or a private one the CA will never explain to you.

Ideally, updating your OpenSSL software will get you an interpretation of this OID. If you can't update, or if no update is available yet, or if your software hasn't yet added this OID, you can

perform an Internet search on this OID to, perhaps, unearth what it represents.

#### Skip Keys and Signatures

Many times you want details from a certificate, but you have no need to examine the public key or the digital signature. There's no need to display them.

The `-certopt` option lets you specify how to display a certificate. Here I tell `openssl x509` to skip the public key and the digital signature when showing the certificate.

```
$ openssl x509 -in mwlio.cer -text -noout \
   -certopt no_pubkey,no_sigdump
```

The x509 man page lists additional ways to tailor the output with `-certopt`. Is it easier to type those extra characters, or scroll up past the signatures? That's up to you.

### *Multi-Name Certificates*

Many sites use multiple host names. My own site is at **`mwl.io`**, but if someone makes the perfectly understandable decision to put a **`www`** in front of that, I want them to reach my site without getting any scary TLS errors. My certificate claims to be legitimate for both **`mwl.io`** and **`www.mwl.io`**. This is not at all uncommon, as you'll see if you examine the details of certificates used at big web sites.

A modern TLS certificate uses *Subject Alternative Names* or *SANs* to identify all of the hostnames the certificate is good for. SANs are certificate extensions, just like constraints and policies.

To view all of the SANs in a certificate, use the `-ext` flag

```
$ openssl x509 -in mwlio.cer -noout -ext subjectAltName
X509v3 Subject Alternative Name:
    DNS:cdn.mwl.io, DNS:mwl.io, DNS:www.mwl.io
```

For complete details on extensions, and everything you can do with SANs, read x509v3_config(5).

Include all SANs in the initial certificate signing request (Chapter 6) unless your CA declares otherwise.

### *Wildcard Certificates*

Not only can you have multiple hostnames in a certificate, you can have one of those hostnames be a wildcard. For my domain `mwl.io`, I could get a certificate that includes`*.mwl.io` as a Subject Alternative Name. I could install this certificate and its private key on every host in my domain, and the certificate would work.

At first glance, this seems great. Why wouldn't I do this everywhere?

It is great—when everything works. If one of my hosts is compromised and the private key stolen, however, my whole organization is at risk. The intruder could masquerade as any of them. If the intruder wanted to be especially sneaky, they would set up a new machine with a hostname that doesn't exist on my network. None of my customers would be even vaguely surprised to see `www2.mwl.io`, but that's not a real site.

If you use commercial certificates and have many hosts, you might consider using wildcard certificates to save money. Many huge enterprises do. Be aware of the risks before doing so, however. I see many companies restricting wildcards to select parts of their network. They might not have a `*.company.com` certificate, but instead use `*.api.company.com` and `*.images.company.com` to cope with the fluxes of dynamic host provisioning.

Wildcard certificates are headed towards obsolescence. ACME provides a standard interface for issuing and renewing individual certificates on public-facing hosts. Private CAs can use the same techniques for their hosts. Some organizations have built substantial automation around their wildcard certificates, however, and would rather not rip it all out.

### Viewing Remote Certificates

All our examinations has been run against certificates in local files, but sometimes you want to view the certificate being offered on a remote server that you might or might not control. Suppose your users complain that their browsers are spitting warnings when they visit my web site, and you decide to take a look. You could open your Web browser and burrow through menus until you find "View Certificate," but why do that when you have a convenient command prompt?

Start with OpenSSL's `s_client` subcommand, discussed at length in Chapter 2. Here I run `openssl s_client` to grab the certificate, and pipe that certificate into `openssl x509` to interpret it.

```
$ openssl s_client -connect www.mwl.io:443 < /dev/null \
  | openssl x509 -text -noout \
  -certopt no_pubkey,no_sigdump
```

This spills the certificate information across your screen. Hit `CTRL-C` to interrupt the network connection.

If you want to write the certificate information to a file, add the `-out` option and the filename.[12] You'll see a bit of TLS debugging on the screen, and will still need to CTRL-C out of the connection. You can now study the certificate as much as you want.

To view all of the certificates in the Chain of Trust, use the `-showcerts` option of `s_client`. The `x509` subcommand won't parse all of these certificates simultaneously, so you'll need to save this to a file and analyze them separately.

```
$ openssl s_client -showcerts -connect mwl.io:443 \
  > mwl.chain
```

Cut the certificates into separate PEM-encoded files and you can view them.

12 Yes, this means you need both `-out` and `-noout`. Float away on wings of paradox.

### Choosing a CA

For certificates facing the public, you must use a CA included in the major trust bundles. Many companies offer certificate authority services, both free and commercially. Which should you use?

Do you need to purchase a certificate, or will a free one suffice? If you're buying one, do you need an increased validation level or are you buying merely because the company doesn't trust free stuff? I've worked for organizations that sincerely believed a thousand-dollar OV certificate was superior to a functionally identical hundred-dollar OV certificate, specifically because it cost more.[13]

Make a list of candidate CAs that provide the type of certificates you want. Once you eliminate wholly unsuitable companies, a bewildering variety of marginally suitable ones remain. Go through and ask questions to rule them out.

If you want ECDSA certificates or wildcard certificates, verify that the candidate CAs support them.

Consider location. CAs are bound by the laws of their home nation, as well as any treaties. If you're setting up an ecommerce site for your wine shop, you probably don't care. I know people doing sensitive work that can't trust any organization in a specific country, or any country that's signed certain treaties. They want to get even their free certificates from outside those countries. Who does your organization trust?

How many certificates must you manage? If you need one certificate, you can put up with a clunky web interface. An organization that needs hundreds of certificates must have a CA that supports fully automatic certificate management. Managing many certificates when the CA demands you manually paste certificate signing requests into a browser window is a full-time career, and nobody qualified to manage certificates is willing to take such a tedious job.

13 Telling your boss that you'll buy the thousand-dollar certificate, purchasing the hundred-dollar cert, and using the difference to sponsor the sysadmin team's Wine Friday, makes you a Bad Person with everyone but your teammates. Sorry.

Your CA should support Certificate Revocation Lists (CRLs) and Online Certificate Status Protocol (OCSP) as discussed in Chapter 4. In theory all CAs have CRLs, but some provide only empty lists. The CRL is available on a web site, available in any certificate signed by that CA. Check that it exists. While you hope to never need revocation lists, if you need them they will be vital.

I recommend choosing a certificate authority that's been in business for a long time and considers X.509 certificates a major component of its business. These companies are less likely to evaporate before your certificate expires.

A good CA will also support Certification Authority Authorization (CAA) records, discussed in Chapter 8.

Finally, your certificate authority must address its own security issues. Any organization that declares that it has never had an issue is either hiding their incidents, or unaware of them. Both are bad. You want an organization that admits its incidents, swiftly and thoroughly addresses them, and changes its processes to prevent similar incidents.

When you have a short list of candidates, talk to your fellow sysadmins. If some of the CAs have been difficult to work with, cross them off your list. The CAs that work when everything is okay are fine, but the really interesting ones are those that helped your peers when things went wrong. Effective customer support that can resolve real-world problems is the most important factor in choosing any commercial vendor.[14]

Once you have chosen a CA, but before you have told your coworkers that you have made a choice, carefully read the CA's procedures carefully. CAs are inflexible on procedures, and some CAs have special CSR requirements. If the procedures and requirements are unacceptable, pick another CA.

You now know enough to understand how TLS goes wrong. Which it does. Regularly. Next up, we'll cope with that.

14 Sysadmin Rule #60 states, "Blame the vendor. Not because it is easy, but because RFC 873 declares it proper."

## Chapter 4: Revocation and Invalidation

The whole point of certificates is to indicate trust. "I have a certificate, therefore I am trustworthy." The thing about trust is that it can be lost.

The certificate's integrity rests upon the private key remaining confidential. Suppose an intruder breaks into your servers and copies your private key files. The intruder can now stand up servers that authoritatively declare they are the penetrated server, as well as anything else listed in the certificate's Common Name or SANs. If the intruder steals the CEO's user certificate, the intruder can send email as the boss. Applications and users alike will believe the intruder. This certificate is no longer trustworthy.

Further suppose that you notice the intrusion. You must communicate to clients that the certificate is no longer trustworthy.

The application can't tell clients to not trust the certificate. An intruder setting up a fake version of your server certainly won't pass that information along to clients! That information must come from further up the validation chain. Informing the CA that a certificate is compromised is called certificate *revocation*.

Revocation is troublesome. Even when everyone involved acts in good faith, it doesn't always work. Some applications deliberately ignore revocation information, or implement their own revocation system that ignores global standards. Understanding exactly *how* our applications misbehave is a huge part of a career in computing, however, and that requires first understanding how they *should* behave.

### *Revoking Certificates*

Revoking a certificate instructs the CA to inform clients that the certificate is no longer trustworthy. The exact mechanism of revocation depends on how you got your certificate. If you have a handful of certificates that you acquired through the CA's web site, you probably have a web interface that allows you to revoke that same certificate. Automated systems like ACME, as well as proprietary CA APIs, have revocation tools and interfaces.

You generally request a replacement certificate when revoking the old one. The replacement certificate must have a brand-new private key. Before generating the replacement CSR, be sure the intruder is locked out. Figure out how they broke in and plug the hole. This probably means reinstalling the system. Otherwise, the intruder will kidnap your new private key and gleefully carry on. Note that the replacement certificate costs the same amount as the original certificate, unless the CA offered revocation insurance against this possibility.

The biggest problem I encounter with revocation is that sysadmins *loathe* admitting their gear got hacked. They take it as a personal failing, rather than a routine possibility. The most careful driver in the world occasionally gets in a wreck. The greatest swordsman in the world occasionally gets beaten. Sometimes you do everything correctly and still lose. If they're using free certificates, they might just blow away the existing certificate and start over, not caring that the old certificate is out there ready to stab them.

Even worse, some organizations don't gather the data necessary to prove an intrusion occurred, or lack the expertise to assess that data. The sysadmin might not be certain that a breach happened. Every sysadmin occasionally gets that weird itchy feeling that *something* is happening, and maybe it was an intrusion, but we're not certain? It's difficult to go to your manager and say "Hey, there's a couple weird

things that might be a hack." It's even harder to add "…so you need to pay for all new certificates."

If your organization needs expensive, highly-validated EV or OV certificates, an intrusion might also trigger reports to regulatory agencies. No sysadmin wants that responsibility, especially when a government will receive a report that includes "because Inigo had a funny feeling that a Hacker in Black was lurking around the web server."

If you can inexpensively replace revoked certificates, test the revocation process by deliberately revoking your own certificate. Can your CA rapidly process certificate revocation and replacement? Maybe they can. Maybe they can't. You must know either way. Similarly, you need to test your own ability to deploy updated certificates and private keys.

Certificate revocation itself is simple. The trick comes in how the CA communicates revocation information with the client. Several protocols have been developed to cope with this issue, including Certificate Revocation Lists, Online Certificate Status Protocol, and OCSP Stapling. Today's clients might use any or all of these.

### Certificate Revocation Lists

The traditional way for a CA to offer a list of unexpired but revoked certificates is the *Certificate Revocation List* or *CRL*. The root certificate of such CAs contain a "CRL Endpoint" with a link to download the latest CRL. The first time a client contacts a TLS-protected network service, it downloads the current CRL and checks the certificate's serial number against it. If the certificate is on the list, the client rejects it.

CRLs were created in the 1990s when certificates were expensive, CAs were few, and very few web sites needed encrypted connections. It was an age when a reporter writing about the Internet bought `mcdonalds.com`, because he could. Like so many early Internet protocols, CRLs did not scale.

Consider the certificate invalidation process. Hard numbers about how many certificates are revoked are difficult to find and unreliable, but let's make some working assumptions. When you suspect you might have an intrusion, you should revoke your certificate. Let's say one percent of all certificates get revoked—it should be higher, because if you don't suspect an intrusion at least once every few years you aren't paying attention, but most of us *aren't* paying attention so let's go with one percent.

Over one billion X.509 certificates get issued each year. Every CA's CRL would combine to ten million entries. Some CAs have more, some fewer. A 2015 study showed that CRL sizes from different CAs varied from 51 KB to 76 MB. If you're sitting behind a gigabit connection that 76 MB might not seem like much, but if you're dialing in from a less advanced country it's a nightmare. On the CA's end, distributing that CRL to millions of clients demands substantial network capacity.

Fortunately, the client doesn't need to download the entire CRL on every click. CRLs can define a caching time for their CRL lookups. Most CRLs can be cached for twenty-four hours, but not all CAs follow that practice. Doubling the cache time halves the bandwidth and equipment necessary to support CRL downloads.

The CRL is also updated irregularly. Many CAs update their CRL every three hours. Others… do not.

Put together, even if a CA updates their CRL promptly and sets a standard caching time, your clients might be vulnerable for up to twenty-seven hours after you revoke the certificate.

OpenSSL can decode CRLs. Grab the root certificate your CA uses to sign your certificates and find the CRL Endpoint. Download your CA's CRL from that URL and crack it open with `openssl crl`.

```
$ openssl crl -text -inform DER -noout -in myCA.crl
```

Most CAs distribute their CRLs in DER format to reduce bandwidth needs, but you might find PEM-encoded ones.

As TLS grew more common and CRLs bloated, we tried to improve the protocol. Some CAs have abandoned CRLs in favor of OCSP.

### Online Certificate Status Protocol

The *Online Certificate Status Protocol*, or *OCSP*, sought to unobtrusively improve the performance of revocation checks by allowing clients to query the status of single certificates against the CA over an HTTP interface, rather than downloading the entire list. The CA's OCSP responder answers with either *good*, *revoked*, or *unknown*, as well as how long the client can cache this answer. OCSP responders use HTTP. OCSP responses are signed by the CA or an intermediary, so they don't need to be wrapped in TLS.

If you're trying to diagnose validation issues and suspect network connectivity, you can extract the OCSP responder URI from the full certificate chain with `openssl x509` and the `-ocsp_uri` argument. You must examine the full certificate chain, not just the end certificate. Here I check a local file generated by dehydrated (Chapter 7).

```
$ openssl x509 -noout -ocsp_uri -in fullchain.pem
http://r3.o.lencr.org
```

You can query remote servers with `s_client -showcerts`, and feed that into `x509`.

```
$ openssl s_client -showcerts -connect mwl.io:443 \
  | openssl x509 -noout -ocsp_uri
```

You can use OpenSSL's `ocsp` subcommand to test a certificate's validity. You might do this during troubleshooting, when you want to remove any outside applications from the validation process. Give the chain file with the `-issuer` option and the certificate with `-cert`. The `-text` flag says to provide human-readable output. Finally, give the URI of the OCSP server with `-url`. Here I test the validity of one of my certificates.

```
$ openssl ocsp -issuer chain.pem -cert cert.pem \
   -text -url http://r3.o.lencr.org
OCSP Request Data:
   Version: 1 (0x0)
…
```

The OCSP request includes all sorts of validation hashes and serial numbers and nonces to prevent spoofing. The part we care about is further down.

```
OCSP Response Data:
   OCSP Response Status: successful (0x0)
   Response Type: Basic OCSP Response
…
```

The query was successful, hurrah! After more hashing, certificate, and protocol details, we finally get the answer we're interested in.

```
Cert Status: good
This Update: Jan 22 18:00:00 2021 GMT
Next Update: Jan 29 18:00:00 2021 GMT
…
```

This certificate is still good. We then get the time the response was given, and the response's expiration date.

OCSP uses substantially less bandwidth than downloading the entire CRL. The CA must invest more in processing power to respond to queries, but given today's computers that's a fair trade. OCSP leaks client information to the CA, however. Where the CA once knew that a client was accessing a certificate signed by the CA, the CA now knows that the client is accessing this *particular* web site. Additionally, the client must query the CA more frequently. The CA is still a validation bottleneck, and bottlenecks exist to be removed.

### OCSP Stapling

When an OCSP client checks a certificate's validity, the CA returns a bunch of information that the client can use to validate this certificate not just now but in the future. The "next update" information we received in our manual OCSP query above is the expiration date of the

current query. A server can make an OCSP query, capture the response, and digitally *staple* it to TLS sessions. Incoming clients receive a digitally signed statement from the CA that a certificate has not yet been revoked as part of the TLS negotiations. The staple is only good for a week or so. Every few days, the server must reach out to the CA and renew its staple. This offloads the work of checking for revocation from the client to the server. Rather than one revocation check for each client whenever they contact the server, the server makes one revocation check per week.

Clients that can validate both the certificate and the staple do not contact the CA for verification. Not all clients, and not all applications, support stapling. All major web browsers do.

Stapling is more efficient than raw OCSP or CRLs, but must be configured on the application server. Many applications have built-in support for fetching and renewing OCSP stapling. If you enable stapling in Apache, it contacts the CA and renews the staple without any outside intervention. Other applications require you download the staple to a file and poke the application to read the file.

Many applications, including OpenSSL, can download the OCSP staple to a file. Most ACME clients can download an OCSP staple file, even if they don't manage the related certificate. The OpenSSL command is horribly tangled, and I recommend using anything else.[15]

If you ask, a CA can flag a certificate as "must include staple, or it's invalid." You must ask the CA to set this *Must-Staple* flag when you request the certificate. This effectively gives the certificate an expiration date the same as the staple. If the staple is not renewed, the client will not accept the certificate. I consider Must-Staple an advantage, but people who have experienced stapling failures on their servers vehemently disagree. Consider your failure modes before enabling it.

15 Since the invention of OpenSSL, there have been five cryptographic commands that were rated the most complex, the most brain-melting. This one leaves them all behind.

### *Revocation Failures*

Revocation checks don't always work, for reasons varying from application support, CA support, and developer choices.

Which applications support which revocation methods? That depends on the underlying TLS library and the application's technical debt. I've worked with some mission-critical applications that haven't been fundamentally updated since CRLs were standard.

Not all CAs support all revocation methods. Some CAs provide only empty CRLs, expecting that their clients have no technical debt and rely on OCSP. A few CAs offer OCSP but not staples. The combination of obsolete applications and irregular CA offerings leave gaps in revocation validation.

The more interesting revocation check failures come from choices made by application developers. Suppose the application cannot reach the CA to perform an OCSP check, but the certificate validates otherwise? Should your web browser show errors when the biggest OCSP responders are unreachable? Strictly speaking, yes. In reality, users will complain. If people are paying you for your application, answers like "dear customer, your Internet connection is rubbish" will not go over well. Many applications declare this a "soft failure," shrug, and permit the connection.

As a sysadmin, you must test how the applications react to being unable to check for revocation. Most of the time, a hosts file entry pointing the OCSP responder's IP address to a bogus address like 127.0.0.2 suffices. Truly confidential systems might need to fail hard when they cannot verify certificate revocation status. Check the documentation. When you discover the behavior cannot be changed, complain to the developer.

### *Browsers Versus Revocation*

TLS exists in many applications, and we've discussed the standard revocation methods they use. The most *visible* TLS applications, however, are web browsers. Developers of Firefox, Chrome, and Safari have fallen back to CRLs that they distribute as part of browser updates. They do not use a common method, either. Chrome uses CRLSets, Firefox uses OneCRL, and Safari doesn't give their method a name.

Safari and Chrome do not check OCSP or OCSP stapling by default. You can enable these checks.

You should be aware of Chrome's revoked certificate handling. Where Firefox and Safari try to provide complete revocation lists, Chrome provides a curated list. Chrome's CRLSets contain only sites the Chrome team considers "important." The curation process is not transparent, but people have written tools to extract the list. When I revoke my web site's certificate, I expect that Chrome users will never become aware of it.[16] Chrome also does not support OCSP Must-Staple, and at this time it's clear they have no intention of doing so despite the Chromium FAQ saying such support is a better solution than pure OCSP.

Nothing prompts an application to change its behavior as effectively as me writing a book about it. Many organizations provide deliberately revoked web pages that you can use to test your organization's browsers. Call up a site like `https://revoked-rsa-dv.ssl.com/` in Firefox, Chrome, and Safari and compare the results.

16 The way CRLSets encourage further centralization of the Internet is a detail that I'm certain completely escaped Google.

### *Validation Solutions*

If everything browsers deploy is awful, what are the solutions? Two incomplete real-world solutions exist.

If you work at a huge organization, short-lived certificates reduce risk. Many financial institutions, even ones as small as my credit union, use certificates that expire in one day. Deploying short-lived certificates demands automation. Some cloud companies can provide these certificates as part of their global load balancing service. This solution is expensive. OCSP Must-Staple emulates short-lived certificates, but Google Chrome has explicitly chosen to not support Must-Staple and passes that refusal down to its derivatives.

The other method is *DNS-based Authentication of Named Entities*, or *DANE*. DANE provides X.509 certificate fingerprints in DNS. Ensuring those fingerprints are valid requires DNSSEC, however, and many organizations are unwilling to protect their DNS records. DANE validation is mostly implemented in email servers, but most browsers support DANE validation plugins. My book *DNSSEC Mastery* discusses DANE.

Now let's have fun storming the server, and inspect some TLS connections.

# Chapter 5: TLS Negotiation

TLS is a complicated protocol, designed to evolve and to cope with hosts independently deploying advancements. If all hosts on the Internet supported the exact same encryption algorithms and deployed patches in lockstep TLS could be much simpler. In the real world, TLS servers and clients must negotiate with one another at every connection. Which algorithms can each side use? Which TLS versions? Each side can request or demand specific protocol options. Server and client offer their best algorithms, present their requests and demands, and try to reach agreement.

