THIRD EDITION

OPENSSL COOKBOOK

The Definitive Guide to the Most Useful Command Line Features

Ivan Ristić

Feisty Duck

Last update: Mon Feb 08 04:03:55 GMT 2021 (build 698)

OpenSSL Cookbook

Ivan Ristić

OpenSSL Cookbook

by Ivan Ristić

Third edition (build 698), published in February 2021.

First published in May 2013.

Feisty Duck Limited
www.feistyduck.com
contact@feistyduck.com

Technical reviewer: Matt Caswell

Production editor: Jelena Girić-Ristić

Copyeditors: Melinda Rankin, Nancy Wolfe Kotary

Table of Contents

Preface

For all its warts, OpenSSL is one of the most successful and most important open source projects. It's successful because it's so widely used; it's important because the security of large parts of the Internet infrastructure relies on it. The project consists of a high-performance implementation of key cryptographic algorithms, a complete TLS and PKI stack, and a command-line toolkit. I think it's safe to say that if your job has something to do with security, web development, or system administration, you can't avoid having to deal with OpenSSL on at least some level. The majority of the Internet is powered by open source products, and most of them rely on OpenSSL.

This book covers two ways in which OpenSSL can be used. Chapter 1, *OpenSSL Command Line*, will help users who need to perform routine tasks of key and certificate generation, and configure programs that rely on OpenSSL for TLS functionality. This chapter also discusses how to create a complete private CA, which is useful for development and similar internal environments. Chapter 2, *Testing TLS with OpenSSL*, focuses on server security testing using OpenSSL. Although sometimes time consuming, this type of low-level testing can't be avoided when you wish to know exactly what's going on.

Both chapters are borrowed from my larger work, called *Bulletproof SSL and TLS*. I decided to publish the OpenSSL chapters as a separate free book because there is a severe lack of good and easily available documentation. As is often true complex and long-lived projects, the OpenSSL documentation you can find on the Internet is often wrong and outdated.

Besides, publishers often give away one or more chapters in order to show what the book is like, and I thought I should make the most of this practice by not only making the OpenSSL chapters free, but also by committing to continue to maintain and improve them over time. So here they are.

Feedback

Reader feedback is always very important, but especially so in this case, because this is a living book. In traditional publishing, often years pass before reader feedback goes back into the book, and then only if another edition actually sees the light of day (which often does not

happen for technical books, because of the small market size). With this book, you'll see new content appear in a matter of days. Ultimately, what you send to me will affect how the book will evolve.

The best way to contact me is to use my email address, *ivanr@webkreator.com*. Sometimes I may also be able to respond via Twitter, where you will find me under the handle *@ivanristic*.

Acknowledgments

This is a short book, but it's packed with technical information. As a result, there are ample opportunities for mistakes. I am very grateful to Matt Caswell for his help to keep the mistakes away. Matt, who is a member of the OpenSSL development team, joined me as technical reviewer when I started to work on the third edition.

Various people have written to me with their thoughts and corrections. They, too, made this book better. I extend my thanks to Brian Howson, Christian Folini, Jeff Kayser, Martin Carpenter, Michael Reschly, Karsten Weiss, Olivier Levillain, and Stephen N. Henson.

My special thanks goes to my copyeditor, Melinda Rankin. She has been a pleasure to work with.

About Bulletproof SSL and TLS

Bulletproof SSL and TLS is the book I wish I had back when I was starting to use SSL. I don't remember when that was exactly, but it was definitely very early on, back when you still had to patch Apache to get it to support SSL. What I do remember is how, in 2005, when I was writing my first book, *Apache Security*, I started to appreciate the complexities of cryptography. I even began to like it.

In 2009 I started to work on SSL Labs, and for me, the world of cryptography began to unravel. Fast-forward a decade, and in 2020 I am still learning. Cryptography is a unique field in which the more you learn, the less you know.

In supporting SSL Labs users over the years, I realized that there was a lot written on SSL/TLS and PKI, but that the material generally suffered from two problems: *(1)* all you need is not in one place, making the little bits and pieces (e.g., RFCs) difficult to find, and *(2)* most of it is very detailed and low level. Many documents are also obsolete. I tried to make sense of it all and it took me years of work and study to even begin to understand the ecosystem.

Bulletproof SSL and TLS addresses the documentation gap. It's a practical book that starts with a gentle introduction and a solid theory background, but then moves to discuss everything you need for your daily work. It also provides deep coverage of certain key aspects, for example protocol attacks. For those who want even more, there are hundreds of references to research papers and other external resources.

About the Author

Ivan Ristić is a security researcher, engineer, and author, known especially for his contributions to the web application firewall field and development of ModSecurity, an open source web application firewall, and for his SSL/TLS and PKI research, tools, and guides published on the SSL Labs web site.

He is the author of three books, *Apache Security*, *ModSecurity Handbook*, and *Bulletproof SSL and TLS*. You'll often find him speaking at security conferences such as Black Hat, RSA, OWASP AppSec, and others. His latest project, Hardenize, is designed to help everyone deploy the latest network and security standards.

1 OpenSSL Command Line

OpenSSL is the world's most widely used implementation of the *Transport Layer Security* (TLS) protocol. At the core, it's also a robust and a high-performing cryptographic library with support for a wide range of cryptographic primitives. In addition to the library code, OpenSSL provides a set of command-line tools that serve a variety of purposes, including support for common PKI operations and TLS testing.

OpenSSL is a de facto standard in this space and comes with a long history. The code initially began its life in 1995 under the name SSLeay,[1] when it was developed by Eric A. Young and Tim J. Hudson. OpenSSL as a separate project was born in 1998, when Eric and Tim decided to begin working on a commercial SSL/TLS toolkit called BSAFE SSL-C. A community of developers picked up the project and continued to maintain it.

Today, OpenSSL is ubiquitous on the server side and in many client programs. The command-line tools are also the most common choice for key and certificate management. When it comes to browsers, OpenSSL also has a substantial market share, albeit via Google's fork, called *BoringSSL*.[2]

OpenSSL used to be dual-licensed under OpenSSL and SSLeay licenses. Both are BSD-like, with an advertising clause. With version 3.0 (still in development at the time of writing), OpenSSL simplified its licensing by moving to Apache License v2.0.

Getting Started

If you're using one of the Unix platforms, getting started with OpenSSL should be easy; you're virtually guaranteed to have it already installed on your system. Still, things could go wrong. For example, you could have a version that's just not right, or there could be other tools (e.g., LibreSSL) configured to respond when OpenSSL is invoked. The best approach is to check first and resort to custom compiling only if necessary. Another option is to look for a pack-

[1] The letters "eay" in the name SSLeay are Eric A. Young's initials.

[2] BoringSSL (Chromium web site, retrieved 19 July 2020)

aging platform. For example, for OS X you could use Brew[3] or MacPorts.[4] As always, compiling something from scratch once is rarely a problem; maintaining that piece of software indefinitely is.