You need the ability to look at a connection and see what happens. What TLS features are enabled and disabled? If a connection fails because of a specific protocol detail, can you enable that feature on one side or the other? Or is that failure *desirable*, because one side or the other accepts only primordial SSL? We'll examine TLS connections through `s_client` connections. Once you're familiar with how the process works, you can use any tool you prefer.

Whether you're using TLS 1.2 or 1.3, you'll see three main parts of the connection: certificate validation, protocol settings, and resumption. We'll examine connections to my web server using TLS 1.2, and to Gmail's servers using TLS 1.3, using commands like those in Chapter 2.

```
$ openssl s_client -crlf -tls1_2 -connect www.mwl.io:443
$ openssl s_client -connect imap.gmail.com:995
```

Let's compare and contrast the two versions.

### *Certificate Validation*

Every TLS negotiation begins by validating all the certificates involved. Your certificate might be anchored to a trusted root certificate by a broad Tree of Trust, but the TLS client only needs to find one valid path between the trust anchor and the host certificate. The OpenSSL suite stops at the first valid path it discovers, and presents that path.

No matter if you're using TLS 1.2 or 1.3, the validation process looks identical, so we'll use my web server as an example.

```
depth=2 O = Digital Signature Trust Co.,
  CN = DST Root CA X3
verify return:1
depth=1 C = US, O = Let's Encrypt,
  CN = Let's Encrypt Authority X3
verify return:1
depth=0 CN = mwl.io
verify return:1
```

First, `s_client` validates all of the certificates in the chain. "Depth" here refers to how many links up the certificate chain each certificate is. At depth=2 we have the root CA. O is X.500 shorthand for Organization: "Digital Signature Trust Co" in this case. CN, the Common Name, is "DST Root CA X3." If this certificate is not a trusted root certificate, the validation fails right here. Note that each of these certificates has a different style of Distinguished Name. That's fine.

The `verify return:1` statement means that OpenSSL performed this operation normally. It refers to the OpenSSL code, not the certificate.

At depth=1, we have the organization "Let's Encrypt," with the Common Name "Let's Encrypt Authority X3."

The site's certificate is always at Depth 0. This certificate contains only one piece of information, the host's hostname. The hostname comes from the Server Alternative Name part of the certificate, which includes the host **`mwl.io`** among others. TLS validates the hostname against any name constraints.

```
---
Certificate chain
   0 s:CN = mwl.io
     i:C = US, O = Let's Encrypt, CN = Let's Encrypt Authority X3
   1 s:C = US, O = Let's Encrypt, CN = Let's Encrypt Authority X3
     i:O = Digital Signature Trust Co., CN = DST Root CA X3
---
Server certificate
-----BEGIN CERTIFICATE-----
MIIFXDCCBESgAwIBAgISA6Gn7sL7taoAbCDwb8hL8aKGMA0GCSqGSIb3DQ
…
-----END CERTIFICATE-----
subject=CN = mwl.io

issuer=C = US, O = Let's Encrypt, CN = Let's Encrypt Authority X3

---
```

We see a Chain of Trust again, this time ordered from the host up to the CA. The *s* in front of the certificate indicates a X.509 certificate subject, while the *i* is the issuer—the entity that signed the certificate. Certificate 0 has a subject with the CN of **`mwl.io`**, and this certificate was issued by Let's Encrypt. Certificate 1 has Let's Encrypt's Distinguished Name, and the certificate was issued by Digital Signature Trust Co.

We then have the actual server certificate. After the certificate we get a statement of the certificate's CN, and who issued the certificate. Yes, `s_client` repeats the same information in different places.

Next we have basic information about the session as it stands so far.

```
---
No client certificate CA names sent
Peer signing digest: SHA256
Peer signature type: RSA-PSS
Server Temp Key: X25519, 253 bits
---
```

The "no client certificate CA names sent" message tells us that the client has not sent a client certificate—or, if it did, the certificate is not recognized. The server is using SHA256 and RSA with Probabilistic Signature Scheme (RSA-PSS).

The *Server Temp Key* is used for Perfect Forward Secrecy (Chapter 1). X25519 is a curve type and algorithm for Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) key agreement. When the client sees this, it will generate its own X25519 keypair and send its public key back. They use these ECDHE keys to agree upon a symmetric key that will be used to protect data in transit.

```
---
SSL handshake has read 3230 bytes and written 314 bytes
Verification: OK
---
```

We have verified all certificates, and can proceed to the protocol.

### Protocol Settings

Next we have the mutually agreed-upon settings for this TLS session.

```
---
New, TLSv1.2, Cipher is ECDHE-RSA-AES256-GCM-SHA384
Server public key is 2048 bit
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
```

We're using TLS 1.2, with the cipher ECDHE-RSA-AES256-GCM-SHA384. This cipher uses 256-bit ECDH for key agreement, 256-bit AES with Galois/Counter Mode (GCM) for encryption, and SHA384 for MACs. This is a standard TLS 1.2 cipher, presented with OpenSSL's naming system, and every TLS 1.2 host should support it.

TLS 1.2 supports Secure Renegotiation. We discuss Secure Renegotiation and why it's bad in Chapter 1.

Just about every protocol and application supports *compression*. SSL experimented with compression, and early versions of TLS inherited that support. Compressing data in TLS risks confidentiality and integrity, and was removed in TLS 1.3. You can manually enable compression in `s_client` with the `-comp` argument.

*Expansion* refers to AEAD cipher suites. AEAD keys can be expanded beyond normal limits, up to a limit set by TLS.

Application Layer Protocol Negotiation (ALPN) is a TLS add-on that allows applications to integrate TLS setup into the rest of their protocol setup. It's most broadly used in HTTP/2. Enable ALPN in `s_client` with the `-alpn` flag. See s_client(1) for a description of how to use `-alpn`.

If you're using TLS 1.3, you'll get a couple more fields here.

```
Early data was not sent
Verify return code: 0 (ok)
```

*Early Data* lets you bundle application requests into resumed TLS connections. You'll also see this described as *0-RTT data*. You can send early data with `s_client` by adding the `-early_data` option with an argument of a file containing the data to be sent early.

The *Verify return code* refers us back to the certificate verification. A return code of 0 means that OpenSSL could verify it. As discussed in Chapter 3, OpenSSL failing to validate a certificate doesn't necessarily mean that the certificate is invalid. OpenSSL is continually updated to keep up with changes in the trust ecosystem. Every few years, someone finds a bug where OpenSSL cannot validate a legitimate certificate and everyone needs to update their software.

## *Session and Resumption*

TLS session and resumption details vary just enough between TLS 1.2 and 1.3 that we must consider each separately.

### TLS 1.2 Session and Resumption

Here's what you'll see with a TLS 1.2 session.

```
SSL-Session:
  Protocol  : TLSv1.2
  Cipher    : ECDHE-RSA-AES256-GCM-SHA384
  Session-ID: E22E5B9717BED0047C0E9C896D6107CB-
  1C8EA11C7E29B43621B3B3762CEA88B9
  Session-ID-ctx:
  Master-Key: 8320F39FF558B5567874761A1769F85D6B2AE-
  96C1BF604C818A14AFE590FCA80ABE…
  PSK identity: None
  PSK identity hint: None
  SRP username: None
  …
```

The session information starts with the TLS version and the cipher.

The *Session-ID* identifies a particular TLS session. Each session exists within a particular session context, or Session-ID-ctx. By setting the context, a server can make declarations like "This session is usable for the web server, but not the database." Context is most often implemented for network load balancing.

The *Master-Key* is the result of the key agreement.

The *PSK identity* and *PSK identity hint* are used for TLS via pre-shared keys, or TLS-PSK. This TLS variant is a standard, but is almost never used. This is not the same pre-shared key used in TLS 1.3 resumption.

The Secure Remote Password protocol integrates usernames and passwords into TLS. Any username associated with this session gets stuffed into the *SRP username* field.

Next we see session tickets. A TLS 1.2 session ticket contains the information needed for session resumption.

```
TLS session ticket lifetime hint: 86400 (seconds)
TLS session ticket:
0000 - 6d 83 a5 67 31 b5 bb 0a-e0 c3 72 6c b8 62...
…
Start Time: 1603377596
Timeout   : 7200 (sec)
Verify return code: 0 (ok)
Extended master secret: yes
---
```

The *TLS session ticket lifetime hint* tells the client how long it should cache this ticket.

At *TLS session ticket* we have several lines of data, comprising the actual ticket.

The *Start Time* is the epochal time when the TLS connection was negotiated. The way to convert epochal time to a date and time varies not only between operating systems, but between releases, so you'll need to look that up for your system. It's present for the *Timeout*, which shows how long the session ticket would be good for.

The *Extended master secret* declares if the ticket included an additional integrity MAC, or not.

At this point, you have a functional TLS connection and your application takes over.

### TLS 1.3 Session and Resumption

TLS 1.3 session and resumption information looks an awful lot like that of TLS 1.2. It even includes fields for deprecated components, like parts of the session ticket resumption data. This surplus is a response to network security device vendors.

Many enterprises deployed interception hardware, proxies, or other tools that performed something they called "deep inspection," validating that all TLS traffic was legitimately TLS and not some other traffic. Predictably, some of this inspection was slipshod and poorly written. Some vendors didn't keep up with the emerging standards. Even if a vendor offered TLS 1.3 support the moment the protocol was released, not all organizations apply updates in a timely manner.

An organization that used such a device could not access applications that used TLS 1.3. Rather than making sites that upgraded to TLS 1.3 become unavailable behind poorly engineered middleware devices, the standards bodies allowed TLS 1.3 to wear a mask and disguise itself as TLS 1.2.[17] Certain TLS 1.3 features are defined as TLS

17 This perfectly illustrates how and why such middleware boxes are pointless.

1.2 extensions. If you inspect a TLS 1.3 conversation you'll see obsolete TLS 1.2 fields that help maintain the masquerade.

Here we have the start of TLS 1.3's session and resumption information.

```
Post-Handshake New Session Ticket arrived:
SSL-Session:
   Protocol  : TLSv1.3
   Cipher    : TLS_AES_256_GCM_SHA384
   Session-ID: 314E771488008EBF29001D48AC6173D7A4D2695A
   …
   Session-ID-ctx:
   Resumption PSK: 0AB8007A67DB78EA27688BBBEE438C7
   …
   PSK identity: None
   PSK identity hint: None
   SRP username: None
   TLS session ticket lifetime hint: 172800 (seconds)
   TLS session ticket:
   0000 - 01 fc d2 64 5f 16 46 43-83 18 8b 7b 71
   …
   Start Time: 1598635979
   Timeout   : 7200 (sec)
   Verify return code: 0 (ok)
   Extended master secret: no
   Max Early Data: 0
  ---
```

The ticket starts with the TLS version and cipher.

The session ID and session ID context are provided for passing as TLS 1.2. TLS 1.3 does not use them.

The *Resumption PSK* is not the private key to the previous TLS session, but rather a key that proves it had a session. The server uses this, and other information in the session ticket, to compute the previous pre-shared key.

The PSK identity, PSK identity hint, and SRP username are identical to TLS 1.2, but still used only rarely.

The *TLS session ticket lifetime hint* says how long in seconds this information is good for. It's set by the server. The client could choose to discard it earlier, but rarely does.

The *TLS session ticket* is the actual ticket used for TLS 1.3. A TLS 1.3 session ticket is good for only one use, which eliminates most of the problems with session tickets in TLS 1.2.

The *Start Time* is the epochal time when this information was issued. The way to convert epochal time varies not only between operating system, but between OS releases, so you'll need to look this up yourself.

All of this data validates, as per the *Verify Return Code* of zero.

The *Extended Master Secret* used by TLS 1.2 is obsolete, but is provided for masquerading. It is always no in TLS 1.3.

A small request, like an HTTP GET, might fit in a single packet along with a TLS resumption request. The *Max Early Data* field shows how much early data can be included in a resumed TLS session. A zero indicates early data is disabled.

### TLS Failure Examples

The main reasons a modern TLS connection fails is either because the TLS client won't accept a certificate, or the client and server cannot find a mutually agreeable set of TLS options, algorithms, and protocols. Output from `s_client` and browser error messages can provide guidance on where these negotiations failed. If you've made the mistake of hand-winnowing the cipher suites your server offers, start there.

Often those errors result from misconfigured servers. A client is absolutely correct to reject expired, self-signed, revoked, and otherwise invalid certificates. As a user you might override those warnings, but that's on you. Some errors are more insidious. An Internet search recently dropped me on a "security expert's" web site that supported a maximum of TLS 1.1. Any security blog still offering nothing better than TLS 1.1 serves best as a bad example.

If you want bad examples, the best place to look is BadSSL. The site **`https://badssl.com`** links to a whole bunch of subdomains that have known-good and deliberately broken TLS configurations. Many of the misconfigured sites are certificates that will not validate. Others have validating certificates, but only speak obsolete TLS versions. Some of the good sites push acceptable standards as far as they go. If you want to see what particular failures look like, run `s_client` on some of those domains. Be sure to include `-verify_return_error` so that `s_client` will process and capture TLS-level errors.

One common error sites make is not offering a certificate chain file. Here, I poke at the BadSSL site **`incomplete-chain.badssl.com`**.

```
$ openssl s_client -verify_return_error -crlf \
   incomplete-chain.badssl.com:443
CONNECTED(00000003)
depth=0 C = US, ST = California, L = Walnut Creek,
   O = Lucas Garron Torres, CN = *.badssl.com
verify error:num=20:unable to get local issuer
   certificate
…
```

We immediately see the error: "unable to get local issuer certificate." It's right up front. Dropping the `-verify_return-error` and allowing the TLS to continue despite the invalid certificate gets us additional details.

```
$ openssl s_client -crlf incomplete-chain.badssl.com:443
CONNECTED(00000003)
depth=0 C = US, ST = California, L = Walnut Creek,
   O = Lucas Garron Torres, CN = *.badssl.com
verify error:num=20:unable to get local issuer
   certificate
verify return:1
depth=0 C = US, ST = California, L = Walnut Creek,
   O = Lucas Garron Torres, CN = *.badssl.com
verify error:num=21:unable to verify the first
   certificate
verify return:1
---
```

```
Certificate chain
   0 s:C = US, ST = California, L = Walnut Creek, O =
   Lucas Garron Torres, CN = *.badssl.com
     i:C = US, O = DigiCert Inc, CN = DigiCert SHA2 Se-
     cure Server CA
---
Server certificate
-----BEGIN CERTIFICATE-----
MIIGqDCCBZCgAwIBAgIQCvBs2jemC2QTQvCh6x1Z/TANBgkqhkiG9w
...
```

Even if you don't know what the error "unable to get local issuer certificate means," you can see that the certificate chain has only one certificate in it. Not even a trust anchor can work this way: those certificates lack SANs and CNs that would match the server. It's pointing right at the problem.

Admittedly, TLS is complicated. X.509 extensions are numerous. A skill at copying error messages into search engines is invaluable, once you have the understanding to place the answers in context. I copied "unable to get local issuer certificate" into four major search engines, and in all of them the correct solution was the first hit. Not all errors are that accessible, admittedly, but they're a place to start.

Next, we'll get our own certificates.

# Chapter 6: Certificate Signing Requests and Commercial CAs

Getting a certificate is theoretically easy. A sysadmin or an automated process generates a *Certificate Signing Request* or *CSR*. The CSR contains all the information that the CA verifies, and perhaps more. You can think of a CSR as an unsigned certificate, although that's not quite correct. No matter how you get your certificates, you create CSRs. When something goes wrong, you need the ability to scrutinize them. RFC 2986 documents CSRs.

One year, as currently offered by commercial CAs, is a perfect length of time to forget everything you've ever known about generating CSRs. If an intruder steals your private key, you must immediately generate a new CSR and private key. Document the CSR creation process and any configuration files you need so you can easily repeat it on demand. Write a script or, better still, entirely automate the request process.

If you're using ACME, you'll configure your certificate signing requests once and then let the automation handle them. This means you'll ignore them until something breaks catastrophically, at which point you'll have to re-learn CSRs all over again.

Most sysadmins deal with CSRs primarily when purchasing commercial certificates. We'll look at CSRs from that perspective, but everything applies to ACME certificate signing requests as well.

### *Reusing CSRs*

While you *can* reuse CSRs to renew a certificate, it's terrible practice. Each CSR is tied to a public key pair. While you can't prove a negative, "prove that your private key hasn't been stolen" is especially difficult. Intruders vastly prefer people reuse private keys. It lets them use your brand-new certificate with the key they stole last year to masquerade as your systems. Even if your key is never stolen, after a few reuses that key will be weaker than recommended.

Any time you renew a certificate, create a new private key and a new CSR using that key. Generating a new private key is fast, inexpensive, and easy. It's even part of the CSR generation command.

Many tutorials recommend reusing private keys, but that's like driving over the speed limit while not wearing a seat belt so you can more easily grab a fresh can of beer from the back seat. You'll get away with it, perhaps for quite a while, but it will eventually hurt you.

### *Why Go Commercial?*

For decades, the only way to get an X.509 certificate was to purchase one from a commercial certificate authority. Today, DV certificates can be free. If you can get free certificates via ACME, why use a commercial CA?

Some organizations require high validation levels (Chapter 3), but most free and inexpensive CAs offer only domain validated (DV) certificates. Regulatory, legal, or military standards might require your organization to use OV or even EV certificates. Many organizations that specialize in money institutionally distrust free services. If your organization's leaders declare "You must use a commercial certificate authority" or even "you must use this *particular* CA," the decision is made regardless of your heartfelt and technologically sound opinions on the matter.

You might also use a commercial CA if you want a special-purpose certificate, such as for running your own CA. We discuss these in Chapter 10.

When using a commercial certificate authority, be sure to allocate money for replacement certificates in case you must revoke the certificate. Some CAs offer insurance for revocations. Buy it, and use it any time you feel even faintly suspicious of an intrusion.

The CSR is the key to dealing with a commercial certificate authority. The certificate signing request contains all the information that the CA will verify. For Domain Validated (DV) requests this might be only

the domain name of the server that will use the certificate. OV and EV certificates need much more information, and it all must be correct.

We'll discuss some considerations for gathering information, the certificate's public key algorithm and how to request and receive a certificate.

### Gathering Information

A certificate that's more than domain validated requires more information. Do yourself a favor and gather all information before running any commands. Certificate authorities are fairly competent at rejecting certificate signing requests that don't match official documents.

The *country name*, shown as *C* in X.509, uses a two-letter code for your country, as defined in ISO 3166. If you're unsure of your country's ISO code, check before filing the certificate.

The *ST* X.509 code represents your state or province. Do not abbreviate the name of your state. Spell it out.

*Locality*, or *L*, is a fancy way to say "city." What city is your organization truly in? Perhaps everyone says that your office is in one city, but the formal business address is in the next town over. I've worked in offices that claimed to be in one city, but were just barely over the border of a less prestigious city. The owners claimed they were in the more upper-class city, because they could see it from the windows of the executive suite. Certificate authorities don't care about such political games; use the official city name.

The *Organization*, or *O*, is where most people screw up. What is your organization's *legal* name? Maybe you call your workplace The Pit, but any CA validating your organization needs the complete formal name, "The Pit of Despair, LLC."

Gather and document these facts. Write them down. Consult the documentation whenever you create a new Certificate Signing Request.

### Public Key Algorithm

TLS certificates can use two public key authentication algorithms, RSA and ECDSA. The person creating the certificate request chooses the algorithm.

The Rivest-Shamir-Adleman (RSA) algorithm has been the premier public key encryption method since 1977. It's the standard technique underlying almost every encryption method. While increasing computing capacity has required RSA keys to get longer, the algorithm itself has withstood many years of focused investigation from cryptographers.

In 2005 the National Institute for Science and Technology (NIST) released the Elliptic Curve Digital Signature Algorithm, or *ECDSA*. This newer algorithm promises the same level of confidentiality as RSA but with less number-crunching, which makes it attractive for devices with less computing power and environments where performance is paramount. ECDSA was released by a US government agency, however, which means some people automatically distrust it. Cryptographers have pummeled ECDSA for less than two decades, and while it's endured so far, it lacks RSA's long history.

Which should you use? Everybody supports RSA. If you're targeting mobile platforms, or want to reduce computation overhead, consider ECDSA.

Common wisdom declares that if you want an ECDSA certificate, you need a CA that has an ECDSA root certificate. This isn't true. A CA that only has an RSA certificate can sign your ECDSA certificate. If you've chosen to use an ECDSA certificate because you want your application to work on smaller systems that struggle with RSA, having an ECDSA certificate signed by an RSA CA certificate forces the client to perform both RSA and ECDSA calculations. The client does less work than a pure RSA certificate chain, but more work than a purely

ECDSA chain. You're better off choosing a CA that can provide an RSA-free certificate.

Many applications can support both RSA and ECDSA certificates by letting you set one as preferred and the other as the backup. Buying two certificates and installing both is a viable option.

### Common Names

If you've previously dealt with X.509 certificates or read any documentation on getting them, you've probably encountered tutorials telling you to run a command and walk through a set of spiffy prompts to set certificate information. Unfortunately, that's utterly obsolete—except when it's not.

Way back at the murky dawn of SSL, certificates officially stored the hostname under Common Name (CN). In 2000 the Subject Alternative Name extension for storing hostnames and other subjects was approved, and storing this information in the Common Name became a mere fallback for obsolete software. This fallback was removed from the standard in 2011. It's been obsolete for four times as long as it was used, so it should be a historical footnote like UUCP over Filling Jon's Van With Backup Tapes And Driving To The Campus Across Town. Right?

It just so happens that storing hostnames in CN is only *mostly* dead.