In this chapter, I assume that you're using a Unix platform because that's the natural environment for OpenSSL. On Windows, it's less common to compile software from scratch because the tooling is not readily available. You can still compile OpenSSL yourself, but it might take more work. Alternatively, you can consider downloading the binaries from the Shining Light Productions web site.[5] If you're downloading binaries from multiple web sites, you need to ensure that they're not compiled under different versions of OpenSSL. If they are, you might experience crashes that are difficult to troubleshoot. The best approach is to use a single bundle of programs that includes everything that you need. For example, if you want to run Apache on Windows, you can get your binaries from the Apache Lounge web site.[6]

Determine OpenSSL Version and Configuration

Before you do any work, you should know which OpenSSL version you'll be using. TLS and PKI continue to develop at a fairly rapid pace, and you may find that what you can do is limited if your version of OpenSSL doesn't support them. Here's what I get for version information with `openssl version` on Ubuntu 20.04 LTS, which is the system that I'll be using for the examples in this chapter:

```
$ openssl version
OpenSSL 1.1.1f  31 Mar 2020
```

At the time of this writing, OpenSSL 1.1.1 is the main release branch and has all the nice features. On older systems, you may find a release from the 1.1.0 branch, which is fine because it can be used securely with TLS 1.2, but it won't support modern features, such as TLS 1.3. The next branch will be OpenSSL 3.0. This branch is a major update of the libraries, with substantial architectural changes and a switch to the Apache License 2.0 for better interoperability with other programs and libraries. That said, the command-line tooling, which is what I am covering here, is not expected to change much.

> **Note**
>
> Although you wouldn't know it from looking at the version number, various operating systems don't actually ship vanilla OpenSSL releases. More often than not, they contain forks that are either customized for a specific platform or patched to address various known issues. However, the version number generally stays the same, and

[3] Brew (retrieved 3 August 2020)

[4] MacPorts (retrieved 3 August 2020)

[5] Win32/Win64 OpenSSL (Shining Light Productions, retrieved 19 July 2020)

[6] Apache 2.4 VC16 Windows Binaries and Modules (Apache Lounge, retrieved 19 July 2020)

> there is no indication that the code is a fork of the original project that may have different capabilities. Keep this in mind if you notice something unexpected.

To get complete version information, use the `-a` switch:

```
$ openssl version -a
OpenSSL 1.1.1f  31 Mar 2020
built on: Mon Apr 20 11:53:50 2020 UTC
platform: debian-amd64
options:  bn(64,64) rc4(16x,int) des(int) blowfish(ptr)
compiler: gcc -fPIC -pthread -m64 -Wa,--noexecstack -Wall -Wa,--noexecstack -g -O2 ↩
-fdebug-prefix-map=/build/openssl-P_ODHM/openssl-1.1.1f=. -fstack-protector-strong ↩
-Wformat -Werror=format-security -DOPENSSL_TLS_SECURITY_LEVEL=2 -DOPENSSL_USE↩
_NODELETE -DL_ENDIAN -DOPENSSL_PIC -DOPENSSL_CPUID_OBJ -DOPENSSL_IA32_SSE2 ↩
-DOPENSSL_BN_ASM_MONT -DOPENSSL_BN_ASM_MONT5 -DOPENSSL_BN_ASM_GF2m -DSHA1_ASM ↩
-DSHA256_ASM -DSHA512_ASM -DKECCAK1600_ASM -DRC4_ASM -DMD5_ASM -DAESNI_ASM -DVPAES↩
_ASM -DGHASH_ASM -DECP_NISTZ256_ASM -DX25519_ASM -DPOLY1305_ASM -DNDEBUG ↩
-Wdate-time -D_FORTIFY_SOURCE=2
OPENSSLDIR: "/usr/lib/ssl"
ENGINESDIR: "/usr/lib/x86_64-linux-gnu/engines-1.1"
Seeding source: os-specific
```

I don't suppose that you would find this output very interesting initially, but it's useful to know where you can find out how your OpenSSL was compiled. Of special interest is the `OPENSSLDIR` setting, which in my example points to `/usr/lib/ssl`; it will tell you where OpenSSL looks for its default configuration and root certificates. On my system, that location is essentially an alias for `/etc/ssl`, Ubuntu's main location for PKI-related files:

```
lrwxrwxrwx  1 root root   14 Apr 20 11:53 certs -> /etc/ssl/certs
drwxr-xr-x  2 root root 4096 May 14 21:38 misc
lrwxrwxrwx  1 root root   20 Apr 20 11:53 openssl.cnf -> /etc/ssl/openssl.cnf
lrwxrwxrwx  1 root root   16 Apr 20 11:53 private -> /etc/ssl/private
```

The `misc/` folder contains a few supplementary scripts, the most interesting of which are the scripts that allow you to implement a private *certification authority* (CA). You may or may not end up using it, but later in this chapter I will show you how to do the equivalent work from scratch.

Building OpenSSL

In most cases, you will be using the system-supplied version of OpenSSL, but sometimes there are good reasons to use a newer or indeed an older version. For example, if you have an older system, it may be stuck with a version of OpenSSL that does not support TLS 1.3. On the other side, newer OpenSSL versions might not support SSL 2 or SSL 3. Although this is the right thing to do in a general case, you'll need support for these older features if your job is to test systems for security.

You can start by downloading the most recent version of OpenSSL (in my case, 1.1.1g):

```
$ wget https://www.openssl.org/source/openssl-1.1.1g.tar.gz
```

The next step is to configure OpenSSL before compilation. For this, you will usually use the `config` script, which first attempts to guess your architecture and then runs through the configuration process:

```
$ ./config \
--prefix=/opt/openssl \
--openssldir=/opt/openssl \
no-shared \
-DOPENSSL_TLS_SECURITY_LEVEL=2 \
enable-ec_nistp_64_gcc_128
```

The automated architecture detection can sometimes fail (e.g., with older versions of OpenSSL on OS X), in which case you should instead invoke the `Configure` script with the explicit architecture string. The configuration syntax is otherwise the same.

Unless you're sure you want to do otherwise, it is essential to use the `--prefix` option to install OpenSSL to a private location that doesn't clash with the system-provided version. Getting this wrong may break your server. The other important option is `no-shared`, which forces static linking and makes self-contained command-line tools. If you don't use this option, you'll need to play with your `LD_LIBRARY_PATH` configuration to get your tools to work.

When compiling OpenSSL 1.1.0 or later, the `OPENSSL_TLS_SECURITY_LEVEL` option configures the default security level, which establishes default minimum security requirements for all library users. It's very useful to set this value at compile time as it can be used to prevent configuration mistakes. I discuss security levels in more detail later in this chapter.

The `enable-ec_nistp_64_gcc_128` parameter activates optimized versions of certain frequently used elliptic curves. This optimization depends on a compiler feature that can't be automatically detected, which is why it's disabled by default. The complete set of configuration options is available on the OpenSSL wiki.[7]

> **Note**
>
> When compiling software, it's important to be familiar with the default configuration of your compiler. System-provided packages are usually compiled using various hardening options, but if you compile some software yourself there is no guarantee that the same options will be used.[8]

If you're compiling a version before 1.1.0, you'll need to build the dependencies first:

[7] Compilation and Installation (OpenSSL, retrieved 12 August 2020)

[8] Hardening (Debian, 3 August 2020)

```
$ make depend
```

OpenSSL 1.1.0 and above will do this automatically, so you can proceed to build the main package with the following:

```
$ make
$ make test
$ sudo make install
```

You'll get the following in `/opt/openssl`:

```
drwxr-xr-x 2 root root  4096 Jun  3 08:49 bin
drwxr-xr-x 2 root root  4096 Jun  3 08:49 certs
drwxr-xr-x 3 root root  4096 Jun  3 08:49 include
drwxr-xr-x 4 root root  4096 Jun  3 08:49 lib
drwxr-xr-x 6 root root  4096 Jun  3 08:48 man
drwxr-xr-x 2 root root  4096 Jun  3 08:49 misc
-rw-r--r-- 1 root root 10835 Jun  3 08:49 openssl.cnf
drwxr-xr-x 2 root root  4096 Jun  3 08:49 private
```

The `private/` folder is empty, but that's normal; you do not yet have any private keys. On the other hand, you'll probably be surprised to learn that the `certs/` folder is empty too. OpenSSL does not include any root certificates; maintaining a trust store is considered outside the scope of the project. Luckily, your operating system probably already comes with a trust store that you can use immediately. The following worked on my server:

```
$ cd /opt/openssl
$ sudo rmdir certs
$ sudo ln -s /etc/ssl/certs
```

Examine Available Commands

OpenSSL is a cryptographic toolkit that consists of many different utilities. I counted 48 in my version. If there was ever an appropriate time to use the phrase *Swiss Army knife of cryptography*, this is it. Even though you'll use only a handful of the utilities, you should familiarize yourself with everything that's available because you never know what you might need in the future.

To get an idea of what is on offer, simply request help:

```
$ openssl help
```

The first part of the help output lists all available utilities. To get more information about a particular utility, use the `man` command followed by the name of the utility. For example, `man ciphers` will give you detailed information on how cipher suites are configured. However, `man openssl-ciphers` should also work:

```
Standard commands
asn1parse         ca               ciphers          cms
crl               crl2pkcs7        dgst             dhparam
dsa               dsaparam         ec               ecparam
enc               engine           errstr           gendsa
genpkey           genrsa           help             list
nseq              ocsp             passwd           pkcs12
pkcs7             pkcs8            pkey             pkeyparam
pkeyutl           prime            rand             rehash
req               rsa              rsautl           s_client
s_server          s_time           sess_id          smime
speed             spkac            srp              storeutl
ts                verify           version          x509
```

The help output doesn't actually end there, but the rest is somewhat less interesting. In the second part, you get the list of message digest commands:

```
Message Digest commands (see the `dgst' command for more details)
blake2b512        blake2s256       gost             md4
md5               rmd160           sha1             sha224
sha256            sha3-224         sha3-256         sha3-384
sha3-512          sha384           sha512           sha512-224
sha512-256        shake128         shake256         sm3
```

And then in the third part, you'll see the list of all cipher commands:

```
Cipher commands (see the `enc' command for more details)
aes-128-cbc       aes-128-ecb      aes-192-cbc      aes-192-ecb
aes-256-cbc       aes-256-ecb      aria-128-cbc     aria-128-cfb
aria-128-cfb1     aria-128-cfb8    aria-128-ctr     aria-128-ecb
aria-128-ofb      aria-192-cbc     aria-192-cfb     aria-192-cfb1
aria-192-cfb8     aria-192-ctr     aria-192-ecb     aria-192-ofb
aria-256-cbc      aria-256-cfb     aria-256-cfb1    aria-256-cfb8
aria-256-ctr      aria-256-ecb     aria-256-ofb     base64
bf                bf-cbc           bf-cfb           bf-ecb
bf-ofb            camellia-128-cbc camellia-128-ecb camellia-192-cbc
camellia-192-ecb  camellia-256-cbc camellia-256-ecb cast
cast-cbc          cast5-cbc        cast5-cfb        cast5-ecb
cast5-ofb         des              des-cbc          des-cfb
des-ecb           des-ede          des-ede-cbc      des-ede-cfb
des-ede-ofb       des-ede3         des-ede3-cbc     des-ede3-cfb
des-ede3-ofb      des-ofb          des3             desx
rc2               rc2-40-cbc       rc2-64-cbc       rc2-cbc
rc2-cfb           rc2-ecb          rc2-ofb          rc4
rc4-40            seed             seed-cbc         seed-cfb
seed-ecb          seed-ofb         sm4-cbc          sm4-cfb
sm4-ctr           sm4-ecb          sm4-ofb
```

Building a Trust Store

OpenSSL does not come with a collection of trusted root certificates (also known as a *root store* or a *trust store*), so if you're installing from scratch you'll have to find them somewhere else. One possibility is to use the trust store built into your operating system, as I've shown earlier. This choice is usually fine, but the built-in trust stores may not always be up to date. Also, in a mixed environment there could be meaningful differences between the default stores in a variety of systems. A consistent and possibly better choice—but one that involves more work—is to reuse Mozilla's work. Mozilla put a lot of effort into maintaining a transparent and up-to-date root store for use in Firefox.[9]

Because it's open source, Mozilla keeps the trust store in the source code repository:

```
https://hg.mozilla.org/releases/mozilla-beta/file/tip/security/nss/lib/ckfw↩
/builtins/certdata.txt
```

Unfortunately, its certificate collection is in a proprietary format, which is not of much use to others as is. If you don't mind getting the collection via a third party, the Curl project provides a regularly updated conversion in *Privacy-Enhanced Mail* (PEM) format, which you can use directly:

```
http://curl.haxx.se/docs/caextract.html
```

If you'd rather work directly with Mozilla, you can convert its data using the same tool that the Curl project is using. You'll find more information about it in the following section.

> **Note**
>
> If you have an itch to write your own conversion script, note that Mozilla's root certificate file is not a simple list of certificates. Although most of the certificates are those that are considered trusted, there are also some that are explicitly disallowed. Additionally, some certificates may only be considered trusted for certain types of usage. The Perl script I describe here is smart enough to know the difference.

At this point, what you have is a root store with all trusted certificates in the same file. This will work fine if you're only going to be using it with, say, the `s_client` tool. In that case, all you need to do is point the `-CAfile` switch to your root store. Replacing the root store on a server will require more work, depending on what operating system is used.

On Ubuntu, for example, you'll need to replace the contents of the `/etc/ssl/certs` folder. Ubuntu ships with a tool called `update-ca-certificates` that might work. Alternatively, you could make the changes manually by replicating the structure of the existing data. From the looks of it, that folder contains the trusted certificates as individual files, as well as all of them

[9] Mozilla CA Certificate Store (Mozilla; 9 August 2020)

in a single file called `ca-certificates.crt`. You will also observe some symbolic links; they are created by the OpenSSL's `rehash` or `c_rehash` tools. The drawback of any manual changes is that they may be overwritten when the system is updated.