Thanks to decades of obsolete and incorrect tutorials, some of them written last week, many sysadmins and programmers don't know that putting a hostname in CN is deprecated. Even some CAs still expect to find a CN in the certificate request. Other CAs accept that users follow those bad tutorials. They use the information in CN to populate the SAN field, and their web form has spaces for adding additional SANs. Other CAs expect you to populate both CN and SAN in your certificate request.

CNs cannot handle modern Internet hostnames. The maximum length of a CN is 63 characters. I own the web site **`www.YouKeepUsingThatWordIDoNotThinkItMeansWhatYouThinkItMeans.com`**. If I attempt to request a TLS certificate for my web site using my perfectly legitimate site name, it will fail.[18] I must order a certificate for a shorter site, and add the legitimate name that I'm truly interested in as a SAN.

Also, CNs are not checked for name constraints. A few clients do check name constraints against CN, but this can make the client reject valid certificates.

Lingering support for hostnames in CN in all sorts of software encumbers further improvements in TLS and other protocols. People are actively working to remove it from the TLS ecosystem. It will go away. Check your infrastructure for dependencies on hostnames in CN and remove them.

We're going to skip all the confusion and future-proof your certificate requests by skipping the prompt-based method and including both CNs and SANs in your requests. This requires understanding OpenSSL configuration files.

### *OpenSSL Configuration Files*

Your host has a default OpenSSL configuration file, telling the various OpenSSL commands how to behave and what commands to use. It's in your system's OpenSSL directory. The default OpenSSL directory is *`/usr/local/openssl`*, but almost no operating system puts it there. The simplest way to find the directory is to ask OpenSSL.

```
$ openssl version -a | grep -i openssldir
OPENSSLDIR: "/etc/pki/tls"
```

This CentOS system puts its OpenSSL configuration in *`/etc/pki/tls`*. Wherever your operating system puts it, go to this directory and look for *`openssl.cnf`*.

18 It's like OpenSSL doesn't even realize that my job is on the line.

The configuration is broken up into sections named after the command they affect, labeled in brackets. For `openssl req` configurations, look for sections with names beginning in `req`.

```
[ req ]
default_bits            = 2048
default_md              = sha256
...
```

OpenSSL configuration consists of variables, equals signs, and values for the variables. Hash marks (`#`) indicate comments. You don't need to quote variable values, but you can use quote marks to preserve leading and trailing white space. If a value must wrap around to the next line, use the traditional backslash (`\`) to do so.

The variables in a section only apply to that section. Anything under `[ req ]` affects the `openssl req` command alone. A configuration could have a different section for a different command that also had a `default_bits` variable, but TLS and cryptography are already sufficiently confusing and the OpenSSL developers feel no need to make it gratuitously worse.

To make OpenSSL more exciting, configuration settings can change the configuration file format. Study the manual before changing anything, and keep meticulous backups.

Editing this file might be fine. It might ruin everything. It depends on your Unix variant, the installed software, and the distance to the nearest pirate ship at this exact moment. Leave the system-wide defaults alone. Most functions, like certificate creation and management, allow you to create a standalone configuration file containing only the options needed for that operation. Entries in this local config file override the defaults.

We'll use configuration files to create CSRs.

### *Creating CSRs*

A certificate signing request contains information for the CA to validate, a digital signature of that information, and a code for the signature algorithm. Per best practices, we'll also generate a new keypair. You can create a CSR either with a configuration file or a lengthy command line. Both use the `openssl req` command. We'll use server certificates for most of our examples, but also demonstrate client certificates.

Creating a renewal CSR with a new private key requires running the same command used to create the first CSR. Whichever method you use, keep a record of exactly what you did. If you used a configuration file, keep that file. If you used a command line, script it. In either case, give the file a clear name that will let you easily identify it both next year and in ten years. You won't remember any of these commands next week, let alone next year, so comment it well.

Certificate requests create files. Before requesting your first certificate, consider how you're going to keep track of those files. If you need several certificates, naming all your private key files *`private.key`* will drop you headfirst into the Fire Swamp. A certificate's filenames should include the certificate's primary host. I give my keys file names like *`mwl.io-private.key`* for the web server's private key, *`mwl.io.csr`* for the certificate signing request, and *`mwl.io.crt`* for the complete certificate. If you have many certificates add the date as a suffix, in YYYY-MM-DD format. Don't rely on directories to keep files straight; use filenames. If copying a file to the wrong directory destroys your filing system, you've named your files incorrectly.

We will start with the configuration file method. Once you understand that, you'll have all the context needed for pure command line.

## *Creating ECDSA CSRs*

As we create CSRs with the `openssl req` command, the configuration file options must go in a `req` section. Here are some examples from the default *`openssl.cnf`*, and the openssl-req(1) man page lists many more options. We'll look at five components: the main `req` section, private key password management, the Distinguished Name, and Extensions, and put those together to request the certificate.

### Main req Section

The `req` section contains settings for the `openssl req` command. I create the configuration file *`mwl.io.conf`*.

```
#create an ECDSA certificate for mwl.io
#
[ req ]
prompt             = no
default_keyfile    = mwl.io-private.key
distinguished_name = req_distinguished_name
req_extensions     = v3_req
```

…

The very first line of this configuration file is a comment. It gives explicit details on what this configuration is used for. When the time comes to renew the certificate, I need only look at the comment to proceed.

The `prompt` option controls the interactive dialog shown in so many OpenSSL tutorials. By setting it to *no*, we tell OpenSSL that this configuration file is for non-interactive use.

In a generic OpenSSL configuration file like */etc/openssl.cnf*, the `default_keyfile` variable sets a location for private key files created on the command line. Here, I use `default_keyfile` to save the key in a domain-specific file. Remember that the private key is both confidential, and a vital part of the certificate. Protect it as discussed in Chapter 1.

The `distinguished_name` illustrates how *`openssl.cnf`* can pull part of a configuration out into separate sections. The `distinguished_name` settings are broken out into the section `req_distinguished_name`.

Modern certificates store SANs in X.509v3 extensions, so I add the `req_extensions` statement and refer it to `v3_req`.

#### Password Management

OpenSSL defaults to encrypting private key files with a password. (We discuss the advantages and disadvantages of this approach in Chapter 1.) Create a new private key every time you create a new CSR. You can either leave the private key file unencrypted, put the password in the configuration file, or enter the password on the command line when creating the CSR.

To not encrypt the private key, add the `encrypt_key` option to the `req` section and set it to *no*. You could also do this by adding the `-nodes` option to your command line, but part of the goal of a configuration file is to simplify the command line. The option -nodes means "no DES." DES was replaced long ago, but the command line option remains unchanged for compatibility.

```
encrypt_key = no
```

To include the password in the configuration file, add the `output_password` option to the `req` section. Set it to the private key password.

```
output_password = haxorsKnowThisPassword
```

If you set neither option, OpenSSL prompts you for a password when creating the private key.

#### req_distinguished_name

The default *`openssl.cnf`* has a whole bunch of sample entries under `req_distinguished_name`. They're designed for CN-centered

certificates. If you're creating your certificate from a configuration file, the format changes. (Technically, it's setting `prompt` to *no* that changes the configuration file format, but that's the purpose of that setting.)

Under the `req_distinguished_name` section, list all of the components of the Distinguished Name. For an EV or OV certificate, that includes the country, state, city, organization, and Common Name.

```
[ req_distinguished_name ]
C = US
ST = Michigan
L = Detroit
O = Inconceivable Incorporated
OU = IT
CN = mwl.io
```

I'm creating a DV certificate, so I need only the Common Name. Yes, storing the hostname in CN is deprecated. We're going to do it anyway.

```
[ req_distinguished_name ]

CN = mwl.io
```

The CN is present only for legacy compatibility—but *wow*, is there a bunch of legacy stuff out there.

#### Extensions

Version 3 of X.509 has all sorts of extensions. There's extensions to define how a certificate can be used, extensions to set constraints, extensions to define the CRL location and to tell you where to find CA policies. It's easy to look at this stuff and bog down in trying to figure out which should apply to your Certificate Signing Request.

Don't do any that.

Extensions can be applied at the signing stage. Your CA will add its own information. They'll put restrictions in place, like "this certificate cannot be used to sign more certificates" and "only suited for web applications." Request extensions only for things your application documentation explicitly declares that it needs, and make the CA refuse you.

The go-to use for extensions is Subject Alternative Names, or SANs. This is the standard location for identifying the hosts a certificate is valid for. Identify the names your certificate needs in the `v3_req` section. SANs go in the `subjectAltName` field. List multiple SANs in this field, separated by commas. Each SAN declaration has two parts, the type of SAN and the value, separated by colons.

```
[ v3_req ]
subjectAltName = DNS:mwl.io,DNS:www.mwl.io,DNS:cdn.mwl.io
```

This CSR has three SANs: **`mwl.io`**, **`www.mwl.io`**, and **`cdn.mwl.io`**.

Note I include the Common Name as a SAN. SAN is the standard way to get valid names. Support it.

A CSR can include many SANs. The standard defining Subject Alternative Names (RFC 5280) does not set a maximum number of SANs that a certificate can support. Certificate authorities can and do set limits, however. 100 is a common limit, but a few CAs do support thousands—for a modest fee per name, of course.

While you *can* technically put thousands of entries on a single line in a configuration file, in practical terms that's as daft as going against a Sicilian when death is on the line. If you want to have each hostname on its own line, create an *`openssl.cnf`* array and make your SANs entries in that array.

```
…
[ v3_req ]
subjectAltName = @alt_names

[alt_names]
…
```

Under the `v3_req` section, the `subjectAltName` is set to `@alt_names`. That tells OpenSSL to look for another section, called `alt_names`, which we define next. Each entry under `alt_names` has this format.

```
type.number = hostname
```

All of our SANs are of *type* DNS. Other types are discussed in x509v3_config(5), but only DNS is applicable for most of us. (The SAN IP address type looks to be of interest, but no publicly trusted CA will accept a certificate with one.) The *number* is an increasing integer that assigns the value a place in the array. You can skip numbers, but don't duplicate them; putting multiple values into one space in an array doesn't work. Last, give the *hostname* you want to appear in the SAN.

```
[alt_names]
DNS.1 = mwl.io
DNS.2 = www.mwl.io
DNS.3 = mwlucas.org
DNS.4 = tiltedwindmillpress.org
DNS.5 = michaelwarrenlucas.com
...
```

In large organizations, array support eases CSR automation. You can write a script to pull the hostnames from a database and create the configuration file. While SANs appear in the CSR in array order, the order does not affect validation and has no impact on TLS.

We discuss wildcard certificates, and their numerous disadvantages, in Chapter 3. If you persist in creating one, list the wildcard as a SAN.

```
[alt_names]
DNS.1 = mwl.io
DNS.2 = www.mwl.io
DNS.3 = *.api.mwl.io
DNS.4 = *.cdn.mwl.io
```

This creates a certificate that's good for select hostnames in **`mwl.io`**, plus wildcards for two subdomains. This certificate can be installed on my web server, as well as any host in **`api.mwl.io`** and **`cdn.mwl.io`**. I must also install the private key on each of these machines, so a compromise of one compromises the network. Wildcard certificates add risk to not just hosts, but your entire ecosystem.

Put together, the complete configuration file for a certificate with an ECDSA key looks like this.

```
[ req ]
prompt                  = no
default_keyfile         = mwl.io-private.key
distinguished_name      = req_distinguished_name
req_extensions          = v3_req

[ req_distinguished_name ]
CN = mwl.io

[ v3_req ]
subjectAltName                  = @alt_names

[alt_names]
DNS.1 = mwl.io
DNS.2 = www.mwl.io
DNS.3 = *.api.mwl.io
DNS.4 = *.cdn.mwl.io
```

Once you have a complete configuration file, make sure you have a parameters file.

#### Elliptic Curve Parameters Files

Elliptic curve cryptography is different than RSA. From the sysadmin perspective they seem the same: you protect the private key and give away the public key. You submit CSRs to the CA and get certificates back. You lose the private key passphrase and have to buy a new certificate. It's all good.

ECDSA is defined by *elliptic curves*. An elliptic curve is a mathematical construction. An ECDSA certificate uses one of a set of well-known elliptic curves. Each of these well-known curves has a name. The most widely supported curves are P-256, P-384, and P-251 as defined by NIST. You might see these curves erroneously described as lengths, but curves are not bit lengths. (If you need a complete list of elliptic curves your system supports, run `openssl ecparam -list_curves`.) If you know nothing about elliptic curves except that ECDSA certificates are better for small devices, use the curve P-256.

Before you can generate an ECDSA CSR, you need a *parameters* file for the chosen curve. Curve parameters describe a curve. The parameters file is not a key, nor is it confidential. It's kind of like the */etc/protocols* file, but mathy. Everybody knows what's in it. OpenSSL's `genpkey` command, normally used for generating private keys, also writes out parameter files.

```
$ openssl genpkey -genparam -out filename.pem \
   -algorithm ec -pkeyopt ec_paramgen_curve:curvename
```

You have only two decisions to make: the name of the created file, as given in `-out`, and which curve to use. Here I create a P-256 parameters file, *ec256-params.pem*.

```
$ openssl genpkey -genparam -algorithm ec \
   -out ec256-params.pem -pkeyopt ec_paramgen_curve:P-256
```

If you poke around you'll see that DHE uses similar parameter files. If the need ever arises, you can generate those files with `openssl genpkey` as well.

#### Requesting ECDSA Certificates

We have a configuration file and a parameters file. We can now create an ECDSA certificate signing request, using the `openssl req` command.

```
$ openssl req -newkey ec:parameters.pem \
   -config filename.conf -out filename.csr
```

The `-newkey` option creates a new key, using the parameters in the parameters file. I give the configuration file with `-config` and the destination file with `-out`.

Here I use the ECDSA configuration file and the parameters file we just created to create an ECDSA CSR for my domain **`mwl.io`**.

```
$ openssl req -newkey ec:ec256-params.pem \
   -config mwl.io.conf -out mwl.io.csr
Generating an EC private key
writing new private key to 'mwl.io-private.key'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----
```

The file *mwl.io.csr* now contains an ECDSA CSR.

Some tutorials suggest creating a new key and then using that key to create the CSR. That's valid, but leads to more typing.

## *Generating RSA CSRs*

Create an RSA CSR by writing a configuration file and running a command.

### RSA CSR Configuration File

The configuration file for an RSA CSR is almost identical to the ECDSA version, with one exception. You must set the length of your RSA key with the `default_bits` option. If you want a non-default hash algorithm, specify that with `default_md`.

In this configuration I set a default RSA key length of 2048 bits and hard-code SHA256 hashing. The rest is exactly like the ECDSA key.

```
[ req ]
prompt             = no
default_bits       = 2048
default_md         = sha256
default_keyfile    = mwl.io-private.key
distinguished_name = req_distinguished_name
req_extensions     = v3_req

[ req_distinguished_name ]
CN = mwl.io

[ v3_req ]
subjectAltName = @alt_names

[alt_names]
DNS.1 = mwl.io
DNS.2 = www.mwl.io
```

I have not specified any provisions for password management, so the CSR creation command will prompt me for a password. With this file, I can generate an RSA CSR.

### Requesting RSA Certificates

Use the `openssl req` command a configuration file to generate an RSA certificate request. Unlike ECDSA, you don't need a parameters file.

```
$ openssl req -newkey rsa -config filename.conf \
  -out filename.csr
```

The `-newkey` argument tells `req` to create a new private key. We give it one argument, `rsa`, to create an RSA key. The `-config` argument sets the configuration file, while `-out` lets you give the filename for the completed CSR.

Here I generate a certificate signing request for my domain **`mwl.io`**, using the configuration file *`mwl.io.conf`*.

```
$ openssl req -newkey rsa -config mwl.io.conf \
  -out mwl.io.csr
Generating a RSA private key
.............+++++
.....................................................
.................+++++
writing new private key to 'mwl.io-private.key'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----
```

I enter my pass phrase twice, and am rewarded with the file *`mwl.io.csr`* containing my RSA CSR.

### Client CSRs

The process and algorithms for a client certificate are identical to that of a server certificate, but the information in the certificate is slightly different. Unlike server certificates, there is no defined standard for the information that must appear in a client certificate. Different

applications might require different details. I'm describing a common usage, but double-check your documentation for any special needs.

Many applications include their own CA for signing client certificates. You don't want your VPN device to accept all client certificates signed by any public CA. Such a device should only accept certificates that it has signed. Such systems might have their own system for generating CSRs.

Here is a configuration file for a client certificate that authenticates a user to a corporate VPN. The certificate needs the user's name as the Common Name, and the certificate must include the user's email address. Back when CN contained a hostname, the Distinguished Name could include the user's email address as the emailAddress attribute. That's now obsolete, but it persists much the way hostnames in CNs do. The proper way to give an email address is with an rfc822Name value as a Subject Alternate Name, much as we do with hostnames. I'll use both in the configuration file, but if your application lets you drop emailAddress from the Distinguished Name, please do so.

The configuration resembles one for a server certificate request. I could add organization information in the Distinguished Name, but it's most often unnecessary.

```
[ req ]
prompt                  = no
default_bits            = 2048
default_keyfile         = client-private.key
distinguished_name      = req_distinguished_name

[ req_distinguished_name ]
CN = Michael W Lucas
emailAddress = mwl@mwl.io

[ v3_req ]
subjectAltName                  = email:mwl@mwl.io
```

OpenSSL doesn't care if a certificate request is for a client or a server. CSR creation is exactly the same.

```
$ openssl req -newkey rsa -config client.conf \
   -out client.csr
```

The file *client.csr* contains my Certificate Signing Request.

Client certificates used by people must have a passphrase. The purpose of a client certificate is to authenticate a user. This certificate will always be used by a human being, and are most often used on mobile devices like laptops. An unencrypted private key can be used by anyone who steals the laptop. If one of the lords of your organization whines that they can't seamlessly double-click on the VPN application without having to type their passphrase, ask them why they want to risk the company's integrity on their ability to avoid muggers.

Client certificates used by applications don't need to have a passphrase. Maybe your mail server must relay outbound email through your service provider's mail server, and must use the certificate to authenticate to that relay. Protecting the certificate's private key the same way you would protect any other private key on the server is sufficient.

#### Certificates Without Subjects

If CN is obsoleted in favor of SANs, do you need them in your certificates at all? Nope. You can create certificates without Subjects or Distinguished Names.

While the Distinguished Name is important for highly validated certificates, you can skip it entirely for DV certificate requests. Older clients might have trouble with the certificate, but they should upgrade anyway. Here's a configuration file for a client certificate without a Distinguished Name.

```
[ req ]
default_bits            = 2048
default_keyfile         = mwlucas-private.key
distinguished_name      = req_distinguished_name
req_extensions          = v3_req

[ req_distinguished_name ]

[ v3_req ]
subjectAltName                  = email:mwl@mwl.io
```

The main annoyance with these is that you can't set `prompt` to *no*. That triggers an OpenSSL safety check. Instead, you can use `-subj` on the command line to set the Distinguished Name to `/`. This is just the leading slash of the DN, with nothing behind it. The command line is otherwise identical.

```
$ openssl req -newkey rsa -config client.conf \
  -out client.csr -subj /
```

If you examine the CSR, you'll see that it has the email address in the Subject Alternative Name but there isn't even a Subject field.

Try it sometime. See how much of your software breaks. Yell at the developers.

#### CSRs Without Configuration Files

The configuration file isn't difficult, but sometimes you need to create a CSR entirely on the command line. This command isn't pretty. Creating a temporary configuration file is much less onerous and increases your ability to double-check your work before creating the CSR. But if you have an automated process to create CSRs, you'll need that process to assemble and run the command line. This only works with OpenSSL 1.1.1 and newer.

The command line looks like this.

```
$ openssl req -newkey algo:length -keyout privatekey.pem \
   -out request.csr \
   -subj /C=country/ST=state/L=city/O=organization/CN=hostname \
   -addext "subjectAltName=DNS:hostname,DNS:hostname"
```

The `-newkey` argument needs two colon-separated arguments, the algorithm and the key length or parameter file.

The `-keyout` argument sets the name of the file to write the private key to. If you don't want to encrypt the private key, add the `-nodes` flag.

The `-subj` argument lets you set the Distinguished Name. If you have a subject, put a slash before each directory entry. Use the `-subj /` trick to create a certificate with no subject and rely on SANs instead.

If you're creating an EV or OV certificate, you must set the C, ST, L, and O values. *C* represents Country, and must be the ISO 3166 two-letter code. *ST* represents State or Province, and should be the full name. *L* is for Locality, or city where the organization is legally located. *O* is for Organization, or complete, legal business name. *OU* represents the Organizational Unit, or department within the organization. You will also see these as *countryName*, *stateOrProvinceName*, *organizationName*, and *organizationalUnitName*. The labels can be used interchangeably.

The `-addext` flag lets you add X.509v3 extensions to your CSR. SANs use the SubjectAltName extension. Specify them exactly as you would on a single line in the configuration file. If you have dozens of SANs, either type very carefully or use a configuration file.

Here I create an RSA certificate for **`mwl.io`**, adding SANs for **`mwl.io`** and **`www.mwl.io`** as well as a wildcard for the subdomain **`cdn.mwl.io`**. I must escape spaces in the organization and location information.

```
$ openssl req -newkey rsa:2048 -keyout www-private.pem \
   -out www-request.csr \
   -subj /C=FL/ST=Humperdinck/L=Florin\ City/\
   O=Inconceivable\ Incorporated/CN=mwl.io \
   -addext \
   "subjectAltName=DNS:mwl.io,DNS:www.mwl.io,DNS:*.cdn.mwl.io"
```

Again, if you're using DV certificates, you can skip all of the organization information and provide only the CN.

```
$ openssl req -newkey rsa:2048 -keyout www-private.pem \
   -out www-request.csr -subj /CN=mwl.io \
   -addext \
   "subjectAltName=DNS:mwl.io,DNS:www.mwl.io,DNS:*.cdn.mwl.io"
```

I strongly recommend putting these commands in well-named, well-commented shell scripts, so you or your successors can trivially generate new CSRs every year for the rest of your organization's existence.