Manual Conversion

To convert Mozilla's root store, the Curl project uses a Perl script originally written by Guenter Knauf. This script is part of the Curl project, but you can download it directly by following this link:

```
https://raw.githubusercontent.com/curl/curl/master/lib/mk-ca-bundle.pl
```

After you download and run the script, it will fetch the certificate data from Mozilla and convert it to the PEM format:

```
$ ./mk-ca-bundle.pl
SHA256 of old file: 0
Downloading certdata.txt ...
Get certdata with curl!
[...]
Downloaded certdata.txt
SHA256 of new file: cc6408bd4be7fbfb8699bdb40ccb7f6de5780d681d87785ea362646e4dad5e8↩
e
Processing  'certdata.txt' ...
Done (138 CA certs processed, 30 skipped).
```

If you keep previously downloaded certificate data around, the script will use it to determine what changed and process only the updates.

Key and Certificate Management

Most users turn to OpenSSL because they wish to configure and run a web server that supports SSL. That process consists of three steps: *(1)* generate a private key, *(2)* create a *Certificate Signing Request* (CSR) and send it to a CA, and *(3)* install the CA-provided certificate in your web server. These steps (and a few others) are covered in this section.

Key Generation

The first step in preparing to run a TLS server is to generate a private key. Before you begin, you must make several decisions:

Key algorithm
: OpenSSL supports RSA, DSA, ECDSA, and EdDSA key algorithms, but not all of them are useful in practice. For example, DSA is obsolete and EdDSA is not yet widely supported. That leaves us with RSA and ECDSA algorithms to use in our certificates.

Key size

The default key sizes might not be secure, which is why you should always explicitly configure key size. For example, the default for RSA keys used to be 512 bits, which is insecure. If you used a 512-bit key on your server today, an intruder could take your certificate and use brute force to recover your private key, after which she could impersonate your web site. Today, 2,048-bit RSA keys are considered secure, or 256 bits for ECDSA.

Passphrase

Using a passphrase with a key is optional, but strongly recommended. Protected keys can be safely stored, transported, and backed up. On the other hand, such keys are inconvenient, because they can't be used without their passphrases. For example, you might be asked to enter the passphrase every time you wish to restart your web server. For most, this is either too inconvenient or has unacceptable availability implications. In addition, using protected keys in production does not actually increase the security much, if at all. This is because, once activated, private keys are kept unprotected in program memory; an attacker who can get to the server can get the keys from there with just a little more effort. Thus, passphrases should be viewed only as a mechanism for protecting private keys when they are not installed on production systems. In other words, it's all right to keep passphrases on production systems, next to the keys. If you need better security in production, you should invest in a hardware solution.[10]

To generate an RSA key, use the following `genpkey` command:

```
$ openssl genpkey -out fd.key \
-algorithm RSA \
-pkeyopt rsa_keygen_bits:2048 \
-aes-128-cbc
.....................................+++++
.....................................................................+++++
Enter PEM pass phrase: ************
Verifying - Enter PEM pass phrase: ************
```

Here, I specified that the key be protected with AES-128. You can also use AES-256 (with the `-aes-256-cbc` switch), but it's best to stay away from the other algorithms (e.g., DES, 3DES, and SEED).

[10] A small number of organizations will have very strict security requirements that require the private keys to be protected at any cost. For them, the solution is to invest in a *Hardware Security Module* (HSM), which is a type of product specifically designed to make key extraction impossible, even with physical access to the server. To make this work, HSMs not only generate and store keys, but also perform all necessary operations (e.g., signature generation). HSMs are typically very expensive.

> **Warning**
>
> By default, OpenSSL will set the public exponent of new RSA keys to 65,537. This is what's known as a *short public exponent*, and it significantly improves the performance of RSA verification. You may come across advice to choose 3 as your public exponent and make verification even faster. Although that's true, there are some unpleasant historical weaknesses associated with the use of 3 as a public exponent, which is why you should stick with 65,537. This choice provides a safety margin that's been proven effective in the past.

When you use the `genpkey` command, the generated private keys are stored in PKCS#8 format,[11] which is just text and doesn't look like much:

```
$ cat fd.key
-----BEGIN ENCRYPTED PRIVATE KEY-----
MIIFLTBXBgkqhkiG9w0BBQ0wSjApBgkqhkiG9w0BBQwwHAQInW7GrFjUhUcCAggA
MAwGCCqGSIb3DQIJBQAwHQYJYIZIAWUDBAEqBBBn8AErtRKB9p7ii1+g2OhWBIIE
OMnC2dwGznZqpTMXOMYekzyxe4dKlJiIsVr1hgwmjFifzEBs/KvHBV3eIe9wDAzq
[21 lines removed...]
IfveVZzM6PLbDaysxX6jEgi4xVbqWugd9h3eAPeBv9Z5iZ/bZq5hMbt37ElA2Rnh
RfmWSzlASjQi4XAHVLCs6XmULCda6QGvyB7WXxuzbhOv3C6BPXR49z6S1MFvOyDA
2oaXkfS+Ip3x2svgFJj/VpYZHUHwRCzXcDl/CdVg9fxwxcYHuJDH16Qfue/LRtiJ
hqr4fHrnbbk+MZpDaU+h4shLRBg2dONdUEzhPkpdOOkF
-----END ENCRYPTED PRIVATE KEY-----
```

However, a private key isn't just a blob of random data, even though that's what it looks like at a glance. You can see a key's structure using the following `rsa` command:

```
$ openssl pkey -in fd.key -text -noout
Enter pass phrase for fd.key: ****************
RSA Private-Key: (2048 bit, 2 primes)
modulus:
    00:be:79:08:22:1a:bc:78:3c:17:34:4a:d3:5f:2b:
    [...]
publicExponent: 65537 (0x10001)
privateExponent:
    10:20:95:54:b5:e8:d1:51:5d:31:9b:48:4c:5d:90:
    [...]
prime1:
    00:f5:3f:74:cf:ef:8f:93:e9:54:b3:79:a1:f2:91:
    5a:7e:15:13:26:f7:f9:d7:a8:f3:f9:6b:2b:90:93:
```

[11] You will often see advice to generate private keys using the `genrsa` command. Indeed, the earlier versions of this very book used this command in the examples. However, `genrsa` is a legacy command and should no longer be used. There is an entire new family of commands that deal with private keys in a unified manner (i.e., one command for all private key operations, no matter the algorithm). You should also be aware that `genrsa` outputs keys in a legacy format. Here's how to tell them apart: if you see `BEGIN ENCRYPTED PRIVATE KEY` at the top of the file, you're dealing with PKCS #8, which is the new format. If you see `BEGIN RSA PRIVATE KEY`, that's the legacy format.

```
    57:54:cc:84:c9:ea:6f:9f:39:ad:ad:60:4c:f0:68:
    16:db:1a:49:51:56:87:f1:70:ae:c9:42:89:2a:38:
    55:3e:17:a0:78:a7:52:49:10:79:cf:99:ae:53:c8:
    e0:60:5d:7e:91:26:86:3b:79:d2:70:c0:39:38:dd:
    ed:ee:75:c0:15:c6:30:51:00:a8:93:f3:8b:25:01:
    04:25:72:fc:9c:e9:73:d0:93:11:2d:82:e2:e3:d0:
    66:c0:36:2f:b6:de:de:0d:47
prime2:
    00:c6:d2:ce:66:b5:35:6b:35:d7:bb:b0:e3:f4:2d:
    [...]
exponent1:
    00:e9:2e:e9:b9:5f:f5:2b:54:fa:c5:1f:4c:7d:5f:
    [...]
exponent2:
    00:83:ea:bc:ad:a2:cf:a5:a9:9c:d0:d8:85:f6:ae:
    [...]
coefficient:
    68:18:a7:4f:aa:86:a7:e0:92:49:76:8d:24:65:fa:
    [...]
```