Before sending any CSR to any CA, double-check your work by examining the contents.

### *Viewing CSRs*

You meticulously entered all of the correct data into a fiendish OpenSSL command and generated a file that supposedly contains your CSR. The file contains a bunch of gibberish, surrounded by markers that declare BEGIN CERTIFICATE REQUEST and END CERTIFICATE REQUEST. How can you check your CSR to be sure it's correct?

Like other OpenSSL commands, `openssl req` can read files and transform them to other formats. Use `-in` to give the filename containing the CSR and `-text` to produce textual output. You can add `-noout` to skip printing the certificate signing request in the output. All of these should look familiar from our other OpenSSL commands. Here I double-check a CSR for my domain **`mwl.io`**.

```
$ openssl req -in mwl.io.csr -noout -text
Certificate Request:
   Data:
      Version: 1 (0x0)
      Subject: CN = www.mwl.io
      Subject Public Key Info:
         Public Key Algorithm: id-ecPublicKey
…
```

The *Subject* field shows the organizational data included in your CSR, such as the company name and the hostname. Below that you have the public key information, including the key algorithms, followed by SANs and other extensions.

### *Using the CSR and Certificate*

You now get the CSR to your Certificate Authority. If you're using ACME, this happens automatically—just like creating the CSR. If you're using a commercial CA, you must follow the CA's CSR-signing procedure. In exchange, you get a shiny new X.509 certificate!

Look at the certificate before installing it. The CA might add its own extensions, or remove some of those you requested. Verify that the certificate includes everything you need.

Now install the certificate in your application. This book can't help with specific applications, but here's some general advice. Name your files after the fully qualified domain name in the certificate's common name. In production, my **`mwl.io`** certificate is called *`mwl.io.crt`* and the private key is *`mwl.io.key`*. Protect the key files, preferably in a directory readable only by the application and **`root`**.

On a Unix system, you might put all of your certificates in a directory like */etc/certs*. This way, any application can access the certificate for this host. Private keys could go into */etc/certs/keys*. Make that directory owned by **`root`**, but create a group that can read the directory and the files therein. Add application users to that group.

Back up your certificate. Double check the backups. Hire a giant to protect the backups.

### Reconnecting Files and Finding Reused Keys

Reusing private keys is bad practice. But how can you be sure that some flunky didn't follow one of those obsolete Internet tutorials and reuse a private key instead of generating a new one? If your CSR filing system turns out to be inadequate and you have to sort out which certificate belongs with which private key, how do you sort them out?

These are separate problems with the same solution.

A keypair's *modulus* is one of its unique numerical characteristics. You don't need to know the math, only that it's unique. You can extract the modulus from a file with OpenSSL's `-modulus` option. The modulus is a very long number, but you can easily check it using `openssl md5`. (While MD5 is no longer cryptographically secure, it's perfectly usable as a non-cryptographic hash.) Here I extract the key modulus from a certificate, a private key, and a CSR.

```
$ openssl x509 -noout -modulus -in server.crt | openssl md5
d261c8136ff4154881814df84cc0829a
$ openssl rsa -noout -modulus -in server.key | openssl md5
d261c8136ff4154881814df84cc0829a
$ openssl req -noout -modulus -in server.csr | openssl md5
2606bbdc0f7b099117342eefd049d5b0
```

The hash of the certificate and key begin with *d*. The hash of the CSR begins with *2*. We can stop looking; this CSR was not used to create this certificate.

Every organization has unique procedures for creating keys, so I can't tell you where to implement such checks to ensure private keys are not reused. Worst case, you could put them in your network management system. A certificate modulus that hasn't changed for over a year indicates it reused the previous private key. It would be best to catch these *before* deployment, though.

Speaking of automation, let's get some certificates automatically.

# Chapter 7: Automated Certificate Management Environment

For a protocol that gets used everywhere, TLS has a whole bunch of disadvantages. Any process that an organization uses annually, like renewing certificates, suffers neglect and inevitably either gets forgotten until it's too late or, after a few failures, attracts intrusive high-level supervision. You install the new cert, and three days later a zero-day exploit roots your web server. Certificate revocation is troublesome and not wholly effective. Your organization gets bought out and all the certs need re-purchasing. You automate certificate management using your CA's API, and three months later someone from the Board of Directors is at a party where a smarmy prince who can't be trusted to clean up after a puppy mentions that your CA disrespected his ignorance and so you have to change CAs.

What TLS needs is a standard protocol for automating certificate management. A protocol implemented in a bunch of different tools, letting you choose software exactly as you choose which web server you want to run. You could point this tool at any CA and get certificates automatically renewed, revoked, or edited. This would let you turn certificate requests from an infrequent annoyance to a forgettable steady state.

The Internet Security Research Group (ISRG) created exactly this, with the Automated Certificate Management Environment (ACME). ACME is a client-server protocol for client-CA interactions, and has dozens of implementations. Unlike the various APIs developed by

separate CAs, ACME has gone through the standards process and is officially RFC 8555. Initially tested and deployed by ISRG's Certificate Authority, Let's Encrypt, ACME is now being deployed by a variety of free and commercial CAs. Other organizations are also trialing ACME for non-server certificates, such as the S/MIME certificates available at `https://acme.castle.cloud`.

## *How ACME Works*

ACME combines a standard client-server application with the addition of DNS-based verification challenges. Remember, Domain Validation is specifically about the Domain Name Service. A CA can legitimately use DNS to verify control of a site.

A CA's ACME server is available at a URL, usually a HTTPS web site. Any host that can reach the API can create an account, submit CSRs, and request a signature.

### ACME Registration

When you first use an ACME client, it creates its own key pair to uniquely identify the client. The first time the client contacts a server, it must accept the CA's terms and then register an account tied to that public key. ACME clients have specific single-use flags for these operations. All further interaction between that client and the server are signed with that key. When you're using a free CA, that process is automatic and only takes a few seconds. With an account, the client can submit certificate signing, revocation, and renewal requests.

The client's public key serves as a unique identifier, so you can use ACME with commercial CAs. The CA would validate an organization's identity and provide you with a nifty web portal where you can update your credit card number and list your ACME client keys. As ACME is now a standard protocol, many CAs are plunging full speed into their deployments.

### ACME Process

Once the client is registered, it may then submit orders to the server. An order declares, "Here are the hosts and/or domains I control and want to get certificates for."

The server must validate that the client controls those domains, so it sends a list of supported *challenge methods*. The challenge method describes ways the ACME server can reach back to the server and validate that the client really does control it. The client picks its preferred challenge method from the list. Each hostname the client wants to get a certificate for gets its own challenge.

Once the client picks a challenge for each domain, the server sends details on how to perform the challenges. The client performs the challenge and tells the server to verify its work. The server validates that the challenge is successful, or not. If the client fails the challenge, the order fails. If the client succeeds, it may finalize the order and submit CSRs to the server. The server signs the requests and returns certificates to the client. The client is responsible for boring sysadmin tasks like copying the certificate into place and restarting any servers.

## *ACME Challenges*

An ACME challenge includes a token and a key authorization.

The *token* is a location. The client is responsible for creating and placing the token.

The *key authorization* is a combination of the token and a digest of the account key. The key authorization is used as the contents of the token.

The ACME server checks the token location and, if it finds the key authorization, agrees that the client controls that server.

ACME currently supports three different challenge methods: HTTP-01, DNS-01, and TLS-ALPN-01.[19]

19 An SNI-01 challenge method once existed, but it was thrown to the Dread Pirate Roberts—and rightfully so.

### HTTP-01

In the HTTP-01 method, the ACME server tells the client to make the challenge token available at a specific URL at the host the certificate is for, under the directory `/.well-known/acme-challenge/`.

Suppose I use HTTP-01 to validate I control **`https://mwl.io`**. When my client files an order and selects the HTTP-01 challenge, the server will say "Here is your challenge token and your key authorization." My client must create a file named after the token and make it available in the directory **`https://mwl.io/.well-known/acme-challenge`**. This file must contain the key authorization.

A few seconds later, the ACME server checks to see if the file exists and if it contains the key authorization. If the check succeeds, the ACME server informs the client that it may send its CSRs.

This challenge only runs on port 80, but you can redirect it to HTTPS on port 443. You cannot use other ports or other protocols. It's perhaps the easiest way to use ACME. Configure a web server to offer a directory owned by your ACME client at `/.well-known/acme-challenge`.

HTTP-01 is meant for simple configurations. If you have multiple web servers behind a load balancer, your ACME client can either quickly copy the challenge file everywhere and clean them up afterwards, or tell the load balancer to direct the query to a specific server. The way you copy these files around varies depending on your environment, so few ACME clients support such functions.

Let's Encrypt policy will not allow the use of HTTP-01 to create wildcard certificates, understandably. Other CAs might choose different policies.

### DNS-01

Where the HTTP-01 challenge indicates that the ACME client has access to a host, the DNS-01 challenge declares that the agent has access to alter the domain's DNS records. This is the same principle that Google Site Verification uses. The key authorization must go in a TXT record in the target domain, under *_acme-challenge.domainname*. DNS validation requires greater integration between the ACME client and the network.

When my ACME client selects the DNS challenge to validate `mwl.io`, the ACME server responds with something like "Put this key authorization as a TXT under `_acme_challenge.mwl.io`." The ACME client inserts the record and tells the server to check. The ACME server checks the domain's DNS for the challenge entry. If the challenge exists, the server tells the client to submit its CSR.

Use DNS-01 when your web servers are not publicly accessible, when you use an alternate port for HTTP, or when you want a wildcard certificate. You can also use it when you need to run the ACME client on a host other than the destination server.

### TLS-ALPN-01

The TLS-ALPN-01 challenge method takes place entirely within TLS. *ALPN*, or *Application Layer Protocol Negotiation* , is a way for applications to request different services on a single port. Each protocol is assigned a unique protocol ID, much like a TCP/IP port. It's yet another way to multiplex protocols, this time on a single encrypted port.[20]

The ALPN challenge works much like the HTTP challenge. The destination server must run a TLS-speaking server on port 443. This server does not necessarily need to be a web server, but it must respond to ALPN requests for ACME.

20 If you want a picture of the future of IT, imagine blocking TCP/IP ports and reinventing them. Forever.

The data flow for TLS-ALPN-01 is slightly different. The client has all the information it needs to deploy this challenge before it chooses the challenge, so the client sets up the challenge and tells the server "I choose TLS-ALPN-01, and the challenge is ready for you." The server checks immediately. If the challenge is successful, the server tells the client to send the CSR.

ALPN is nice in that it runs purely in TLS. It's not nice in that the application server must support both ALPN and the ACME ALPN protocol. The earliest solutions involved pure ACME challenge responders. The ACME client would shut down the web server, start an ACME responder on port 443, renew the certificate, shut down the ACME challenge responder, and reactivate the web server. This was less than ideal, but allowed testing of the protocol. If your application doesn't support ACME over ALPN, and you can handle a few moments of downtime, you might investigate responders such as the one included in dehydrated.

Applications are adding native support for ALPN. Apache's mod_md serves as both ACME client and an ALPN responder. Nginx uses a proxy configuration to divert ACME requests from HTTPS ones. People are rapidly developing and improving ALPN solutions.

Consider TLS-ALPN-01 when you have proxies or load balancers in front of your web servers, and you want those proxies to divert ACME TLS requests to a responder. ALPN is not suitable for wildcard certificates, and the destination host must be on the public Internet.

#### Which Challenge Should I Use?

Use the simplest challenge method that meets your needs. If you need wildcard certificates with current free CAs, you must use DNS-01. Otherwise, choose between TLS-ALPN-01 and HTTP-01.

Certificate operations are not confidential. Everything in a CSR is public information; the only secret data is the private key. You can safely run ACME over unencrypted HTTP. I have a bias to err on the

side of confidentiality, however. If your application has an integrated, stable, and reliable TLS-ALPN-01 implementation, I'd recommend trying it. Similarly, if you have a complicated environment involving words like "cluster," see if your infrastructure gear supports TLS-ALPN-01.

If your environment is simple, HTTP-01 is perfectly reasonable. Most ACME clients use HTTP-01.

While ACME lets you choose between multiple challenge methods, I would encourage you to pick one and stick with it. Multiple challenge possibilities create multiple paths to failure. Instead, start trying to renew your certificates about two-thirds of the way through their lifetime. Certificates from free ACME CAs expire in ninety days, so that gives you thirty days to renew the certificate. When a transitory issue prevents you from renewing a certificate on the first attempt, you have plenty of time to try again. Or to discover how your host is misconfigured.

### *Testing ACME*

Each ACME challenge presents system administration challenges. When you're first configuring an ACME client and it's refusing to function properly, you might edit a configuration file and rerun a command a dozen times in two minutes. That's a normal part of testing a new tool or protocol.

Any free resource will be abused. If a CA offers free certificates, some selfish jerk who understands neither certificates nor the meaning of "shared resources" is going to create new certificates for all his sites every hour on the hour.

Free CAs impose strict per-account resource limits. If you hit one of those rate limits, the CA will refuse all further requests until the limit expires. Those rate limits are generally hit only by large server farms, ACME client developers, and sysadmins struggling with their first ACME deployment. Even when you get one challenge method

working reliably, adding a second method might push you up against those limits again.

That's why CAs like Let's Encrypt and Buypass offer *staging* or *test* environments. Each staging environment has a test CA. Certificates signed by staging environments are not globally trusted, but you can download a root certificate for your client. Staging environments let you test your setup without running into production resource limits.

A staging ACME server has a different URL than the production server. Changing from staging to production means changing that URL in your ACME client.

When you're first using ACME, or especially when you're trying a different challenge method, check if your chosen CA has a staging environment. You don't want your tests to hit a limit and interfere with production.

### ACME clients

As with any open protocol, you're free to choose the ACME client that best suits your needs. Some operating systems or application stacks include their own ACME client. OpenBSD has acme-client(1), which is my favorite for HTTP-01 challenges but is sadly not neatly packaged on other operating systems. Apache has mod_md for managing ACME directly from the web server. As I write this mod_md is not quite ready, but many people are invested in it. Docker has a container that integrates Let's Encrypt directly into your system without having to worry about any of the messy ACME or TLS details.[21] You should consider using clients that are tightly integrated into your application or operating system.

Certbot (`https://certbot.eff.org/`) was the first ACME client, developed by the Electronic Frontier Foundation and Let's Encrypt so people could test the protocol. While it was the first, it's not the Official Standard. It's in Python, which is included in many operating systems

21 Until something goes horribly wrong.

but an add-on on others. At this time, certbot lacks TLS-ALPN-01 support.

We'll use dehydrated (**`https://dehydrated.io/`**) as our reference implementation. It's written in shell. It is widely used and well-supported by a community as well as commercial sponsors. Its only dependencies are the trivially available bash and curl, plus core system programs like sed and grep. A tiny machine like an older Raspberry Pi can easily run dehydrated. Dehydrated's major advantage are its *hooks*, which allow easy extension to support a variety of challenges.

### *Dehydrated*

All of our reference Unixes include a dehydrated package. It's part of CentOS' EPEL repository, like all the other useful software. If your Unix doesn't have a dehydrated package, grab it from **`https://dehydrated.io`** and decompress it. It contains only documentation, shell scripts, and example configurations; you don't need to compile anything.

Dehydrated has only a few components. The main script, `dehydrated`, provides the core functionality. This is the command you'll run to perform all ACME tasks. Dehydrated needs a configuration file, just called *`config`*. (It can run without a configuration file by specifying everything on the command line, but for long term use a configuration file is more manageable.) The script checks for */usr/local/etc/dehydrated* as used on BSDs, and */etc/dehydrated/config* as most Linux flavors use, then falls back to choices like your current working directory and the directory `dehydrated` was run from. Don't rely on the latter two. If your package doesn't provide the directory, create one and put everything there. We'll use */etc/dehydrated* for our examples. Copy *`config.sample`* to this directory as a reference, then copy that to *`config`*.

### Dehydrated Hooks

Dehydrated supports no challenge methods. Instead, external *hook* scripts provide all challenge functionality. When another challenge method is developed, you need only add a new hook to support it. Dehydrated includes a hook script for simple HTTP-01 validation on a single host, `hook.sh`. If you're not using load balancers and don't need DNS validation, `hook.sh` should meet your needs.

Every load balancer and DNS server has its own management interface, and dehydrated can't provide hooks for them all. The dehydrated web site has a collection of hooks written by contributors. There are any number of hooks for third party DNS providers, including registrars, and hooks for nameservers that support nsupdate. There are hook scripts for HTTP-01 validation on popular load balancers. Check the dehydrated wiki for a selection. If you don't find a hook for your setup on the wiki, search the Internet at large.

Dehydrated automatically searches for a script called `hook.sh` in the configuration directory. (Some Unixes override this default.) Any script you place there gets run as the hook script. If you use a script other than the provided `hook.sh`, I recommend giving it a different name and setting that name in the configuration file.

If I reference `hook.sh`, I mean the HTTP-01 verification script included with dehydrated. "Hooks" or "hook" refers generically to any hook script.

### Certificate Directory and User

Dehydrated generates certificates. It needs a place to stash those certificates. You want those certificates to be accessible only to the dehydrated program and **`root`**. This means you need a user for dehydrated. If your operating system package creates this user and the home directory, use them.

Debian and CentOS expect to run `dehydrated` as **`root`** and provide working directories owned by **`root`**. No ACME client needs

access to create filesystems. I strongly recommend creating a new, unprivileged user and directory.

Rather than creating a **dehydrated** user, make a more generic one named **acme**. If you decide to change software, you can reuse this account for whatever ACME client you try next. As the managed certificates are changing information, they belong in */var*. Here I create the **acme** user, with the home directory of */var/acme*. This user has the UID and GID of 443, but you can use any free UID on your system.

You'll need to run commands as the **acme** user. Protecting this account varies depending on your Unix. If you can disable the account and still run commands as that user, do so. Otherwise, give the user a lengthy, unpleasant, random password to keep other people from easily accessing it.

```
# mkdir /var/acme
# pw groupadd -n acme -g 443
# pw useradd -n acme -u 443 -g 443 -d /var/acme/ \
   -w no -s /nonexistent
# chown -R acme:acme /var/acme/
```

Dehydrated can manage certificates and related files here.

For the rest of this book, I'll assume you're using the user **acme**. If you prefer another username, substitute it as needed. Now that you have a directory for dehydrated to play in, we can configure `dehydrated`.

### Core Dehydrated Configuration

The main dehydrated configuration file is available in */etc/dehydrated/config*. It's a series of variable assignments in the grand Unix tradition. Hash marks indicate comments. Every dehydrated install requires a few common settings, no matter which challenge method you use.

The BASEDIR variable defines the directory where dehydrated will keep all certificates, keys, and working documents. This must be owned by **acme**.

```
BASEDIR=/var/acme
```

Tell dehydrated which user and group it should run as.

```
DEHYDRATED_USER=acme
DEHYDRATED_GROUP=acme
```

Every account at a CA needs a contact email address. The CA will notify this address of impending certificate expirations, as well as major changes such as a new API address.

```
CONTACT_EMAIL=mwl@mwl.io
```

Every dehydrated install needs a list of domains to get certificates for, normally stored in the file *domains.txt*. The default location is the working directory. If there's a problem, dehydrated could trash the list. Put your list of domains somewhere that dehydrated can't touch, like the configuration directory.

```
DOMAINS_TXT= "/etc/dehydrated/domains.txt"
```

The default challenge type is HTTP-01. Tell dehydrated if you're using a different one.

```
CHALLENGETYPE= "dns-01"
```

Each challenge method has its own configuration options that you'll add to *config* as needed.

### Changing CAs

Dehydrated's default CA is Let's Encrypt, but dehydrated 0.7.0 and newer has built-in configurations for the most popular free certificate authorities. At this time I'm writing this, it has Let's Encrypt staging and production (*letsencrypt-test*, *letsencrypt*), Buypass staging and production (*buypass-test*, *buypass*), and ZeroSSL (*zerossl*). Set these with the `CA` variable. Here I point dehydrated at Let's Encrypt's staging environment.

```
CA="letsencrypt-test"
```

When using an older version of dehydrated, or you want to use a CA that dehydrated doesn't know about, set `CA` to the certificate authority's API URL. Every ACME CA provides this in its documentation.

```
CA="https://api.dreadpirateroberts.ca/directory"
```

When you change CAs and run dehydrated, it automatically registers an account at the new CA and fetches new certificates. It retains all account information for the old CA in a directory restricted to the **`acme`** user, so if you switch back it can use the old account.

### Additional Settings

Dehydrated has a variety of other settings and configurations you might need in weird circumstances.

Rather than making changes to the main configuration file, you can read in additional configuration files that override the default settings. Set a directory for override files with `CONFIG_D`, as Debian does by default.

```
CONFIG_D=/etc/dehydrated/conf.d
```

Dehydrated reads each file with a name ending in *`.sh`* and adds it to the configuration. The files are read in alphanumerical order, and entries in later files override earlier ones. Minimize how many override files you have. I find this most useful for per-challenge or per-hook configuration files, where hooks have unique variable names.

If you need special curl(1) options to reach the CA, such as using a proxy server, specify them in CURL_OPTS.

```
CURL_OPTS="-x snoop.mwl.io"
```

We'll explore other settings as needed.