If you need to have just the public part of a key separately, you can do that with the following `rsa` command:

```
$ openssl pkey -in fd.key -pubout -out fd-public.key
Enter pass phrase for fd.key: ***************
```

If you look into the newly generated file, you'll see that the markers clearly indicate that the contained information is indeed public:

```
$ cat fd-public.key
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAvnkIIhq8eDwXNErTXytD
U1JGrYUgFsN8IgFVMJmAuY15dBvSCO+6y9FAOHO8utJVtHScyWeOlo1uoOTQ3RWr
Pe7W3O2SaW2gIby2cwzGf/FBExZ+BCNXkN5z8Kd38PXDLt8ar+7MJ3vrb/sW7zs2
v+rtfRar2RmhDPpVvI6sugCeHrvYDGdA/gIZAMMg3pVFivPpHnTH4AR7rTzWCWlb
nCB3z2FVYpvumrY8TvIo5OioD2I+TQyvlxDRo14QWxIdZxvPcCUxXMN9MC8fBtLu
IlllDmah8JzF2CF5IxVgVhi7hyTtSQfKsK91tAvN3OF9qkZNEpjNX37M5duHUVPb
tQIDAQAB
-----END PUBLIC KEY-----
```

It's good practice to verify that the output contains what you're expecting. For example, if you forget to include the `-pubout` switch on the command line, the output will contain your private key instead of the public key.

The process is similar for ECDSA keys, except that it isn't possible to create keys of arbitrary sizes. Instead, for each key you select a *named curve*, which controls key size, but it controls other EC parameters as well. The following example creates a 256-bit ECDSA key using the P-256 (or `secp256r1`) named curve:

```
$ openssl genpkey -out fd.key \
-algorithm EC  \
-pkeyopt ec_paramgen_curve:P-256 \
-aes-128-cbc
Enter PEM pass phrase: ****************
Verifying - Enter PEM pass phrase: ****************
```

OpenSSL supports many named curves, but for web server keys, you're generally (still) limited to only two curves that are widely supported: P-256 (also known as `secp256r1` or `prime256v1`) and P-384 (`secp384r1`). Of these two, P-256 is sufficiently secure and provides better performance. If you're curious to see a list of all named curves supported by OpenSSL, you can get it using the `ecparam` command and the `-list_curves` switch.

The recent additions x25519, x448, ed25519, and ed448 are also supported, but they are different types of curves and have to be specified using the `-algorithm` switch—for example:

```
$ openssl genpkey -algorithm ed25519
-----BEGIN PRIVATE KEY-----
MC4CAQAwBQYDK2VwBCIEIF6K3m4WM7/yMA9COn6HYyx7PjJCIzY7bnBoKupYgdTL
-----END PRIVATE KEY-----
```

Creating Certificate Signing Requests

Once you have a private key, you can proceed to create a *Certificate Signing Request* (CSR). This is a formal request asking a CA to sign a certificate, and it contains the public key of the entity requesting the certificate and some information about the entity. This data will all be part of the certificate. A CSR is always signed with the private key corresponding to the public key it carries.

CSR creation is usually an interactive process during which you'll be providing the elements of the certificate distinguished name. Read the instructions given by the `openssl` tool carefully; if you want a field to be empty, you must enter a single dot (`.`) on the line, rather than just hit Return. If you do the latter, OpenSSL will populate the corresponding CSR field with the default value. (This behavior doesn't make any sense when used with the default OpenSSL configuration, which is what virtually everyone does. It *does* make sense once you realize you can actually change the defaults, either by modifying the OpenSSL configuration or by providing your own configuration files.)

```
$ openssl req -new -key fd.key -out fd.csr
Enter pass phrase for fd.key: ****************
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
```

```
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:GB
State or Province Name (full name) [Some-State]:.
Locality Name (eg, city) []:London
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Feisty Duck Ltd
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:www.feistyduck.com
Email Address []:

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

> **Note**
>
> According to Section 5.4.1 of RFC 2985,[12] *challenge password* is an optional field that was intended for use during certificate revocation as a way of identifying the original entity that had requested the certificate. If entered, the password will be included verbatim in the CSR and communicated to the CA. It's rare to find a CA that relies on this field; all instructions I've seen recommend leaving it alone. Having a challenge password does not increase the security of the CSR in any way. Further, this field should not be confused with the key passphrase, which is a separate feature.

After a CSR is generated, use it to sign your own certificate and/or send it to a public CA and ask it to sign the certificate. Both approaches are described in the following sections. But before you do that, it's a good idea to double-check that the CSR is correct. Here's how:

```
$ openssl req -text -in fd.csr -noout
Certificate Request:
    Data:
        Version: 1 (0x0)
        Subject: C = GB, L = London, O = Feisty Duck Ltd, CN = www.feistyduck.com
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:8a:d5:de:69:30:c7:77:b0:a0:54:f7:b3:34:9a:
                    96:1c:23:81:e3:9c:0c:81:a6:8a:a5:14:76:f4:4c:
                    b3:10:cb:ee:50:d1:ea:70:e9:7f:8f:75:67:f9:12:
                    83:b0:11:e7:6c:64:de:bc:af:bd:3f:43:da:b8:41:
                    96:75:34:63:85
                ASN1 OID: prime256v1
                NIST CURVE: P-256
```

[12] RFC 2985: PKCS #9: Selected Object Classes and Attribute Types Version 2.0 (M. Nystrom and B. Kaliski, November 2000)

```
    Attributes:
        a0:00
Signature Algorithm: ecdsa-with-SHA256
     30:44:02:20:52:b9:cf:ca:d1:25:1c:b7:57:65:fb:24:5d:95:
     15:f0:39:79:36:6c:d6:0a:42:6e:26:7c:54:e8:71:17:a5:99:
     02:20:5a:e0:cd:b3:60:ec:2c:fc:29:8c:f9:21:01:08:9a:a3:
     0d:fc:9a:d3:4f:24:fb:23:4f:c6:d7:a2:14:d1:54:f9
```

Creating CSRs from Existing Certificates

You can save yourself some typing if you're renewing a certificate and don't want to make any changes to the information presented in it. With the following command, you can create a brand-new CSR from an existing certificate:

```
$ openssl x509 -x509toreq -in fd.crt -out fd.csr -signkey fd.key
```

> **Note**
>
> Unless you're using some form of public key pinning and wish to continue using the existing key, it's best practice to generate a new key every time you apply for a new certificate. Key generation is quick and inexpensive and reduces your exposure in case of a compromise that went undetected.

Unattended CSR Generation

CSR generation doesn't have to be interactive. Using a custom OpenSSL configuration file, you can both automate the process (as explained in this section) and do certain things that are not possible interactively (e.g., how to have multiple domain names in the same certificate, as discussed in subsequent sections).

For example, let's say that we want to automate the generation of a CSR for `www.feistyduck.com`. We would start by creating a file `fd.cnf` with the following contents:

```
[req]
prompt = no
distinguished_name = dn
req_extensions = ext
input_password = PASSPHRASE

[dn]
CN = www.feistyduck.com
emailAddress = webmaster@feistyduck.com
O = Feisty Duck Ltd
L = London
C = GB
```

```
[ext]
subjectAltName = DNS:www.feistyduck.com,DNS:feistyduck.com
```

Now you can create the CSR directly from the command line:

```
$ openssl req -new -config fd.cnf -key fd.key -out fd.csr
```

Signing Your Own Certificates

If you're configuring a TLS server for your own use or for a quick test, sometimes you don't want to go to a CA for a publicly trusted certificate. It's much easier just to use a self-signed certificate.[13]

If you already have a CSR, create a certificate using the following command:

```
$ openssl x509 -req -days 365 -in fd.csr -signkey fd.key -out fd.crt
Signature ok
subject=C = GB, L = London, O = Feisty Duck Ltd, CN = www.feistyduck.com
Getting Private key
Enter pass phrase for fd.key: ****************
```

You don't actually have to create a CSR in a separate step. The following command creates a self-signed certificate starting with a key alone:

```
$ openssl req -new -x509 -days 365 -key fd.key -out fd.crt
```

If you don't wish to be asked any questions, use the `-subj` switch to provide the certificate subject information on the command line:

```
$ openssl req -new -x509 -days 365 -key fd.key -out fd.crt \
 -subj "/C=GB/L=London/O=Feisty Duck Ltd/CN=www.feistyduck.com"
```

Creating Certificates Valid for Multiple Hostnames

By default, certificates produced by OpenSSL have only one common name and are valid for only one hostname. Because of this, even if you have related web sites, you are forced to use a separate certificate for each site. In this situation, using a single *multidomain* certificate makes much more sense. Further, even when you're running a single web site, you need to ensure that the certificate is valid for all possible paths that end users can take to reach it. In

[13] To be honest, getting a valid public certificate quickly has become much easier since Let's Encrypt started offering them for free in an automated fashion. We're now seeing the rise of operating systems and even software packages that seamlessly integrate with Let's Encrypt to provide public certificates out of the box. We're not very far from the moment when creating self-signed certificates will be the option that requires more work.

practice, this means using at least two names, one with the `www` prefix and one without (e.g., `www.feistyduck.com` and `feistyduck.com`).

There are two mechanisms for supporting multiple hostnames in a certificate. The first is to list all desired hostnames using an X.509 extension called *Subject Alternative Name* (SAN). The second is to use wildcards. You can also use a combination of the two approaches when it's more convenient. In practice, for most sites, you can specify a bare domain name and a wildcard to cover all the subdomains (e.g., `feistyduck.com` and `*.feistyduck.com`).

> **Warning**
>
> When a certificate contains alternative names, all common names are ignored. Newer certificates produced by CAs may not even include any common names. For that reason, include all desired hostnames on the alternative names list.

First, place the extension information in a separate text file. I'm going to call it `fd.ext`. In the file, specify the name of the extension (`subjectAltName`) and list the desired hostnames, as in the following example:

```
subjectAltName = DNS:*.feistyduck.com, DNS:feistyduck.com
```

Then, when using the `x509` command to issue a certificate, refer to the file using the `-extfile` switch:

```
$ openssl x509 -req -days 365 \
-in fd.csr -signkey fd.key -out fd.crt \
-extfile fd.ext
```

The rest of the process is no different from before. But when you examine the generated certificate afterward (see the next section), you'll find that it contains the SAN extension:

```
X509v3 extensions:
    X509v3 Subject Alternative Name:
        DNS:*.feistyduck.com, DNS:feistyduck.com
```

Examining Certificates

Certificates don't look like much in a text editor, but they contain a great deal of information; you just need to know how to unpack it. The `x509` command does just that, so let's use it to look at the self-signed certificates you generated.

In the following example, I use the `-text` switch to print certificate contents and `-noout` to reduce clutter by not printing the encoded certificate itself (which is the default behavior):

```
$ openssl x509 -text -in fd.crt -noout
Certificate:
    Data:
```

```
        Version: 3 (0x2)
        Serial Number:
            76:bc:fb:f6:06:0e:61:eb:99:5e:83:ea:ef:92:0b:32:4f:fd:3b:51
        Signature Algorithm: ecdsa-with-SHA256
        Issuer: C = GB, L = London, O = Feisty Duck Ltd, CN = www.feistyduck.com
        Validity
            Not Before: Aug 15 09:31:54 2020 GMT
            Not After : Aug 15 09:31:54 2021 GMT
        Subject: C = GB, L = London, O = Feisty Duck Ltd, CN = www.feistyduck.com
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:8a:d5:de:69:30:c7:77:b0:a0:54:f7:b3:34:9a:
                    96:1c:23:81:e3:9c:0c:81:a6:8a:a5:14:76:f4:4c:
                    b3:10:cb:ee:50:d1:ea:70:e9:7f:8f:75:67:f9:12:
                    83:b0:11:e7:6c:64:de:bc:af:bd:3f:43:da:b8:41:
                    96:75:34:63:85
                ASN1 OID: prime256v1
                NIST CURVE: P-256
        X509v3 extensions:
            X509v3 Subject Alternative Name:
                DNS:*.feistyduck.com, DNS:feistyduck.com
    Signature Algorithm: ecdsa-with-SHA256
         30:45:02:20:4d:36:34:cd:e9:3e:df:18:52:e7:74:c4:a1:97:
         91:6a:e7:c1:6d:12:01:63:d1:fd:90:28:32:70:24:5c:be:35:
         02:21:00:bd:02:64:c9:8b:27:8f:79:c7:a4:41:7c:31:2f:98:
         29:3e:db:8c:f3:f1:d7:bb:fa:fe:95:48:be:16:e1:ab:1b
```

Self-signed certificates usually contain only the most basic certificate data, and most of it is self-explanatory. In essence, there's the main body of the certificate, to which a signature is added. By comparison, certificates issued by public CAs are much more interesting, as they contain a number of additional fields (via the X.509 extension mechanism).

Examining Public Certificates

When you look at a public certificate, in the first part of the output you will find information that is similar to that in self-signed certificates. In fact, the only difference will be that this certificate has a different parent, as indicated by the issuer information.

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            03:5e:50:53:75:08:1a:f2:7d:27:64:4f:d5:6f:1a:02:07:89
        Signature Algorithm: sha256WithRSAEncryption
```

```
        Issuer: C = US, O = Let's Encrypt, CN = Let's Encrypt Authority X3
        Validity
            Not Before: Aug  2 23:10:45 2020 GMT
            Not After : Oct 31 23:10:45 2020 GMT
        Subject: CN = www.feistyduck.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                RSA Public-Key: (2048 bit)
                Modulus:
                    00:c8:14:4f:33:9a:db:bb:e7:e3:78:93:46:5d:56:
                    a7:bc:58:86:43:dc:ea:c1:01:52:4b:0f:20:b7:38:
                    [...]
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            [...]
```

The main differences are going to be in the X.509 extensions, which contain a great deal of very interesting information. Let's examine what's in the extensions and why it's there, in no particular order.