### *Domain List*

The file *domains.txt* contains a list of all the domains you want to get and maintain certificates for. Each certificate has its own line, like so.

```
mwlucas.org www.mwlucas.org
mwl.io www.mwl.io
```

Every name on the list is a Server Alternative Name on the certificate. The first entry is assigned to the Common Name, and is also used as the name of the directory where the cert information will be kept. Here the version of each site without the leading **www** is the Common Name, and the others become SANs.

Common Names can only be a maximum of 64 characters, including the top-level domain. If you have a domain longer than sixty-four characters, such as my **youkeepusingthatwordIdonotthinkit-meanswhatyouthinkitmeans.com**, you must use it as a SAN and not a Common Name.

```
mwlucas.org www.mwlucas.org youkeepusingthatwordIdo...
```

Even if your line is extremely long, you cannot split it with backslashes.

You can create a dehydrated *alias* for a group of related SANs that you don't want to tie to a particular name in your configuration files. For example, I rented domain names for some of my fiction series, and want to wrap the web redirects for them in TLS. I am easily bewildered. Referring to the certificate for **immortalclay.com** in the configuration file for **montagueportal.com** will confuse me. It's much better if such certificates get a more useful, general name. Put such aliases at the end of the *domains.txt* line, preceded by a greater than symbol (>).

```
immortalclay.com montagueportal.com > fiction
```

The certificate for these two domains now uses the directory *fiction*, as discussed in "The Dehydrated Directory" later this chapter.

Aliases are especially useful for wildcard certificates. Consider this *domains.txt* entry.

```
*.mwl.io
```

This requests a wildcard certificate for all hosts under the **mwl.io** domain (but not **mwl.io** itself), and puts it in the directory **.mwl.io*. I encourage everyone to start a directory name with an asterisk once, as a learning exercise. Once you understand that, you'll appreciate the alias.

```
*.mwl.io > wildcard
```

You could use the actual hostname as the Common Name and the directory, but that's not always desirable.

```
mwl.io *.mwl.io
```

With this, we can grab our first certificate via ACME.

### *Dehydrated with HTTP-01*

Start your ACME experiments using a single web server available to the public Internet, using a free CA such as Let's Encrypt. This is the simplest possible configuration. We'll use this host to set up ACME with the HTTP-01 challenge. You must configure a directory on the web server, set up a hook script to add and remove challenges, and write a deployment script.

Start with the web server.

#### Web Server Setup

When dehydrated sends its challenge to the server, the ACME server will send back a file name and a string to put in that file. Dehydrated (or whatever client you choose) puts that file in a directory. When the server completes the challenge, dehydrated removes the file. It's simplest if you create a single filesystem directory for your ACME challenges across all your sites, and tell the web server to map that directory into that URL. As all my web sites are under */var/www*, I create */var/www/acme* and change its ownership to **acme:acme**.

This directory must be available on every site I want a certificate for as the subdirectory */.well-known/acme-challenge*. The CA challenge requests a specific file in that directory, so the directory does not need to be indexed, browsable, or linked from anywhere. This directory must be available on every site you want to get a certificate for. If I want a certificate that covers both **`https://mwl.io`** and **`https://www.mwl.io`**, the server must offer both **`https://mwl.io/.well-known/acme-challenge/`** and **`https://www.mwl.io/.well-known/acme-challenge/`**.

While I'm avoiding most application-specific instructions, here's a sample configuration for Apache.

### Apache Configuration

Every web site on this host will eventually get a certificate via ACME, and I'll be reusing the ACME configuration many times. This shrieks, "Include file!"

Most Apache configurations have an *Includes* directory, where every file ending in *.conf* gets sucked into the server configuration. It's designed for virtual hosts, where each domain gets its own configuration file. We're not creating a global configuration, but a snippet of Apache that can be included in multiple sites. This means it must not automatically get sucked in. The ACME configuration file can end in anything except *.conf*.

Here's a file *acme.config* containing the following.

```
Alias /.well-known/acme-challenge/ /var/www/acme/
<Directory "/var/www/acme/">
  Options None
  Require all granted
  AllowOverride None
  ForceType text/plain
</Directory>
```

Each virtual host that needs ACME access needs only one configuration statement.

```
Include /usr/local/etc/apache24/Includes/acme.config
```

This lets you more easily manage this directory that has absolutely nothing to do with anything else on each site.

Reload the web server and call up the challenge directory in your browser. It should forbid you to browse the contents. If you create a test file in */var/www/acme*, though, it should be available.

Your web server now supports ACME challenges.

### HTTP-01 Hook Script

Dehydrated includes a hook script for the HTTP-01 challenge method, called `hook.sh`. Your package might have already installed it somewhere. If you copy this script to the configuration directory as */etc/dehydrated/hook.sh*, `dehydrated` finds it automatically. If you put the script somewhere else, inform dehydrated with the `HOOK` variable in *config*.

```
HOOK=/usr/local/scripts/hook.sh
```

You must tell `hook.sh` where to find the filesystem that maps to your web site's *.well-known/acme-challenges* directory with the `WELLKNOWN` config file option. All of my web sites have mapped */var/www/acme* to *./well-known/acme-challenges*.

```
WELLKNOWN="/var/www/acme"
```

You are now ready to attempt HTTP-01 validation. This will get you certificates, but you must edit `hook.sh` to activate them.

### Running Dehydrated

Run all `dehydrated` commands as the user **`acme`**. You could configure sudo(1) for this, but we'll use su(1). Test it by checking your dehydrated version.

```
# su -m acme -c 'dehydrated -v'
# INFO: Using main config file /etc/dehydrated/config
Dehydrated by Lukas Schauer
https://dehydrated.io

Dehydrated version: 0.6.5
…
```

This demonstrates you can run commands as `acme`. First, register this ACME client and accept the terms.

```
# su -m acme -c 'dehydrated --register --accept-terms'
# INFO: Using main config file /etc/dehydrated/config
+ Generating account key...
+ Registering account key with ACME server...
+ Fetching account ID...
+ Done!
```

If you forget this step, dehydrated reminds you whenever you try to do anything.

Now run `dehydrated` with the `-c` flag. (If you're biased towards long options, use `--cron`.) This tells dehydrated to check the expiration dates on all existing certificates, renew everything that's going to expire within 30 days, and request any certificates that don't yet exist. You'll eventually schedule `dehydrated -c` to run weekly.

```
# su -m acme -c 'dehydrated -c'
# INFO: Using main config file /etc/dehydrated/config
Processing www.mwlucas.org with alternative names: mwlucas.org
```

Dehydrated starts by telling you which domains it's working on. This certificate doesn't yet exist, so it proceeds to create new CSRs.

```
+ Signing domains...
+ Generating private key...
+ Generating signing request...
+ Requesting new certificate order from CA...
```

Once dehydrated sends the list of domains to the ACME server, the server sends back a filename for each domain and the contents of that filename. Below, dehydrated creates the files.

```
+ Received 2 authorizations URLs from the CA
+ Handling authorization for mwlucas.org
+ Handling authorization for www.mwlucas.org
+ 2 pending challenge(s)
+ Deploying challenge tokens...
```

Now that the challenge files exist, dehydrated tells the server to go ahead and perform the challenges.

```
+ Responding to challenge for mwlucas.org authorization...
+ Challenge is valid!
+ Responding to challenge for www.mwlucas.org authorization...
+ Challenge is valid!
```

Note that the output says nothing about the type of challenge. All dehydrated cares about is that it fed the hook script the challenge information, and the hook script met the challenge.

If every challenge is met, dehydrated has demonstrated that it controls these domains. It has achieved Domain Validation. It can clean up after itself.

```
+ Cleaning challenge tokens...
+ Requesting certificate...
+ Checking certificate...
+ Done!
+ Creating fullchain.pem...
+ Done!
```

You now have certificates. Somewhere.

### *The Dehydrated Directory*

All certificate-related material winds up in a subdirectory of dehydrated's base directory, as set by `BASEDIR` in *`config`*. Mine is */var/acme*, in one of four directories.

The *`accounts`* directory contains this dehydrated install's account information. Each CA has its own subdirectory. The actual account information and dehydrated's key pair are stored in JSON files in that subdirectory.

The *archive* directory contains expired certificates, keys, and CSRs. While ACME should transparently and seamlessly auto-renew certificates, having the old certificate on hand reassures me. See "Archiving Certificates" for details.

Under *chains* you'll find cached certificate chain files. Dehydrated caches root and intermediate certificate chain files to speed building complete chain files.

#### The Certificate Directory

Dehydrated stores certificates, CSR, keys, and assembled chain files in the *cert* directory. Each domain has its own directory, named after the certificate's Common Name, the first entry on the line in *domains.txt*. You'll see five types of files in the domain's directory: private keys, CSRs, certificates, chain files, and full chain files. All of the certificate material is stored in files named after the epochal time the certificate was created. Dehydrated also creates symlinks to the most recent files, so that outside programs can easily access them. Take a look at these freshly created files for the brand-new certificate for **mwl.io**.

```
cert-1608225253.csr
cert-1608225253.pem
cert.csr
cert.pem
chain-1608225253.pem
chain.pem
fullchain-1608225253.pem
fullchain.pem
privkey-1608225
```

The file *cert-1608225253.pem* is the signed certificate. If you want to know when the certificate was created, use date(1) to convert 1608225253 to a human-readable date. Nobody wants to update their server configurations when the certificate gets renewed, though, so dehydrated symlinks *cert.pem* to the current certificate.

Similarly, the file *cert.csr* points at the latest CSR, *chain.pem* points at the chain file containing the CA certificate and any intermediary

certificates needed to validate the certificate, while *fullchain.pem* includes every certificate from root to host. *privkey.pem* links to the current private key file.

Putting this all together, I would tell my server to find the certificates for **mwl.io** in */var/acme/certs/mwl.io/cert.pem* and the private key in */var/acme/certs/mwl.io/privkey.pem*. Thanks to the symlinks, the application is blissfully unaware of certificate renewals.

### Archiving Certificates

After a few renewals, the certificate directory gets crowded. You might want to keep the last certificate or two around just in case, but they sure clutter up your workspace. Dehydrated has an *archive* feature that moves obsolete certificates, CSR, chains, and private keys to the archive directory.

Running `dehydrated --cleanup` or `dehydrated -gc` tells dehydrated to move unused stuff to a folder under *archive* named after the certificate's Common Name. You might have some excess CSRs or key files from earlier attempts to answer a challenge. Clean them up like so.

```
# su -m acme -c 'dehydrated -gc'
# INFO: Using main config file /etc/dehydrated/config
Moving unused file to archive directory:
  mwl/cert-1608149972.csr
Moving unused file to archive directory:
  mwl/cert-1608150103.csr
...
```

Decide how long you want to keep expired certificates, and schedule both an archive process and a `find -delete` job to clean up anything older.

### *Certificate Deployment*

Most applications only read certificate files on startup. Renewing the certificate and changing the certificate file doesn't change the certificate served by the application. You have to tell the application to reread the certificate. While scheduling `apachectl reload` every day at 3AM would work, editing the hook script gives you more elegance.

The sample `hook.sh` includes a function, `deploy_cert`, that gets called after certs are renewed. It's there specifically so you can add commands. The example shows copying new certs to where the web server expects to find them, but it's simpler to tell the web server to look where `dehydrated` puts them. The deployment stage is the place to issue that `kill -HUP` or `reload` or `restart`, whatever you need to make your server read its new certificates.

The **`acme`** user might need additional privileges to run the deployment commands. If you have trouble with something like sudo but the deployment command is a soft reload or something equally unobtrusive, you might schedule a deployment a few minutes after dehydrated.

With a deployment command in place, you should have everything you need to fully automate certificate renewal with HTTP-01 validation. Once you've achieved that, you can investigate DNS-01.

### *DNS-01 Challenges*

While the HTTP-01 challenge demonstrates that the ACME client has the ability to create arbitrary files on the web server, the DNS-01 challenge demonstrates that the ACME client can create arbitrary DNS entries in the target host's zone. It also demands integrating the ACME client more deeply into the network, which might be impossible in some organizations. It's the only way to get wildcard certificates, however, and the most straightforward way to get certificates for hosts that are not publicly available.

If you must separate your ACME client from your web server, or if you need a certificate for an application other than a web server, DNS-01 might satisfy your requirements. Moving the certificate from the ACME client to your application server is entirely your problem. This can be as easy as cp(1) or rsync(1). An automation system like Ansible or Puppet can also handle distribution, or you can deploy an ACME-specific certificate system like Anvil (https://github.com/dlangille/anvil).

The DNS-01 challenge can only be implemented by people with a working knowledge of the Domain Name Service. If you haven't worked with DNS, or if words like ddns-confgen(1) and nsupdate(1) set you running for the safety of your castle, stick with HTTP-01 or immediately involve your DNS administrator.

Fortunately, the DNS-01 challenge only requires limited access to the DNS zone. Unfortunately, not all DNS servers can implement that restricted access. Many DNS servers either permit write access to the entire zone, or block it. We'll demonstrate with BIND, which can be snugly but not completely restricted.

The DNS-01 challenge requires creating a TXT record for each SAN in the certificate, under the *_acme-challenge* subdomain. If I wanted a certificate for **`mwl.io`**, the ACME client would create a TXT record for **`_acme-challenge.mwl.io`**. If I also have a SAN of **`www.mwl.io`**, the challenge must create an identical record for **`_acme-challenge.www.mwl.io`**. Twenty certificates and two hundred SANs means creating and deleting two hundred TXT records.

You can use CNAMEs (aliases) to point these all at a single TXT record. If you're challenging several domains at once, they might interfere with one another. It depends on your ACME client.

You have a choice. You can give your ACME client permission to create and delete those TXT records in each and every domain, or you can create a CNAME for each and every name and give the ACME client access to update a single record. What is easier with your current

workflow? Which is less intrusive in your environment? Are you truly going to configure all those access controls, or all those CNAMEs? Can your DNS server tightly restrict access, or does it provide only "this program may change records in this zone" access control? If you have no fine-grained access control, you might consider deploying a single "burner" domain that the ACME client can alter, and pointing all the CNAMEs to that domain.

Both methods require about the same time and energy to initially configure, so which should you use?

If you want to renew multiple domains in a single dehydrated run, give dehydrated access to update each domain separately. Most of us only need DNS-01 for special cases, though, and can use HTTP-01 for routine certificates. Otherwise, decide based on the amount of effort creating brand new certificates will require. If gaining access to modify DNS entries requires extensive change control meetings or dealing with recalcitrant vendors, but you can easily add CNAMEs to zones, use the CNAME-based method. If your environment's authoritative DNS service is clunky or lacks fine-grained access control, use CNAMEs and a burner domain. If you have to update multiple domains with DNS-01, you might give each of them their own subdomain in your burner domain. If you can easily create subdomains and restrict access control, maybe you should set up each domain individually.

#### DNS-01 Test Environment

Many of my domains are static, each containing only one or two long-lived IP addresses. Each new SAN means either creating a new CNAME, or creating a new access control statement and enabling dynamic DNS in each. I find creating CNAMEs easier, and it can even be done in an include file.[22] Here I set up a challenge for `mwl.io` with DNS-01 challenge.

22 Include files make me happy. They evenly distribute my errors.

When dehydrated runs, the hook creates a TXT record in the child zone **_acme-challenge.mwl.io**. All of the DNS-01 challenges get directed there by the permanent aliases in each zone. Once the challenges are satisfied, the hook removes the TXT record.

We'll demonstrate this in BIND by setting up a child zone with dynamic DNS, configuring and testing access control for that child zone, setting up CNAMEs, and configuring an nsupdate-based DNS-01 hook script from the dehydrated site.

### Configuring a Dynamic Child Zone

I don't want any ACME client to have full access to any of my domains, so I'm configuring **_acme-challenge.mwl.io** as a child zone of **mwl.io**. This child zone should not be in the zone file for mwl.io; rather, that domain must delegate those entries to the child zone. I can customize the subdomain's access controls while leaving **mwl.io** aloof and untouchable.

Start with your organization's standard zone file, empty any entries except the SOA records and the nameservers. All other zone contents will be provided dynamically. Put it in BIND's working directory, probably */etc/namedb/working*. The nameserver user owns this directory and all files in it. I named this file *acme.mwl.io*.

This subdomain now needs a *named.conf* entry. The hard part of that entry will be the dynamic DNS keys, but BIND includes ddns-confgen(8) to create the keys and tell you where to put them. Here we use it to generate a key called "acme" for the zone **_acme-challenge.mwl.io**.

```
# cd /etc/namedb
# ddns-confgen -k acme
# To activate this key, place the following in
# named.conf, and in a separate keyfile on the system
# or systems from which nsupdate
# will be run:
key “acmekey” {
        algorithm hmac-sha256;
        secret "m6kKl08Q1Pja1o6ikvOOeHv8mRKQ6Wvj...";
};
```

Copy the key into a separate file, *acme.key*. All of your ACME clients need a copy of this file. Either copy this key into *named.conf* or suck it in with an include file. The rest of the output doesn't matter; we'll be implementing something far stricter than what it suggests.

Now that named knows about the key, we can tell it about our new zone and assign access permissions.

```
zone “_acme-challenge.mwl.io” {
   type primary;
   file “/etc/namedb/working/acme.mwl.io”;
   update-policy {
      grant acmekey name _acme-challenge.mwl.io TXT;
   };
};
```

The `update-policy` statement permits dynamic DNS updates, and restricts it to the key named *acme*, the zone named **`_acme-challenge.mwl.io`**, and TXT records.

Verify your nameserver configuration with `named-checkconf -z`. If you haven't scrambled everything, reload your nameserver and verify that it recognizes the new domain.

```
$ dig _acme-challenge.mwl.io @localhost soa +short
```

If this returns anything but the SOA record, check your error log. If you have a working zone, use `nsupdate` to add a TXT record as DNS-01 requires. Use `-k` to load the zone's authorization key.

```
$ nsupdate -k acme.key
> server localhost
> update add _acme-challenge.mwl.io 300 TXT toThePain
```

This should create the entry, but if you're uncertain or haven't previously used dynamic DNS updates, double-check your work with `show`.

```
> show
Outgoing update query:
;; ->>HEADER<<- opcode: UPDATE, status: NOERROR, id:  0
;; flags:; ZONE: 0, PREREQ: 0, UPDATE: 0, ADDITIONAL: 0
;; UPDATE SECTION:
_acme-challenge.mwl.io. 300   IN     TXT   "toThePain"
```

This looks like an actual zone file entry. Commit it.

```
> send
> quit
```

Verify it's in the zone.

```
$ dig _acme-challenge.mwl.io @localhost txt +short
"toThePain"
```

The key works. Now delete it, using the same process but substituting a `delete` command for the `add`. Always remove your ACME challenges when you're finished with them.

…

```
> update delete _acme-challenge.mwl.io TXT
> send
```

Repeat this test on the host that will run your ACME client. If it doesn't work, check the ACME client's access to the authoritative DNS server on UDP port 53. Once everything works, set up your aliases.

### DNS Aliases

Each host that needs a certificate must have a DNS alias pointing their **`_acme-challenge`** alias to the challenge domain, **`_acme-challenge.mwl.io`**. To get a certificate for **`example.com`** and **`www.example.com`**, I must create CNAMEs for each like so.

```
_acme-challenge.example.com.      600 CNAME _acme-challenge.mwl.io.
_acme-challenge.www.example.com.  600 CNAME _acme-challenge.mwl.io.
```

The CNAMEs must expire quickly, within five or ten minutes. You do not want old ACME challenge values drifting around the Internet, as they interfere with validating future ACME challenges.

My challenge domain must not have a CNAME for **`_acme-challenge.mwl.io`**, as that's the real entry the dehydrated hook modifies. It does need a CNAME for **`_acme-challenge.www.mwl.io`**, however.

### DNS-01 Hook Script

The hook script must deploy challenge tokens, remove challenge tokens, and trigger anything needed to activate renewed certificates. Dehydrated's wiki includes a sample hook script that uses nsupdate(1) to add and remove challenge tokens from a zone.

Dehydrated calls its hook scripts with arguments. The hook can use those arguments to build a command line that updates your zone. That's the technically correct method, but I want you to spend your brain cells understanding what the script does rather than decoding printf(1) statements. The `dns-hook.sh` script presented here is a simplified[23] variant of that script on the dehydrated page, and is also available at **`https://cdn.mwl.io/`**. Don't use this simple demonstration script in production: see what it does and how it works, and go grab a production-grade script from the many on the dehydrated wiki.

23 Simplified because this book is not called *printf Mastery*.

```
#!/bin/sh

case “$1” in
  “deploy_challenge”)
    printf 'server mail.mwl.io\nupdate add \
      _acme-challenge.mwl.io 300 TXT \”%s\”\nsend\n' \
      "${4}" | nsupdate -k /etc/dehydrated/acme.key


    ;;
  “clean_challenge”)
    printf 'server mail.mwl.io\nupdate \
      delete _acme-challenge.mwl.io 300 TXT \”%s\”\n \
      send\n'  "{4}" | \

    nsupdate -k /etc/dehydrated/acme.key

    ;;
  “deploy_cert”)
    /usr/local/sbin/apachectl reload
    ;;
  “unchanged_cert”)
    # do nothing for now
    ;;
  “startup_hook”)
    # do nothing for now
    ;;
  “exit_hook”)
    # do nothing for now
    ;;
  esac

exit 0
```

The first argument to a hook script is the challenge step. At the `deploy_challenge` stage, the hook script needs to set up the challenge tokens. At `clean_challenge`, the hook script removes those tokens. When certificates are renewed, dehydrated calls `deploy_cert`. The hook is also called for when certificates are unchanged, plus when dehydrated is started and exits, but those are rarely used.

The second argument is the domain name being challenged. The third argument is a filename, useful for HTTP-01 but unnecessary for DNS-01. The fourth argument is the ASCII string that must go into either the file or the DNS TXT record. This is the only component we need for this demonstration script.

The `deploy_challenge` stage uses printf to build a dynamic DNS update command, and feeds that to nsupdate(1).

The `clean_challenge` stage starts by waiting. Your ACME client can run much more quickly than a Certificate Authority. Without a wait, you might clean up your challenge tokens before all of your challenges finish. After the wait, the script removes the challenge tokens.

In `deploy_cert`, we do a soft reload of the web server. If you have a complicated deployment process, you might want to move those commands into a separate script. You might need to use sudo to give the **`acme`** user permission to run that script, or schedule a deployment as another user.

This script is a fine example of fault-oblivious computing, and is for demonstration only. If you're using BIND, go grab an nsupdate-based script from the dehydrated site. If you're using another DNS server or a domain registrar's service, there's probably a script for that too.

#### Running Dehydrated with DNS-01

Now that you have all the pieces, run dehydrated. As dehydrated manages all the details for you, the command is unchanged.

```
# su acme -c 'dehydrated -c'
# INFO: Using main config file /etc/dehydrated/config
Processing mwl.io with alternative names: www.mwl.io
cdn.mwl.io
  + Signing domains...
  + Generating private key...
  + Generating signing request...
  + Requesting new certificate order from CA...
  + Received 3 authorizations URLs from the CA
  + Handling authorization for cdn.mwl.io
  + Found valid authorization for cdn.mwl.io
  + Handling authorization for mwl.io
  + Found valid authorization for mwl.io
  + Handling authorization for www.mwl.io
  + Found valid authorization for www.mwl.io
  + 0 pending challenge(s)
  + Requesting certificate...
  + Checking certificate...
  + Done!
  + Creating fullchain.pem...
```

You now have certificates. Or error messages. Both are useful, although the former is more convenient.

#### DNS-01 Collisions

The hook script removes all existing DNS-01 challenge tokens and puts in its own tokens. If you run dehydrated on several servers simultaneously, they'll fight over control of your dynamic zone.

The easiest way around this problem is to not run dehydrated simultaneously on multiple servers. If you have numerous domains on each server, and each takes a while to run, schedule them carefully or create a separate update zone for each server.

### *Per-Domain Configurations*

Every organization has that one annoying domain, with its oh-so-special requirements and its petulant insistence on being *special*. Maybe most of your domains work perfectly with HTTP-01 challenges, but this one annoying domain needs a wildcard certificate

and so must use DNS-01. Dehydrated lets you set a directory for per-domain configuration files with the `DOMAINS_D` option in *`config`*.

```
DOMAINS_D=/etc/dehydrated/domains.d/
```

The per-domain configuration file allows setting most but not all options. You can't use a different CA; that requires a whole different dehydrated configuration directory. But you can set different challenge types, hooks, key algorithms, and per-certificate details. Only set the options that differ from your common settings. Name the per-domain configuration file after the certificate's Common Name.

The domain mwl.io needs the DNS-01 challenge, while everything else needs HTTP-01. I create */etc/dehydrated/domains.d/mwl.io* and set the challenge and hook script.

```
CHALLENGETYPE="dns-01"
HOOK=/etc/dehydrated/dns01-hook.sh
```

When I run dehydrated normally, it picks up the per-domain configuration file.

```
…
+ Using certificate specific config file!
   + CHALLENGETYPE = dns-01
   + HOOK = /etc/dehydrated/dns01-hook.sh
+ Signing domains...
…
```

I can now get one wildcard certificate without forcing all of my domains to use cumbersome DNS-01 validation.

Dehydrated has many more features. If your application can't maintain its own OCSP staples, it can download them for you. You can force use of IPv4 or IPv6, revoke certificates, choose algorithms, and more. Check the documentation for the full details, but this should get you started.

### ACME Renewals

The real magic of ACME comes at renewal time. Free ACME certificates expire in ninety days. Most free CAs will renew certificates within thirty days of expiration. Each time your ACME client runs, it checks the expiration date of each of its certificates. If the certificate expires within thirty days, it automatically renews the certificate using the exact same process it used to request that certificate.

Once you have a working client, renewing your certificate means scheduling it to run automatically. A certificate check once a week gives the client four attempts to renew the certificate before it expires. This should be more than sufficient, unless you hit transient issues every week at exactly that moment.

Now that you have automatic certificates, let's discuss some ways to further lock down TLS.

# Chapter 8: HSTS and CAA

Once upon a time, web sites that needed TLS and an X.509 certificate were special. In theory, users noticed that lock icon in the address bar and said "Aha! This is a high-class, trustworthy establishment and I can confide my credit card with them." A certificate meant class.

The advent of ACME and free CAs put certificates everywhere. That lock icon is available to everyone, without a budget or even an excuse. Even us minor authors use TLS on our web sites. Web browsers are starting to flag sites without TLS as "non-secure," whatever that means. The little lock icon is once again meaningless and can be safely ignored.

Omnipresent TLS certificates still leave gaps for attackers to weasel through, however. Two tools to address these gaps are HTTP Strict Transport Security and Certification Authority Authorization records. Neither of these operate within TLS, but leverage other protocols to shore up TLS.

## *HTTP Strict Transport Security*

A *downgrade attack* is when an attacker forces a client's TLS connection to use broken algorithms, broken TLS versions, or fall back to unencrypted communications. The best-known downgrade attack is the *man-in-the-middle*, where an attacker directs victims to a non-TLS version of the web site run by the attacker and proxies user requests to the real site, capturing all data in the process. The user probably won't even notice that the site is no longer using TLS, and most of those who notice its absence will shrug it off. Even if your server redirects all HTTP connections to HTTPS, someone who can man-in-the-middle your clients can replace your redirection.

*HTTP Strict Transport Security* (*HSTS*) is designed to resist these downgrade attacks by informing clients that a non-TLS version of the site does not exist. When a web server sends the HSTS response header to the client, it's telling the client that this site can only be accessed over TLS. The client automatically rewrites any non-TLS requests to be TLS-only. If someone tries to use that client to access the HTTP version of the web site, the client refuses. When an intruder sets up their non-TLS version of your site and steers your victim there, the browser rejects it. Hopefully the email from an annoyed user will alert you that someone is attacking you.

The client caches this header for a number of days, along with other per-site data like cookies and images and so on. If the client flushes its cache, it also loses the HSTS setting.

### HSTS Drawbacks

Once you enable HSTS, you're committed. Clients that cache that header will only connect to your site with TLS. If you have a problem with your certificate, tough. You can set your site to vanilla HTTP while you work on the issue, but clients that previously visited your site and have a cached HSTS header will refuse to connect. You must fix your TLS problem before anything else. You'll probably have to flush your browser cache to debug the issue.

Clients apply HSTS to every web site on the host, not just the main web site. If I'm running `https://mwl.io`, but I also have the unencrypted site `http://mwl.io:8080`, turning on HSTS will break my unencrypted site. The client puts an *https://* in front of its requests even if the user hand-types *http://*. Some applications use private web servers to provide management interfaces on high-numbered ports. If those web servers can't speak TLS and don't accept certificates, your client can't use them. You *can* use an alternate host name for the same IP address to access those ports, however.

If you've enabled the `includeSubDomains` header on your main site, HSTS applies to everything in the domain, on any host, on any port. Enabling HSTS with `includeSubDomains` on **`https://mwl.io`** tells the browser to only use HTTPS on **`www.mwl.io`**, **`unencrypted.mwl.io`**, and every other subdomain of **`mwl.io`**.

### Deploying HSTS

Configuring HSTS on popular web servers like Apache and nginx is straightforward. The HTTP response header `Strict-Transport-Security` establishes HSTS. It has three sub-values. The first two, `max-age` and `includeSubDomains`, are most critical. The `max-age` value tells the client how many seconds to cache the header for. Once the header expires, the client may again attempt unencrypted connections. The `includeSubDomains` value tells the client that the header applies to all subdomains. If **`mwl.io`** sends HSTS, the `includeSubDomains` value tells the client HSTS applies to **`www.mwl.io`** and **`hackersite.mwl.io`** as well. Use these two values to deploy HSTS.

If you're deploying a new web site, HSTS it immediately and fix any problems during testing. Set `max-age` to 31536000 seconds (one year) and set `includeSubDomains`. Here's an example for Apache.

```
Header always set Strict-Transport-Security
  "max-age=31536000; includeSubDomains"
```

Enabling HSTS on an existing web site is trickier. Jumping on the HSTS train and pushing the throttle all the way forward feels tempting. Your web site already runs TLS, why wouldn't you want to always use it? Caching connection information client-side is like climbing a rope up an insanely high cliff. Everything goes great until you're halfway up and someone cuts the rope.

I strongly encourage you to assume that your existing web site is just as bad as everything else on the Internet, and that deploying HSTS will expose previously unknown problems. What happens if your organization's web site is down for five minutes? An hour? A

day? When you first deploy HSTS on an existing site, set `max-age` to something survivable. Also set `includeSubDomains`. If HSTS on a subdomain is going to break your web site, you want to know immediately. Run with that under actual load for at least twice `max-age`. If nothing shows, increase it to a day, a week, a month.

The last value, `preload` tells the client that this site is on the Chrome preload list. We discuss that in the next section. Never enable `preload` before you're ready to submit your site to that list!

### HSTS Preload

One problem with HSTS is that it doesn't take effect until the first time the client contacts the web site. Someone reaching out to my web site for the first time could suffer a man-in-the-middle attack, and HSTS won't stop it because the client hasn't cached that header list.

Chrome contains a list of web sites that always use HSTS. It's called the *preload* list, because it's preloaded into the browser. Hosts on the preload list have a minimum `max-age` of one year. Many other browsers use the Chrome list; it's become a de facto standard.[24] Browsers that use the preload list don't ever try to access the HTTP version of a site; they always and only connect to the TLS site.

You can get your site added to this list. There's a web form. Once you're on the list, though, you're on it forever. Yes, Chrome has a form to remove you from the list. Chrome respects that list, but removals will not reach Chrome users for months and might never reach users of other browsers. Consider the preload list a one-way trip. My web sites have been TLS-only for years now, and I have not submitted most of them to the preload list.

Once you ask Chrome to add your site to the preload list, add the `preload` header to your HSTS configuration.

---

24 Yes, it's bad for one vendor to maintain the standard preload list. It's also bad for each vendor to maintain its own list, as with root certificates. Everything is bad.

### *Certification Authority Authorization*

Intruders might try to get X.509 certificates for a host in your domain. Ignorant or indifferent employees might try to get certificates from a CA that does not meet your regulatory compliance. The *Certification Authority Authorization* (*CAA*) DNS record is a public statement of which CAs may issue certificates for a domain.

When a CA receives a certificate request, it checks the domain for a CAA record. If the CAA record exists and names that CA, it can issue the certificate. If the CAA record names a different CA, however, the certificate request is rejected. Today, absence of a CAA record allows the CA to issue a certificate. The record is intended to eventually become mandatory, so I encourage you to deploy them now before some malicious git takes advantage of their absence. If your Certificate Authority respects CAA records, they provide documentation on how to format records that let you use their service and no others.

A CAA record has this format.

```
mwl.io. IN CAA 0 keyword value
```

The primary keywords for TLS are *issue* and *issuewild*. For both, the value is the official name of a Certificate Authority.

The *issue* keyword indicates that the named CA is permitted to issue certificates for this domain. If you use multiple CAs, the zone needs multiple *issue* records.

The *issuewild* keyword means that the named CA may issue wildcard certificates for this domain. If you use wildcard certificates as well as standard certificates, you need both the *issue* and *issuewild* statements.

To prevent anyone from issuing a certificate matching that keyword, use a semicolon.

The value must precisely match the CA. The easiest way to figure it out is to check your CA's web site.

Here are CAA records for **`mwl.io`**. I allow one CA, Let's Encrypt, to issue certificates. I explicitly forbid issuing wildcard certificates.

```
mwl.io.  3600  IN  CAA  0 issue "letsencrypt.org"
mwl.io.  3600  IN  CAA  0 issuewild ";"
```

Here, Let's Encrypt may issue standard certificates but not wildcard certificates. Only Miracle Max may issue wildcard certificates.

```
mwl.io.  3600  IN  CAA  0 issue "letsencrypt.org"
mwl.io.  3600  IN  CAA  0 issuewild "miracle.max"
```

Hostnames inherit the domain's record, unless explicitly overridden by additional CAA records.

A CAA record can help you implement organization policies, and might prevent an intruder from gaining a deeper hold on your network. They work best if you protect your records with DNSSEC.

Now let's see how to investigate TLS issues.

# Chapter 9: TLS Testing and Certificate Analysis

Your web server is configured perfectly, with rock-solid TLS? *Prove it*.

The difference between "works" and "works well" is tricky. It's easy to see if clients can access a web site or download email. Maybe your application provides a handy lock icon or flashes a notice that says, "connecting securely," whatever *that* means.[25] Public-spirited folks have offered a variety of online tools to investigate and diagnose TLS connections. Most of them focus on web servers, although some apply purely to X.509 certificates. I'll discuss the three most vital services.

## *Server Configuration Testing*

SSL Labs is a free service that lets you audit and test your server's TLS configuration. They also let you test your browser's confidentiality, integrity, and non-repudiation features, and they provide an assessment of world-wide TLS configuration. Their service is so useful, I'm inclined to even forgive their use of the word "SSL."

To evaluate your server visit `https://www.ssllabs.com`, select "Server Test" and enter your site's URL. Their server makes many requests to your site in a short time, analyzes the results, and provides a report that includes a letter grade from A to F.

The first time you run this scan, the results might well horrify you. Most web servers have default configurations that focus on making sites available to the widest variety of clients, rather than providing modern robust TLS. Default configurations open you up to a variety

25 Nothing. It means *nothing*.

of attacks, and SSL Labs details your vulnerability to many of them. In painful detail. Fortunately, fixing everything on the report is straightforward with minor configuration changes. Modern web servers all have options to disable obsolete TLS and SSL versions, use only high-quality algorithms, and enforce cipher ordering. Accomplishing these drags your grade up to A.

Maybe your management has decided that the organization's ecommerce site must support the Netscape browser and SSL version 1. They don't work on the modern Internet, and most web sites don't render properly, but you've had to lug the carcass of these obsolete clients around in the hope that a miracle will happen and they'll generate revenue. Supporting these clients doesn't only mean supporting dangerous and obsolete SSL, but also allowing a variety of exploits against your server. Once you've cleaned up everything you can, present a report explaining that support for these ancient clients puts all of your customers at risk. It might not change their mind, but when the inevitable headline hack happens and the organization tries to blame you, you'll have documentation that you warned management. It might not save your job, but at least you'll be able to explain to the press *I told them so*.

Encryption standards evolve. A configuration that earned an A+ in 2016 is unacceptable today. If you change nothing, the grade will slowly drop. When you do update your server or change its configuration, you might inadvertently reduce its TLS quality without realizing it. Test your site after every upgrade and every time you reconfigure the server.

Clumsy intruders who probe your network for weaknesses send many queries using different TLS versions and ciphers in a very short time, exactly like SSL Labs. If you have an intrusion detection system that automatically blocks such probes, it might also block these tests.

### *Private Testing*

While it's great that SSL Labs offers a public testing service, you also need the ability to test servers that aren't on the public Internet and non-web servers. In most environments I prefer testssl (**`https://testssl.sh`**). The `testssl.sh` program performs checks much like SSL Labs' scanner. It requires only Bash and standard Unix utilities, and most operating systems have packages for it. Running a complete scan against your site is as simple as:

```
$ testssl.sh https://mwl.io
```

It evaluates the TLS and SSL versions available on the site, which ciphers are available, session tickets, and vulnerabilities. At the end, it simulates connections from a variety of operating systems and browsers back to Android 4.4 and Windows XP with Internet Explorer 6.

It also lets you probe non-HTTP applications. The `--starttls` or `-t` flag lets you interrogate applications that use STARTTLS, like POP3, MySQL, and LDAP. Give the protocol as an argument.

```
$ testssl.sh -t imap mail.mwl.io:993
```

Testssl can produce JSON or HTML output, take a list of targets from a file, and far more. I could fill an entire chapter with tweaking and tuning testssl. Those who need to perform regular scans, especially of an entire network, should investigate testssl.

If you prefer a web-based internal system, cryptcheck (**`https://cryptcheck.fr`**) provides similar services as testssl and SSL Labs. Cryptcheck's front end is written in Rails, while SSL Labs uses Go, so you can choose the implementation you loathe the least.

## *Certificate Transparency*

Certificate authorities are imperfect. Certificate authorities can be fooled, hacked, and abused. Certificate authorities have incorrectly issued certificates, allowing malicious actors to masquerade as organizations like Google and Microsoft.

While nothing can prevent all possible fraud, *Certificate Transparency* reduces it. All reputable public CAs record the certificates they sign and make these *certificate logs* public. Auditors can go through the records and verify that each CA is providing certificates correctly.

Transparency also shows if a certificate has been revoked. If you're writing a vital application and you need faster revocation checking than that provided by CRLs, OCSP stapling, and proprietary browser systems, you might finagle API access to a certificate transparency server and get near-real-time revocation information.

Eventually, TLS clients will reject certificates that were not submitted to a transparency server.

### Finding Bogus Certificates

For us lowly sysadmin types, certificate transparency allows us to see the certificates issued for our domains. You can verify that they're all legitimate. The easiest way to perform this search is through a site like `https://crt.sh` or the Google Transparency Project. A search on certificate transparency logs brings up many candidates. Go to the site and enter a domain name to see all the certificates issued to that name and the issuer. It should contain only certificates issued by your usual CAs. If in amidst them you discover a certificate signed by "Malevolent Six-Fingered Lackey's Certificate Authority," someone's scammed a certificate for your organization out of that CA.

Like many TLS monitoring sites, **`https://crt.sh`** is provided as a public service. It's trivial to write a script that checks your domain and compares it to a list of known certificates. Nobody needs to check the list every five minutes, or even every day. It does provide an ATOM feed for searches, however.

If you want to perform serious analysis on the certificate transparency logs, the **`http://crt.sh`** source code is available. Build your own log server.

#### Certificate Transparency in Certificates

Your certificate contains proof that it was submitted to a Certificate Transparency log. This information could also be contained in a stapled OCSP certificate or in a TLS extension, but I see it most often in the certificate itself.

When a CA decides to sign your certificate request, but before it sends you the certificate, it submits a preliminary certificate to a Certificate Transparency Log. The log returns the preliminary certificate with its own digital signature, the Signed Certificate Timestamp (SCT). The CA copies the SCT into the real certificate and signs it. Anyone who examines the certificate can see that it was properly logged. Chapter 3 has an example. Certificates can be submitted to multiple logs, and might have multiple SCTs.

TLS clients increasingly look for SCTs in certificates. Eventually, they will reject certificates without them. A client that encounters a certificate from a well-known trust anchor that seems otherwise valid but lacks SCTs should be suspicious. Browsers will soon notify CAs of such certificates.

### What Failure Looks Like

All well designed applications behave similarly, but poorly designed ones all fall apart in their own way. Everyone eventually finds themselves pushing the limits of applications or deciding *how* things should fail. What you need is a whole bunch of bad examples, edge cases, and stuff that's technically legal but not truly supported.

That's where `https://badssl.com` comes in. The main site is innocuous, but it contains links to a whole bunch of deliberately misconfigured sites. Technically, you can have a certificate with ten thousand alternative names, but will a real browser support that? The badSSL site lets you easily test it. You can fire up every browser and see how they react to current and previous TLS standards.[26]

If this isn't enough annoyance for you, you can consider running your own CA.

26 I considered testing Chrome, Firefox, and Safari against a variety of TLS errors and including the results in this book, but the mere existence of such documentation would compel each browser to change.

## Chapter 10: Becoming a CA

Certificate Authorities are run by people. No, not people like you and I. Running a CA requires both discipline and meticulous attention to detail, qualities most of us only think we have. When given a choice between using an external Certificate Authority and running your own, you should almost certainly use an outside one for public facing systems.

Sometimes you have no choice but to run your own, however.

Many applications and devices, such as VPNs and wireless access points, use client certificates for authentication. They ship with a CA for generating those certificates. These CAs are designed to get you up and running quickly, which is nice. They rarely document long-term operation, however, which is *not* nice. Some of these quick-start CAs include scripts that sabotage long-term operation. If you're using one, understanding how it works and how to operate a CA in the long term is invaluable. Most of these "included" CAs are built on OpenSSL.

In a small organization or home lab, you might run a private CA and deploy your root certificate to your systems. Larger organizations might find it worthwhile to deploy a full certificate authority software suite and pay a commercial CA for a name-constrained signing certificate. A few of you might want to get in on the racket and build your own commercial Certificate Authority.

We'll discuss each of these in turn.

### *Private Trust Anchors*

Private trust anchor CAs create their own self-signed certificates, and use them to sign other certificates. Clients do not trust these certificates until you install the private root certificate on them. An automation system like Ansible or Active Directory can configure clients for you. If you have only a couple clients, you could run around and manually install the certificate everywhere.

Running a certificate authority is an amount of work directly proportional to the number of clients, servers, and applications that need certificates. Global firms that run private CAs have staff dedicated to that work. You should not begin that work lightly. If installing and running a private CA is a lot of work, ripping out that CA in favor of self-signed certificates everywhere can be even more work.

The first vital question is, where will you run your CA? If this is your test lab and you're building a CA for your own edification, a virtual machine is fine. A real CA, used by a real organization, should be more tightly protected. If you were building a public CA and wanted to offer certificate signing to the world, the system holding the root certificate would probably be disconnected from the Internet and locked in a vault. Scale requirements to protect your CA with your environment.

Building and operating your own lab-scale CA will teach you a bunch about how X.509 and TLS work, however. I recommend it in exactly the same spirit that I recommend everyone build their own firewall at least once. It nails that knowledge into your bones.

#### CA Software

Migrating between certificate authority software suites is challenging. Choose something that will suit your needs for the long term. If your organization is larger than a handful of people, consider a tool like easy-rsa, XCA, or Dogtag to act as a full-on internal CA and support

CRLs and an OCSP responder. Large orgs might consider FreeIPA or EJBCA. Any of these would fill a book this size so we won't go into detail, but using any of them requires everything in this book.

Tiny organizations can build their own private CAs with pure OpenSSL commands. We'll look at doing that for your test lab. Before using such a CA in production, however, look at the CAs already deployed in your organization. Do you have Active Directory, or an open source tool like FreeNAS or Puppet? Are you using Hashicorp Vault to store certificates? All of these, and far more, can sign web server certificates. Cloud services often offer their internal CA or CA toolkits to their customers. Java includes a CA. You have *many* choices.

If your organization is large enough that managing certificates would require effort, consider deploying ACME internally. Let's Encrypt's ACME server, Boulder, is open source. Boulder submits to Certificate Transparency logs, however, which leak information about your servers to the public. CA software like step-ca also support ACME. You can use an intermediate certificate in your ACME server, granting you the flexibility to issue certificates either via ACME or your other tools.

### OpenSSL CAs

You can build your own CA using OpenSSL. There's even a command for it, openssl-ca(1). That manual page says, "The ca utility was originally meant as an example of how to do things in a CA. It was not supposed to be used as a full-blown CA itself: nevertheless some people are using it for this purpose." A perusal of the BUGS and RESTRICTIONS section further illuminates that OpenSSL's built-in CA tools are fine for your test lab, but it's *not* fine for a large organization. In particular, openssl-ca is not intended for multiuser use. The CA databases have no locking. CSR signing cannot filter X.509 extensions. Building a CA with OpenSSL will teach you how a CA works, however.

TLS evolved rapidly over the last quarter-century. The journey to our current protocol might look like a straight line, but it went down innumerable dead ends and doubled back. OpenSSL has lingering support for many of these dead ends. OpenSSL is so widely deployed that changing configuration syntax and defaults, just like changing the command line, would adversely impact many people. This means your configuration is going to include statements that seem like they should be defaults. You'll have options that *must* be set to a certain value in the 2020s, because they linger from the 1990s. I'm not going to explain the historical trivia of why a particular value was standardized in 2003. The manual exists for folks who want to support 1990s PKI.

OpenSSL includes an OCSP responder, openssl-ocsp(1). Don't expose it to the Internet. OpenSSL has undergone fairly heavy code audits in the last few years, but writing Internet-facing servers is not their priority. If this constraint clashes with your needs, leverage a tool intended to act as a CA.

When your OpenSSL CA starts getting complicated, don't write lengthy scripts to automate it. I know many sysadmins who have done so, and every one of them[27] regrets it. (Occasional scripts for simple common commands are fine.) Switch to a CA toolkit like easy-rsa, dogtag, or step-ca. If you must debug one of these, however, knowledge of the way a CA works and the underlying OpenSSL operations will greatly aid you.

Additionally, OpenSSL evolves. Bugs get fixed. Features are added, removed, and tweaked. When you update OpenSSL, your CA might break. Any tutorial on how to run a CA is going to break for certain users. You need to be perfectly fine with debugging, searching for answers, and studying the fine print in man pages.

After all this, if you're determined to run a minimal CA with OpenSSL, here's one way to do it.

27 I don't count that maniac who thinks ed(1) is the only editor a *real* sysadmin needs.

### *Building an OpenSSL CA*

A true Certificate Authority has several mandatory components. You might not need them all for your test lab, but if you're skipping key components you might as well forget the whole thing and accept self-signed certificates everywhere.

Your CA files should only be accessible to **`root`**. Given most Unix systems today, placing a learning CA in */root/CA* is not unreasonable.

You must have a strict organization for your certificates and keys. We'll enforce this partially in *openssl.cnf* and partly through the power of our astounding discipline, although you could use a shell script instead.

A modern CA has both a root certificate and an intermediate certificate. Vendors include the root certificate in their trust bundles. The intermediate certificate is used in day-to-day operations for issuing certificates.

Policies and control of X.509 extensions is what separates a CA certificate from a random self-signed certificate. You'll set both in *openssl.cnf*. The root certificate and the intermediate certificate require slightly different policies, and different sorts of certificates need different extensions. We'll build them piecemeal throughout this section, but complete examples are available at **`https://cdn.mwl.io`**.

Every signing certificate, whether root or intermediate, maintains databases of issued and revoked certificates. We'll use `openssl ca` for this, even though it lacks locking.

Each signing certificate also needs a Certificate Revocation List. You'll need to make the CRL available on a web server somewhere. As this is a private CA, that web server should probably be just as private.

Similarly, you'll need an OCSP responder, which needs access to the certificate databases. For a test lab you can run it on the CA machine itself. In a real environment, you wouldn't let the key signing infrastructure anywhere near the public responder.

### Root CA Organization and Defaults

The settings used for a random OpenSSL client do not apply to a root CA. Additionally, the policy used by an intermediate certificate differs subtly from the root policy. Give the root certificate its own private *openssl.cnf*. As the root certificate is going in */root/CA/root*, put this in */root/CA/root/openssl.cnf*. My sample files can be downloaded from **https://cdn.mwl.io**.

Configuration sections that apply to `openssl ca` appear in the `[ ca ]` section.

```
[ ca ]
default_ca = CA_default
```

The `default_ca` option tells OpenSSL where to look for information on the default CA. Yes, you could have multiple CAs in a single *openssl.cnf* file, but that's the sort of thing that sets your fellow sysadmins on multi-year quests for vengeance. Don't do it. Here we tell `openssl ca` that the default CA is defined in a section called `CA_default`.

```
[ CA_default ]
# Directory and file locations.
dir               = /root/CA/root
…
```

The first critical part of the CA configuration is where you put the CA. Once you define `dir`, you can assign other options using that variable. Here's a few vital directories.

```
…
certs             = $dir/certs
crl_dir           = $dir/crl
new_certs_dir     = $dir/newcerts
…
```

The `certs` option sets the directory where this CA stores critical certificates, such as the root certificate. This is set to *$dir/certs*, or */root/CA/root/certs*.

The `new_certs_dir` option tells OpenSSL where to put new certificates. We've set that to */root/CA/root/newcerts*.

Your CA needs a directory to store its Certificate Revocation List. Set this with `crl_dir`. Here, it's *$dir/crl*.

All of these directories must exist before you create certificates.

```
...
database          = $dir/index.txt
serial            = $dir/serial
...
```

The `database` and `serial` options set where OpenSSL stores the database of signed certificates and the list of certificate serial numbers.

```
...
private_key       = $dir/private/ca.key.pem
certificate       = $dir/certs/ca.cert.pem
...
```

Using `certificate` and `private_key` to set the location of the signing certificate and its private key will save you a bunch of typing. The *private* directory must exist before running any commands, but make it accessible only to **root**.

```
...
crlnumber         = $dir/crlnumber
crl               = $dir/crl/ca.crl.pem
crl_extensions    = crl_ext
default_crl_days  = 30
...
```

Hopefully your root certificate won't need to revoke any intermediate certificates, but always prepare for the worst. Each new CRL is assigned an incrementing number. The value `crlnumber` points to a file containing the next CRL number to be used. The current PEM-encoded CRL is stored in `crl`. CRLs use particular X.509 extensions, and the section `crl_ext` will enumerate the ones we need. Finally, the `default_crl_days` option says how many days a CRL is good for. You must generate and issue a new CRL before the old one expires.

```
...
name_opt            = ca_default
cert_opt            = ca_default
default_days        = 375
preserve            = no
policy              = policy_strict
```

The `name_opt and cert_opt` options control how OpenSSL displays information in the signing command. Setting these to `ca_default` tells `openssl` to use modern settings. Leaving them unset triggers obsolete, undesirable behavior.

The `default_days` option gives the standard expiration date of a certificate. Here, certificates we sign will be good for 375 days.

The `preserve` option copes with bugs in certain long-obsolete systems. Set it to `no`.

The `policy` is where things get interesting. A CA policy dictates what sorts of certificates the CA may sign. The big differences between a root certificate and an intermediate certificate appear in the policy. Our root CA gets its policy from a section called `policy_strict`.

#### Configuring CA Policies

A *policy* tells OpenSSL which certificates it should sign. It does so by comparing variables in the signing certificate and the CSR. The root certificate included in trusted bundles should only sign intermediate certificates. This means that the certificates it signs must belong to the same organization as the root certificate. We need a fairly strict policy.

```
[ policy_strict ]
countryName             = match
stateOrProvinceName     = match
organizationName        = match
organizationalUnitName  = optional
commonName              = supplied
emailAddress            = optional
```

A setting of *match* means that the setting in the CSR must be identical to that in the signing certificate. In my organization, the

`countryName` is set to US. If I try to use this certificate to sign something from Florin or Guilder, OpenSSL will reject it.

A setting of *optional* means that the value is not necessary. If a CSR contains an email address or a department, the policy doesn't care.

If it's set to *supplied*, the policy requires that the value be present in the CSR. It accepts any value, so long as it's present.

### Configuring Requests

A private CA root must generate at least one CSR, its own. We'll want to set a few basic values, like hash algorithms. The configuration needs a `req` section.

```
[ req ]
default_bits        = 4096
distinguished_name  = req_distinguished_name
string_mask         = utf8only
default_md          = sha256
x509_extensions     = v3_ca
prompt              = no
```

The `default_bits`, `distinguished_name`, and `default_md` settings should be familiar from our discussions of CSRs in Chapter 6. Setting `string_mask` to *utf8only* restricts what type of characters can appear in variables. UTF-8 has been the standard since 2003. Finally, the `x509_extensions` setting points us to the `v3_ca` section, which contains those extensions that make sense for a CA.

If you set a default private key location with `default_keyfile`, don't point it at the file containing the CA's private key. An incomplete command might overwrite your CA's private key. This would be bad.

The `req_distinguished_name` section comes straight from requesting a CSR.

```
[ req_distinguished_name ]
C = US
ST = Michigan
L = Detroit
O = Inconceivable Incorporated
OU = IT
CN = My CA Root Certificate
```

The main addition is the list of extensions for a CA, in the `v3_ca` section. These extensions are applied to certificates requested using this configuration. The x509v3_config(5) man page includes many more options, but these are the ones most common for a CA.

```
[ v3_ca ]
# Extensions for a typical CA (`man x509v3_config`).
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always,issuer
basicConstraints = critical, CA:true
keyUsage = critical, digitalSignature, cRLSign, keyCertSign
```

The *Subject Key Identifier* contains a unique code identifying particular certificates. Setting `subjectKeyIdentifier` to *hash* has been standard since 2002.

The *Authority Key Identifier* helps identify the public key used to create these certificates. It's useful when someone is migrating from one key pair to another. Setting `authorityKeyIdentifier` to *keyid:always* option tells OpenSSL to not only add this information to the certificate, but to fail if it can't comply. Adding *issuer* copies identifying information from the signing certificate to this certificate, to further help disambiguate the proper keys, but it's not set to *always* so OpenSSL can continue if it can't figure out the issuer.

The *basic constraints* dictate if created certificates can sign further certificates—that is, if it's a signing certificate. This is flagged as a critical extension, so the client must respect it. Further basic constraints are set as a variable, a colon, and a value. Here, we use `basicConstraints` to set `CA` to *true*. This configuration creates CA certificates.

With the `keyUsage` setting, we dictate what certificates created under this policy can do. You can create certificates with very narrow purposes, such as ones that can only encrypt and others that can only decrypt, or certificates that provide only non-repudiation. The important values for a CA are *digitalSignature*, *cRLSign*, and *keyCertSign*. This grants the CA the right to sign certificates and its own CRL. Many clients fail to validate `keyUsage`. The `extendedKeyUsage` option contains additional uses, as we'll see in "User Certificates" later this chapter.

An intermediate certificate should be able to sign certificates, but the certificates it signs should not be able to sign further certificates. This requires setting the X.509 basic extension `pathlen` to zero. The only way to set this additional constraint is to create a new extensions list and tell the `req` section to use it. We have to create this in the root CA's *`openssl.cnf`*, because the root CA imposes this restriction on the intermediate CA.

```
[ v3_intermediate_ca ]
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always,issuer
basicConstraints = critical, CA:true, pathlen:0
keyUsage = critical, digitalSignature, cRLSign, keyCertSign
```

The `v3_intermediate_ca extensions` list is identical to that in `[v3_ca]`, except for adding `pathlen:0` to the end of `basicConstraints`.

#### Creating the Root Certificate

You've defined the directory structure in *`openssl.cnf`*. Make sure that all those directories exist before proceeding. In addition to the *`certs`*, *`crl`*, *`newcerts`*, and *`private`* directories, also create a *`csr`* directory for storing CSRs. Be very disciplined in how you store everything, or in a few months you'll lose yourself wandering in a swamp of similar-looking files.

Additionally, a signing certificate assigns consecutive hexadecimal serial numbers to each certificate. The serial number file must exist before starting, and it needs an initial seed value. For a real CA you'd use a random number, but for ease in reading I'm starting with 1000. Similarly, you need a CRL serial number. You also need an empty certificate index file, which will become the database of signed certificates.

```
# cd /root/CA/root
# echo 1000 > serial
# echo 1000 > crlnumber
# touch index.txt
```

With this, your CA's supporting files are ready. You can create your root certificate with `openssl req`. Go to */root/CA/root*. All of our root CA operations use the `-config` flag to pull in the CA's private *openssl.cnf*. The `-newkey` and `-keyout` options tell OpenSSL to create a new key and where to put it, while `-out` tells it where to put the finished certificate. The `-x509` instructs OpenSSL to create a *self-signed* certificate. We want this certificate to be valid for 7300 days, or twenty years, and to use the X.509v3 extensions we described in the `v3_ca` section.

```
# openssl req -config openssl.cnf -newkey rsa \
  -keyout private/ca.key.pem -x509 -days 7300 \
  -extensions v3_ca -out certs/ca.cert.pem
```

You'll be prompted for a passphrase. A root certificate needs a passphrase. If this was a large commercial CA and not something for testing, you could leverage a hardware security module to store the private key instead.

Take a look at the certificate with `openssl x509`.

```
$ openssl x509 -in certs/ca.cert.pem -noout -text
```

Verify that it's a X.509 version 3 certificate, that the issuer and subject are correct, and that it contains the extensions set up in

*openssl.cnf*. If you somehow created an X.509 version 1 certificate, stop and figure out why before trying to create an intermediate CA.

The file *ca.cert.pem* is your private root certificate. Distribute it to your clients and have them install it in their trusted certificate store.

### Configuring the Intermediate CA

Now that we have a root CA we can create a subordinate, intermediate CA. Put it in */root/CA/intermediate*, and duplicate all the subdirectories we needed for our root certificate.

```
# mkdir intermediate
# cd intermediate
# mkdir certs crl csr newcerts private
# touch index.txt
# echo 1000 > serial
# echo 1000 > crlnumber
```

Rather than create a brand new *openssl.cnf*, we'll copy the one for the root certificate.

```
# cp ../root/openssl.cnf .
```

Edit *openssl.cnf* as needed for an intermediate CA. First, change file and directory locations. The directories and paths are all configured according to the variable `dir`. Changing `dir` to */root/CA/intermediate* gives everything in the intermediate CA its own location.

```
[ CA_default ]
dir    = /root/CA/intermediate
certs  = $dir/certs
…
```

Certificates signed by your intermediate CA should have a one-year expiration, just like a real intermediate CA. I'm going to add a few extra days, just in case I need to storm a castle or something. I also need to set a default algorithm.

```
default_md    = sha256
default_days  = 375
```

Further down in `CA_default`, the certificate and key files should have their own names, as well as the CRL.

```
private_key  = $dir/private/intermediate.key.pem
certificate  = $dir/certs/intermediate.cert.pem
…
crl          = $dir/crl/intermediate.crl.pem
```

One point that demonstrates how OpenSSL's `ca` subcommand was not intended for real-world use is in how it handles X.509 extensions requested by a client. One would expect to be able to filter which extensions a client was allowed to request. Instead, OpenSSL has the `copy_extensions` option. If set to *copy*, any extensions set in the CSR but not by the CA are copied. If set to *copyall*, the settings in the CSR override the CA's defaults. Setting `copy_extensions` to *copy* is the only way an OpenSSL CA can support certificates for multiple hostnames. If you're trying to use OpenSSL as a CA, you must meticulously examine the extensions on all incoming CSRs.

```
copy_extensions   = copy
```

Last in the `CA_default` section, set a policy. The intermediate certificate needs to be able to sign a wider variety of certificates than the root certificate.

```
policy  = policy_loose
```

Now describe that loose policy.

```
[ policy_loose ]
countryName             = optional
stateOrProvinceName     = optional
localityName            = optional
organizationName        = optional
organizationalUnitName  = optional
commonName              = supplied
emailAddress            = optional
```

This policy demands nothing but a Common Name. It's not so much "loose" as "indifferent." Feel free to tighten it to reflect your needs.

Finally, change the Distinguished Name. The organization information should remain unchanged, but the Common Name should reflect that this isn't a root certificate. I call this *Intermediate Certificate 1* because I might need a second or a third later.

```
[ req_distinguished_name ]
C = US
ST = Michigan
L = Detroit
O = Inconceivable Incorporated
OU = IT
CN = My CA Intermediate Certificate 1
```

The root CA has a policy that requires most of these fields match what it claims, so keep everything but the Common Name identical to the root.

We'll add extension lists to the intermediate CA configuration for each of the different types of certificate we want to sign, but this suffices to create the intermediate CA certificate.

#### Creating the Intermediate CA Certificate

Double-check that all of the necessary directories and files exist in your intermediate CA directory before proceeding. The intermediate CA will sign most of your certificates, so it requires even more file storage discipline than the root CA.

Go into `/root/CA/intermediate` to create your CSR. The command looks almost exactly like that for creating the root certificate CSR. Only the names of the created files change.

```
# openssl req -config openssl.cnf -newkey rsa \
  -keyout private/intermediate.key.pem \
  -out csr/intermediate.cert.csr
```

Enter the passphrase when prompted. It should differ from that used for your root certificate.

Now use the root certificate to sign the intermediate CA's CSR. You'll need to refer to files under both *root* and *intermediate*, so it's simplest to do this directly under */root/CA*. We'll use the `openssl ca` command. Many of the options should be familiar from creating the root certificate. The -batch option skips the various "hit yes to continue" prompts. We use `-days` to make the intermediate certificate good for only 10 years, rather than the 20 years of the root certificate. Replacing an intermediate certificate is much simpler than getting a new certificate in vendor trust bundles.

```
# openssl ca -batch -config root/openssl.cnf \
   -extensions v3_intermediate_ca -days 3600 -notext \
   -in intermediate/csr/intermediate.cert.csr \
   -out intermediate/certs/intermediate.cert.pem
Using configuration from root/openssl.cnf
Enter pass phrase for /root/CA/root/private/ca.key.pem:
```

You must use the root certificate's private key passphrase to continue.

```
Check that the request matches the signature
Signature ok
Certificate Details:
   Serial Number: 4096 (0x1000)
   Validity
      Not Before: Jan 20 15:57:55 2021 GMT
      Not After : Nov 29 15:57:55 2030 GMT
...
```

This is the first certificate signed by our CA, so it gets a nice even serial number. Following this we have the certificate details. Verify everything looks the way you want, or live with it for ten years. Finally, down at the end, we have database information.

```
...
Write out database with 1 new entries
Data Base Updated
```

What is this "database" anyway?

### Certificate Databases

The critical CA databases are the serial number and the index. The archive is also useful.

The `serial` file contains the serial number of the *next* certificate the CA issues. Once that number is assigned to a certificate, the `serial` file is moved to `serial.old` and a new `serial` file with the next serial number is created. While I used 1000 to make it easier to read, remember that these are hex numbers.

The `index.txt` file describes every certificate issued by the CA. The OCSP responder uses this database to validate requests. It has six fields, although one is most often empty. Let's look at ours.

```
V  301129155755Z    1000  unknown  /C=US/ST=Michigan/
O=Inconceivable Incorporated/OU=IT/
CN=My CA Intermediate Certificate  1
```

The first field is a single letter giving this certificate's status. *V* means Valid, *R* means Revoked, and *E* means Expired. Our root certificate is still valid.

The second field is the certificate's expiration date, in YYMMDDHHMMSS format. The trailing Z[28] indicates this timestamp is in UTC. This certificate expires on 29 November 2030, just before 4 PM.

The third field is for the certificate's revocation date. It's blank in our database. We'll see that when we revoke a certificate.

The certificate's serial number is in the fourth field.

The fifth field is the filename containing the certificate. For our CA it will always be *unknown*.

Finally, the sixth field is the certificate's Distinguished Name.

---

28 The Z is historically "Zulu Time." The name comes from the phonetic alphabet for Z, meaning "meridian zero," and has nothing to do with magnificent beadwork or speaking isiZulu.

Any time you use the `openssl ca` command to sign a certificate, it updates the databases. It also saves a copy of the certificate in the *`newcerts`* directory, named after the serial number.

Whenever you sign a certificate, back up the database files. Text databases are easily corrupted, and corrupt database files are difficult to repair. Restoring backups is much easier. Real CA software replaces text files with actual databases. The OpenSSL folks do not want to add sqlite as a dependency (reasonably enough), so we get text files and copious warnings.

### Chain File

Just as with any trusted CA that uses an intermediate signing certificate, our CA will need a chain file. This is a single file containing any intermediate certificates. Yes, you might have multiple intermediaries. It could also contain the root certificate, but that's unnecessary.

```
# cat intermediate/certs/intermediate.cert.pem > chain.pem
```

We'll use this to complete individual certificate full chain files.

### Preparing the OCSP Responder

Your OCSP responder needs a certificate, so that clients can validate its legitimacy. The intermediate CA will sign it and add the proper X.509v3 extensions. You'll regenerate this certificate every year, so create a configuration file for it. This certificate is integral to the CA's operation and will be created by the intermediate CA operator, so it's okay to store the OCSP configuration with the rest of the intermediate CA files.

Here's a sample OpenSSL configuration file for a OCSP certificate, *`ocsp.conf`*.

```
[ req ]
prompt    =no
default_bits        = 4096
distinguished_name  = req_distinguished_name
default_md          = sha256
default_keyfile     = ocsp.privkey.pem

[ req_distinguished_name ]
C = US
ST = Michigan
L = Detroit
O = Inconceivable Incorporated
OU = OCSP
CN = OCSP Responder
# openssl req -config ocsp.conf -newkey rsa \
#  -out ocsp.cert.csr
```

Note the last line of the configuration. I only use this file once a year, so I put the command to create the new CSR in the file as a comment. Run that command.

```
$ openssl req -config ocsp.conf -newkey rsa -out ocsp.cert.csr
```

Give it a passphrase and you'll have your OCSP CSR. Drag it into the intermediate CA's *csr* directory.

An OCSP certificate needs specific X.509 extensions. Define those extensions in the intermediate CA's *openssl.cnf*. Here's an `ocsp` policy.

```
[ ocsp ]
basicConstraints = CA:FALSE
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid,issuer
keyUsage = critical, digitalSignature
extendedKeyUsage = critical, OCSPSigning
```

We've seen these `basicConstraints`, `subjectKeyIdentifier`, `authorityKeyIdentifier`, and `keyUsage` settings before. The last line, `extendedKeyUsage`, is for additional roles. While the acceptable values for key usage are set in the standard, extended key usages can include any valid OID. If you can define a role in ASN.1

and assign it an OID, it can be tied to a certificate. You'll see many standard extended key usage values, including this one for signing OCSP certificates. This extension is marked critical. A client that doesn't understand the extension must reject the whole certificate.

With the policy and the CSR, we can sign the OCSP certificate. I'm running this from the *intermediate* directory.

```
# openssl ca -batch -config openssl.cnf -extensions ocsp \
    -notext -in csr/ocsp.cert.csr -out certs/ocsp.cert.pem
```

You'll be prompted for the intermediate CA's passphrase, then shown the certificate. The CA saves the certificate in the file you name after `-out`, but also keeps a copy in its master repository under the *newcerts* directory.

You must make one final decision for your OCSP responder: where it should listen to the network. If you're building a CA as an educational exercise, confine your responder to **localhost**. If your environment does not permit incoming Internet connections, you can allow it to listen to your local network. If you are on the public Internet you can use this certificate and have the responder publicly available, but don't use the OpenSSL responder. Use a third-party responder designed for Internet exposure, and attach it to CA software that uses real databases. OCSP responders normally listen on port 80.

I'm using the hostname **ocsp.mwl.io** for my OCSP responder. It points to a test IP, **203.0.113.207**. By using that hostname in future certificates, I can move the IP as needed.

Once we have revoked certificates for the responder to gripe about, we'll configure it.

### Web Site Certificates

A CA is expected to apply reasonable extensions to certificates it signs. This requires a policy in the intermediate CA's *openssl.cnf*. Here's a policy suitable for typical web sites, called `server_cert`.

```
[ server_cert ]
basicConstraints = CA:FALSE
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid,issuer:always
keyUsage = critical, digitalSignature, keyEncipherment
extendedKeyUsage = serverAuth
authorityInfoAccess = OCSP;URI:http://ocsp.mwl.io:80
crlDistributionPoints = URI:http://crl.mwl.io/intermediate.crl
```

The `basicConstraints`, `subjectKeyIdentifier`, and `keyUsage` should all be familiar by now, so we'll look at the new fields.

The `authorityKeyIdentifier` has the usual *keyid* and *issuer* keywords, but this time *issuer* is set to *always*. If OpenSSL can't figure out how to set *issuer*, the operation fails. Our CA is issuing this certificate, it better know what it is.

The `extendedKeyUsage` of *serverAuth* is a standard OID meaning that this certificate is intended only for servers. If you put such a certificate into a web or email server, it will identify the server. You can't use it for, say, your OCSP server.

Certificates can contain information about their certificate authority with the `authorityInfoAccess` keyword. Here, the CA adds the location of its OCSP responder to every server certificate it signs. Similarly, `crlDistributionPoints` puts the intermediate CA's CRL in the certificate.

Now go to your server and create a CSR for a server, exactly as discussed in Chapter 6. The CA doesn't need your server's private key information, only the CSR. If you're creating certificates in your test lab you might be using the same host for creation as hosting your CA, but at least create a new directory and keep them from cluttering your root and intermediate CAs.

Here's a configuration for a typical server certificate that my CA might be asked to sign.

```
[ req ]
prompt              = no
default_bits        = 2048
default_md          = sha256
default_keyfile     = server-private.key
distinguished_name  = req_distinguished_name
req_extensions      = v3_req

[ req_distinguished_name ]
CN = blackhelicopters.org

[ v3_req ]
subjectAltName      = @alt_names

[alt_names]
DNS.1               = blackhelicopters.org
DNS.2               = www.blackhelicopters.org
DNS.3               = freebsd.blackhelicopters.org
DNS.4               = centos.blackhelicopters.org
DNS.0               = debian.blackhelicopters.org
```

Note that this isn't for the CA's domain, **`mwl.io`**. I'm issuing certificates for a completely different domain. This is where a normal CA would be expected to verify the requester's identity, but I control both domains so I'll skip that step. Create your CSR and copy it into your intermediate CA's *`csr`* directory, go to the intermediate directory, and sign the certificate.

```
# openssl ca -batch -config openssl.cnf \
  -extensions server_cert -notext -in csr/server.csr
Using configuration from openssl.cnf
Enter pass phrase for /root/CA/intermediate/private/in-
termediate.key.pem:
Check that the request matches the signature
Signature ok
Certificate Details:
   Serial Number: 4097 (0x1001)
…
```

You might notice that I didn't tell OpenSSL where to put the created certificate. It's not lost. OpenSSL's `ca` subcommand keeps a copy of all signed certificates under newcerts, by serial number. This is certificate

1001, so it's `newcerts/1001.pem`. If you examine the certificate, you'll see the server extensions.

Return that file to your client. If you're kind, you'll combine it with your existing chain file and send a full chain file.

```
$ cat chain.pem intermediate/newcerts/1001.pem \
  >> server.fullchain.pem
```

You now have a real server certificate.

### Revoking Certificates

No learning experience is complete unless it covers failure. Configure a certificate like this.

```
[ req ]
prompt              = no
default_bits        = 2048
default_keyfile     = revoked-private.key
distinguished_name  = req_distinguished_name
req_extensions      = v3_req

[ req_distinguished_name ]
CN = microsoft.com

[ v3_req ]
subjectAltName      = @alt_names

[alt_names]
DNS.1               = microsoft.com
DNS.2               = google.com
DNS.3               = whitehouse.gov
```

Yes, yes, you're a naughty sysadmin and the CA should reject it. But pretend they miss it. Go ahead and sign it like you would a certificate for any other web site. You can now set up a web server for Microsoft, Google, and the White House and any host that trusts your certificate will believe you. You'll need a hosts entry, but it's worth doing once to drive the lesson home.

This certificate needs revoking.

OpenSSL's `ca` subcommand has a `revoke` feature. It needs only one argument, the certificate to be revoked. This particular certificate has serial number 1003. I'll use the copy in the intermediate CA's archive to revoke it.

```
# openssl ca -config openssl.cnf \
   -revoke newcerts/1003.pem
Using configuration from openssl.cnf
Enter pass phrase for
   /root/CA/intermediate/private/intermediate.key.pem:
Revoking Certificate 1003.
Data Base Updated
```

The database now lists this certificate as revoked. Note that the third field is now populated.

```
R  220131162148Z  210122132923Z  1003  unknown
   /CN=microsoft.com
```

OpenSSL's OCSP responder uses this entry to inform clients that the certificate is revoked.

#### Generating CRLs

While the OCSP responder reads the certificate database, the Certificate Revocation List is a document that clients download. Regenerate the CRL regularly, either on a schedule or whenever you revoke a certificate.

A CRL is a list of items, signed and encoded by a CA. It's essentially a special-purpose certificate, much like that for our OCSP responder. It needs particular X.509 extensions. When we set up our CA's *`openssl.cnf`*, we pointed the CRL extensions at the section `crl_ext`. It's time to define that section.

```
[ crl_ext ]
authorityKeyIdentifier=keyid:always
```

A CRL requires only the Authority Key Identifier extension. The keyword *always* tells us that if the key ID is not present, the CRL cannot be signed.

You can now create the CRL.

```
# openssl ca -config openssl.cnf -gencrl \
   -out crl/2020-01-22.crl.pem
```

OpenSSL reads the index and spits out the CRL. Name your CRLs after the date (and perhaps time) they were created. While you must copy this file to the URL given in your CA's `crlDistributionPoint`, knowing when a file was created will be invaluable in troubleshooting.

View the contents of your new CRL with `openssl crl`.

```
$ openssl crl -text -noout -in crl/2020-01-22.crl.pem
Certificate Revocation List (CRL):
   Version 2 (0x1)
   Signature Algorithm: sha256WithRSAEncryption
…
```

You'll see information about the issuer and the included X.509 extensions before getting down to a list of revoked certificates.

```
Revoked Certificates:
   Serial Number: 1003
      Revocation Date: Jan 22 13:29:23 2021 GMT
…
```

Clients can compare certificate serial numbers to the CRL and act accordingly.

Setting up a web server to provide the CRL to clients is left as an exercise for the sysadmin.

#### Client Certificates

While the command for signing a client certificate closely resembles that for signing a server certificate, a client certificate requires different X.509 extensions. Here's the intermediate CA's client configuration from *`openssl.cnf`*.

```
[ user_cert ]
basicConstraints = CA:FALSE
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid,issuer
keyUsage = critical, nonRepudiation, digitalSignature, keyEncipherment
extendedKeyUsage = clientAuth, emailProtection
authorityInfoAccess = OCSP;URI:http://ocsp.mwl.io:2560
crlDistributionPoints = URI:http://crl.mwl.io/intermediate.crl
```

The difference between this policy and one for a server certificate is the value of `extendedKeyUsage`. This certificate can be used for client authentication and email protection.[29] Apply this policy to the certificate as you sign it.

```
$ openssl ca -batch -config openssl.cnf \
  -extensions user_cert -notext -in csr/client.csr
Using configuration from openssl.cnf
Enter pass phrase for
  /root/CA/intermediate/private/intermediate.key.pem:
```

Enter the passphrase, and you'll get a certificate.

```
Check that the request matches the signature
Signature ok
Certificate Details:
        Serial Number: 4101 (0x1005)
…
```

This is certificate 0x1005. Return *`newcerts/1005.pem`* to the client.

### Private OCSP Responder

Again, don't run OpenSSL's OCSP responder on the open Internet. It's meant for education, not for production. Only play with it in a private lab or, better still, on **`localhost`**. If your CA must provide revocation notices for hosts on the Internet, and you don't want to run larger CA software, skip OCSP and only provide CRLs.

Run the OCSP responder with the `openssl ocsp` command. The `-port` flag lets you set the port it binds to. The `-text` flag tells the command to print its responses in plain text, so you can check the

29 I have no idea what your specific application thinks "email protection" means, but you get it. Congratulations.

terminal where it's running and see what happened. Use `-indexfile` to give the location of the index file and `-CA` to point to your private CA's chain file. Finally, use `-rkey` to point to your OCSP responder certificate's private key file and `-rsigner` to give the actual OCSP certificate. (If you're running LibreTLS, you can specify niceties like the checksum algorithm and an IP as well as a port.)

```
# openssl ocsp -port port -text -index indexfile \
  -CA chain.pem -rkey private-key -rsigner cert.pem
```

Here I've copied all the needed certificates, chain files, and keys from my intermediate CA into one directory and run the OCSP responder on port 80.

```
# openssl ocsp -port 80 -text -index ../index.txt \
  -CA chain.pem -rkey ocsp.privkey.pem \
  -rsigner ocsp.cert.pem
Enter pass phrase for ocsp.privkey.pem:
```

Give the OCSP key's passphrase and the responder will be ready.

```
ocsp: waiting for OCSP client connections...
```

Running is a start, but it's no replacement for *working*. Let's make an OCSP query as per Chapter 4. Open another terminal and pick one of your freshly issued certificates to verify. We'll start with the client certificate.

```
$ openssl ocsp -issuer chain.pem -cert client.cert.pem \
  -text -url http://localhost
```

Both the responder and the client spew the response. Wade through all the stuff about hashes and nonces and you'll find this delight.

```
...
Cert Status: good
...
```

Validating something that's supposed to validate is great, but double-check that OCSP can also reject certificates. We revoked a certificate earlier this chapter. Let's check it.

```
$ openssl ocsp -issuer ../chain.pem \
   -cert revoked.cert.pem -text -url http://localhost
…
Cert Status: revoked
…
```

It works.

You now have a functional test CA. Feel free to play with it, experiment with certificates, and dig into the innards of X.509, public key encryption, and TLS. Remember that an OpenSSL CA is only intended for testing and education, not for heavy-duty use.[30]

### *Name Constraint CAs*

Certificate revocation is dubious, as discussed in Chapter 4. Some industries must be able to effectively remove trust in a certificate. Browsers that contact the CA's OCSP responder will probably assume the certificate is okay. OCSP staples effectively create certificates that expire in seven days, but a week is plenty of time to carry out a bank account heist on a selected target.

Given how TLS functions, the best way us lowly end users can improve the protocol's robustness is to shorten certificate life. If your public certificates are real, and must be validated by a CA widely recognized by the public, and you want to reduce certificate life, you might consider a signing certificate with a name constraint.

A *name-constrained signing certificate* allows an organization to sign certificates within a limited list of DNS names in the Subject Alternative Name. This is a *dNSName Constrained* certificate. You can also get issuing certificates with name constraints on the Distinguished Name, IP addresses, email addresses, and other certificate identifiers. If I purchased a name constrained certificate for my domain `mwl.io`, I could sign certificates for that domain and any host in that domain.

30 You're going to ignore me and roll this delicate flower into production, aren't you? Don't give me that innocent look, I know better. This entire section of this book is a disservice to humanity.

I can issue those certificates with any conditions and usages I like. If I want my server certificates to expire in twenty-four hours, I can do that. The automation to deploy and enable those certificates is left as an exercise for the sysadmin, as is scheduling the replacements with time to resolve any problems before the certificates expire.

Name constrained certificates are for organizations with staff and funding. Everything that applies to running a private trust anchor applies to managing a name constrained certificate. Anyone who gets your organization's private key can create certificates for any host in your network.

A certificate that allows you to sign certificates for any host or client within your domain name might be tens of thousands of dollars. Much as the cost of DV server certificates has plunged, however, many people expect the price of name constrained signing certificates to drop. For organizations with real risks, and particularly organizations that have been burned by TLS failures in the past, the expense might be considered worthwhile.

Using such a certificate requires real PKI software, not the clumsy OpenSSL CA demonstrated in the previous section. If you lack the staffing to properly deploy and manage something like Dogtag, a name constrained certificate is not for you.

This is such an edge case that I'm not going to cover it in any depth. A few of you will have need of this in your careers, though, and being aware it exists will make you a hero. Or responsible for the implosion of your employer. One of those.

### *Becoming a Global Root*

You didn't learn from running the test CA earlier this chapter, and you think you might like to build a globally recognized Certificate Authority. Technologically, this is a well understood problem. You build servers. You protect your private key on some sort of Hardware Security Module (HSM). You find an existing root CA and pay them

a pile of money to cross-sign your certificates while you complete the requirements to get your root certificate accepted in all major trust bundles.

That's your first problem: *all major trust bundles*.

At this time that's Microsoft, Mozilla, Google, Apple, Oracle, and Adobe.

Each vendor has a process for evaluating, accepting, and including an organization's root certificate in their collection. They each have their own standards. In general, prospective root certificate organizations are required to pass several different audits and agree to submit to regular future audits. Gathering and maintaining that sort of data requires a lot of time, attention to detail, and reams of double-checking.

To get rid of the nagging urge to become a public CA, dig up the current version of the CA/Browser Forum's document "Baseline Requirements for the Issuance and Management of Publicly-Trusted Certificates." This will cure most readers. If reading this document sends a warm thrill through your soul, though, you just might be one of nature's CA operators.

Becoming a globally recognized root CA might sound nifty, but it's a fancy way of declaring that you're tired of being a sysadmin and that your true love is complicated, nitpicky paperwork.[31] Success means that people all across the world will try to scam you, rip you off, and fake being someone else.

Sysadmin doesn't look so bad now, does it?

And with some TLS understanding, you can make your systems behave better. Or worse. As you wish.

---

31 While I often recommend that my readers investigate fulfilling and glamorous careers such as sewage tank scuba diving, if you're given a choice between "running a globally recognized CA" or "being a sysadmin" I'd have to tell you to flee for your life. There will be no survivors.

## Afterword

It's a bad sign when a book starts talking to me. But when I sent this pile of words off to copyedit, it whispered, "I've just sucked one year of your life away. Tell me, how do you feel? And remember, this is for posterity, so be honest."

Truth is, I'm feeling pretty okay.

TLS is a huge topic. Rather than presenting you with a complete compendium of its most intimate workings, I've tried to orient you to its nature. Numerous TLS encyclopedias, standards, and RFCs will gleefully illuminate you as to the exact specifics of *this* algorithm in *that* environment with *those* clients. You wouldn't be a sysadmin if you didn't know how to fill in all the tiny holes, but I've tried to provide a map of the territory.

Our industry collaboratively dragged SSL, then TLS, into the modern era. It's not perfect. It's obtuse, complex, frightening, infuriating, and useful all at once. I should know. Here are some documents I trudged through in researching this book: RFC 5246 (TLS 1.2), RFC 6066 (SNI, OCSP stapling, and other extensions), RFC 8446 (TLS 1.3), RFC 6347 (DTLS 1.2), RFC 5280 (X.509v3 Certificates and CRLs), RFC 6960 (OCSP), RFC 6962 (Certificate Transparency), RFC 2986 (CSRs), RFC 8555 (ACME), RFC 7633 (especially the OCSP Must-Staple tidbits but also other TLS extensions), RFC 6125 (handling Common Names and wildcards), RFC 5480 (ECC Subject Public Key Info), RFC 7301 (ALPN), RFC 8737 (the ACME tls-alpn-01 challenge), RFC 6797 (HSTS), and RFC 8659 (DNS CAA Resource Record).[32] More than any other book I've written, while I

32 FT was kind enough to compile the list for me, so I'm handing it off to you.

*read* the standards documents, my technical reviewers made it clear I did not *understand* the reasoning, context, and intent behind them. My thanks go to all of those fine folks once again.

The future of TLS is in the hands of three of the IETF's working groups: *tls*, *lamps*, and *trans*. Dig up their papers if you want a glimpse at our future.

I hope that this book leaves you less bewildered. If you're still confused, go watch *The Princess Bride* and try again.

## Sponsors

The following fine people not only paid for this book before it was done, they overpaid just to get their name in the book. As I plummeted through incomprehensible standards documents, these fine folks kept my lights on and fed the pet rats.

Thank you all.

### *Print Sponsors*

Russell Folk
William Allaire
Trix Farrar
Carsten Strotmann
Bob Eager
Chris Dunbar
Xavier Belanger
Eric LeBlanc
Rogier Krieger
Christopher Kennedy
Lucas Raab
Jan-Piet Mens
Florian Obser
Mischa Peters
Marcus Neuendorf
Dan Parriott
Bob Beck
tanamar corporation
Ariel Sanchez
Trond Endrestøl

Bernd Kohler
Andrew Vieyra
Niall Navin
Roger Winans
Bruce Cantrall
Lex Onderwater
Andrew Dekker
Stefan Johnson
Nicholas Brenckle
Adam Kalisz
Maurice Kaag
Dave Cottlehuber
Maciej Grochowski
Shaun Addison
Niclas Zeising
David Hansen
Phi Network Systems
Eden Berger
Niall Navin
Julio Morales
John Hixson
John W. O'Brien
Herwart Vargens
Craig Maloney
Brad Sliger
JR Aquino

## Patronizers

Make a living as a writer? Inconceivable!

But patronizing the arts is a prestigious way to live, with a long and glorious tradition. My Patronizers (https://patronizeMWL.com) make this lunatic career far more comfortable.

A few folks send me money by the wheelbarrow. The least I can do is list these noble folks—Kate Ebneter, Stefan Johnson, Jeff Marraccini, Eirik Øverby, and Phillip Vuchetich—in the ebook and print versions of everything.

**Never miss a new Lucas release!**

Sign up for Michael W Lucas' mailing list

https://mwl.io

## Symbols



## A



## B



## C

## D

## E



## F



## G



## H



## I

## J



## K



## L



## M



## N



## O

## P

## R



## S