The *Basic Constraints* extension is used to mark certificates as belonging to a CA, giving them the ability to sign other certificates. Non-CA certificates will either have this extension omitted or will have the value of CA set to `FALSE`. This extension is critical, which means that all software-consuming certificates must understand its meaning.

```
X509v3 Basic Constraints: critical
    CA:FALSE
```

The *Key Usage* (KU) and *Extended Key Usage* (EKU) extensions restrict what a certificate can be used for. If these extensions are present, then only the listed uses are allowed. If the extensions are not present, there are no use restrictions. What you see in this example is typical for a web server certificate, which, for example, does not allow for code signing:

```
X509v3 Key Usage: critical
    Digital Signature, Key Encipherment
X509v3 Extended Key Usage:
    TLS Web Server Authentication, TLS Web Client Authentication
```

The *CRL Distribution Points* extension lists the addresses where the CA's *Certificate Revocation List* (CRL) information can be found. This information is important in cases in which certificates need to be revoked. CRLs are CA-signed lists of revoked certificates, published at regular time intervals (e.g., seven days). Let's Encrypt doesn't provide CRLs, so I took the following snippet from another certificate:

```
X509v3 CRL Distribution Points:
    Full Name:
      URI:http://crl.starfieldtech.com/sfs3-20.crl
```

> **Note**
>
> You might have noticed that the CRL location doesn't use a secure server, and you might be wondering if the link is thus insecure. It is not. Because each CRL is signed by the CA that issued it, clients are able to verify its integrity. In fact, if CRLs were distributed over TLS, browsers might face a chicken-and-egg problem in which they want to verify the revocation status of the certificate used by the server delivering the CRL itself!

The *Certificate Policies* extension is used to indicate the policy under which the certificate was issued. For example, this is where you can expect to find an indication of the type of validation used to ascertain the identity of the owner. *Extended validation* (EV) indicators can be found (as in the example that follows). These indicators are in the form of unique object identifiers (OIDs), some of which are generic and some specific to the issuing CA. In the following example, the OID 2.23.140.1.2.1 indicates a domain-validated certificate. In addition, this extension often contains one or more *Certificate Policy Statement* (CPS) points, which are usually web pages or PDF documents.

```
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 *Authority Information Access* (AIA) extension usually contains two important pieces of information. First, it lists the address of the CA's *Online Certificate Status Protocol* (OCSP) responder, which can be used to check for certificate revocation in real time. The extension may also contain a link to where the issuer's certificate (the next certificate in the chain) can be found. These days, server certificates are rarely signed directly by trusted root certificates, which means that users must include one or more intermediate certificates in their configuration. Mistakes are easy to make and will invalidate the certificates. Some clients will use the information provided in this extension to fix an incomplete certificate chain, but many clients won't.

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

The *Subject Key Identifier* and *Authority Key Identifier* extensions establish unique subject and authority key identifiers, respectively. The value specified in the Authority Key Identifier extension of a certificate must match the value specified in the Subject Key Identifier extension in the issuing certificate. This information is very useful during the certification path-building process, in which a client is trying to find all possible paths from a leaf (server) certificate to a trusted root. Certification authorities will often use one private key with more than one certificate, and this field allows software to reliably identify which certificate can be matched

to which key. In the real world, many certificate chains supplied by servers are invalid, but that fact often goes unnoticed because browsers are able to find alternative trust paths.

```
X509v3 Subject Key Identifier:
    A1:EC:11:C6:E1:E8:F7:E6:98:85:FA:9A:53:F8:B8:F1:D6:88:F9:A3
X509v3 Authority Key Identifier:
    keyid:A8:4A:6A:63:04:7D:DD:BA:E6:D1:39:B7:A6:45:65:EF:F3:A8:EC:A1
```

The *Subject Alternative Name* extension is used to list all the hostnames for which the certificate is valid. This extension used to be optional; if it isn't present, clients fall back to using the information provided in the *Common Name* (CN), which is part of the *Subject* field. If the extension is present, then the content of the CN field is ignored during validation.

```
X509v3 Subject Alternative Name:
    DNS:www.feistyduck.com, DNS:feistyduck.com
```

Finally, the most recent addition is the *Certificate Transparency* (CT) extension, which is used to carry proof of logging to various public CT logs. Depending on the certificate lifetime, you can expect to see anywhere from two to five *Signed Certificate Timestamps* (SCTs). There isn't a single set of unified requirements for the numbers and types of SCTs that are necessary to recognize a certificate as valid. Technically, it's up to every client to specify what they expect. In practice, Chrome was the first browser to require CT and other clients are likely to follow its lead.[14]

```
CT Precertificate SCTs:
    Signed Certificate Timestamp:
        Version   : v1 (0x0)
        Log ID    : 5E:A7:73:F9:DF:56:C0:E7:B5:36:48:7D:D0:49:E0:32:
                    7A:91:9A:0C:84:A1:12:12:84:18:75:96:81:71:45:58
        Timestamp : Aug  3 00:10:45.300 2020 GMT
        Extensions: none
        Signature : ecdsa-with-SHA256
                    30:45:02:21:00:BB:7F:D0:E1:E6:CD:4B:E7:79:30:AE:
                    BE:F6:50:4F:36:A4:F6:1D:65:21:1A:05:A9:B3:F0:53:
                    BA:FA:AC:6D:FB:02:20:52:23:B9:F9:B6:73:34:7F:3D:
                    7F:42:5C:E3:9D:3D:DA:D8:7F:B3:7E:21:0C:27:54:9B:
                    DA:E1:3F:0F:8E:09:60
    [...]
```

Key and Certificate Conversion

Private keys and certificates can be stored in a variety of formats, which means that you'll often need to convert them from one format to another. The most common formats are:

[14] Certificate Transparency in Chrome (Chromium; retrieved 15 August 2020)

Binary (DER) certificate
: Contains an X.509 certificate in its raw form, using DER ASN.1 encoding.

ASCII (PEM) certificate(s)
: Contains a base64-encoded DER certificate, with `-----BEGIN CERTIFICATE-----` used as the header and `-----END CERTIFICATE-----` as the footer. Usually seen with only one certificate per file, although some programs allow more than one certificate depending on the context. For example, older Apache web server versions require the server certificate to be alone in one file, with all intermediate certificates together in another.

Legacy OpenSSL key format
: Contains a private key in its raw form, using DER ASN.1 encoding. Historically, OpenSSL used a format based on PKCS #1. These days, if you use the proper commands (i.e., `genpkey`), OpenSSL defaults to PKCS#8.

ASCII (PEM) key
: Contains a base64-encoded DER key, sometimes with additional metadata (e.g., the algorithm used for password protection). The text in the header and footer can differ, depending on what underlying key format is used.

PKCS#7 certificate(s)
: A complex format designed for the transport of signed or encrypted data, defined in RFC 2315. It's usually seen with `.p7b` and `.p7c` extensions and can include the entire certificate chain as needed. This format is supported by Java's `keytool` utility.

PKCS#8 key
: The new default format for the private key store. PKCS#8 is defined in RFC 5208. Should you need to convert from PKCS#8 to the legacy format for whatever reason, use the `pkcs8` command.

PKCS#12 (PFX) key and certificate(s)
: A complex format that can store and protect a server key along with an entire certificate chain. It's commonly seen with `.p12` and `.pfx` extensions. This format is commonly used in Microsoft products, but is also used for client certificates. These days, the PFX name is used as a synonym for PKCS#12, even though PFX referred to a different format a long time ago (an early version of PKCS#12). It's unlikely that you'll encounter the old version anywhere.

PEM and DER Conversion

Certificate conversion between PEM and DER formats is performed with the `x509` tool. To convert a certificate from PEM to DER format:

```
$ openssl x509 -inform PEM -in fd.pem -outform DER -out fd.der
```

To convert a certificate from DER to PEM format:

```
$ openssl x509 -inform DER -in fd.der -outform PEM -out fd.pem
```

The syntax is identical if you need to convert private keys between DER and PEM formats, but different commands are used: `rsa` for RSA keys, and `dsa` for DSA keys. If you're dealing with the new PKCS#8 format, use the `pkey` tool.

PKCS#12 (PFX) Conversion

One command is all that's needed to convert the key and certificates in PEM format to PKCS#12. The following example converts a key (`fd.key`), certificate (`fd.crt`), and intermediate certificates (`fd-chain.crt`) into an equivalent single PKCS#12 file:

```
$ openssl pkcs12 -export \
    -name "My Certificate" \
    -out fd.p12 \
    -inkey fd.key \
    -in fd.crt \
    -certfile fd-chain.crt
Enter Export Password: ****************
Verifying - Enter Export Password: ****************
```

The reverse conversion isn't as straightforward. You can use a single command, but in that case you'll get the entire contents in a single file:

```
$ openssl pkcs12 -in fd.p12 -out fd.pem -nodes
```

Now, you must open the file `fd.pem` in your favorite editor and manually split it into individual key, certificate, and intermediate certificate files. While you're doing that, you'll notice additional content provided before each component. For example:

```
Bag Attributes
    localKeyID: E3 11 E4 F1 2C ED 11 66 41 1B B8 83 35 D2 DD 07 FC DE 28 76
subject=/1.3.6.1.4.1.311.60.2.1.3=GB/2.5.4.15=Private Organization↩
/serialNumber=06694169/C=GB/ST=London/L=London/O=Feisty Duck Ltd↩
/CN=www.feistyduck.com
issuer=/C=US/ST=Arizona/L=Scottsdale/O=Starfield Technologies, Inc./OU=http:/↩
/certificates.starfieldtech.com/repository/CN=Starfield Secure Certification ↩
Authority
-----BEGIN CERTIFICATE-----
MIIF5zCCBM+gAwIBAgIHBG9JXlv9vTANBgkqhkiG9w0BAQUFADCB3DELMAkGA1UE
BhMCVVMxEDAOBgNVBAgTB0FyaXpvbmExEzARBgNVBAcTClNjb3RQc2RhbGUxJTAj
[...]
```

This additional metadata is very handy to quickly identify the certificates. Obviously, you should ensure that the main certificate file contains the leaf server certificate and not something else. Further, you should also ensure that the intermediate certificates are provided in

the correct order, with the issuing certificate following the signed one. If you see a self-signed root certificate, feel free to delete it or store it elsewhere; it shouldn't go into the chain.

> **Warning**
>
> The final conversion output shouldn't contain anything apart from the encoded key and certificates. Although some tools are smart enough to ignore what isn't needed, other tools are not. Leaving extra data in PEM files might result in problems that are difficult to troubleshoot.

It's possible to get OpenSSL to split the components for you, but doing so requires multiple invocations of the `pkcs12` command (including typing the bundle password each time):

```
$ openssl pkcs12 -in fd.p12 -nocerts -out fd.key -nodes
$ openssl pkcs12 -in fd.p12 -nokeys -clcerts -out fd.crt
$ openssl pkcs12 -in fd.p12 -nokeys -cacerts -out fd-chain.crt
```

This approach won't save you much work. You must still examine each file to ensure that it contains the correct contents and to remove the metadata.

PKCS#7 Conversion

To convert from PEM to PKCS#7, use the `crl2pkcs7` command:

```
$ openssl crl2pkcs7 -nocrl -out fd.p7b -certfile fd.crt -certfile fd-chain.crt
```

To convert from PKCS#7 to PEM, use the `pkcs7` command with the `-print_certs` switch:

```
openssl pkcs7 -in fd.p7b -print_certs -out fd.pem
```

Similar to the conversion from PKCS#12, you must now edit the `fd.pem` file to clean it up and split it into the desired components.

Configuration

A common task in TLS server configuration is selecting which cipher suites to use. To communicate securely, TLS needs to decide which cryptographic primitives to use to achieve its goals (e.g., confidentiality). This is done by selecting a suitable cipher suite, which makes a series of decisions about how authentication, key exchange, encryption, and other operations are done. Programs that rely on OpenSSL usually adopt the same approach to suite configuration that OpenSSL uses, simply passing through the configuration options.

Before TLS 1.3, the usual server configuration would include cipher suite configuration and an option for the server to prefer the stronger suites during the negotiation. Because of some differences in the design of TLS 1.3 from earlier protocol versions, OpenSSL decided to configure

it differently, increasing the complexity of server configuration. I'll discuss this in the following sections.

Coming up with a good suite configuration can be pretty time consuming, and there are a lot of details to consider. I wrote this section to serve two goals. If you don't want to spend a lot of time learning how to use OpenSSL and how to rank cipher suites, simply use the default configuration I provide. On the other hand, if you prefer to learn the ins and outs of OpenSSL configuration, this section has the answers.

Obtaining Supported Suites

Let's start this deep dive by first determining which suites are supported by your OpenSSL installation. To do this, invoke the `ciphers` command with the `-v` switch and `ALL:COMPLEMENTOFALL` as a parameter:

```
$ openssl ciphers -v 'ALL:COMPLEMENTOFALL'
```

> **Tip**
>
> From OpenSSL 1.0.0, the `ciphers` command supports the uppercase `-V` switch to provide extra-verbose output. In this mode, the output will also contain suite IDs, which are always handy to have. For example, OpenSSL doesn't always use the RFC names for suites; in such cases, you must use the IDs to cross-check. In this section, I use the lowercase `-v` because the output is easier to show in the book.

At this point you will observe a lot of output, consisting of everything your installation of OpenSSL has to offer. In my case, there were 162 suites in the output. Let's take a look at one line:

```
TLS_AES_256_GCM_SHA384 TLSv1.3 Kx=any Au=any Enc=AESGCM(256) Mac=AEAD
```

Each line of output provides extended information on one suite. From left to right:

1. Suite name
2. Required minimum protocol version[15]
3. Key exchange algorithm
4. Authentication algorithm
5. Encryption algorithm and strength
6. MAC (integrity) algorithm

[15] Some suites on the list show `SSLv3` in the protocol column. This is nothing to worry about. It only means that the suite is compatible with this old (and obsolete) protocol version. Your configuration will not downgrade to SSL 3.0 if these suites are used.

Traditionally, OpenSSL didn't use official suite names, although it does now for TLS 1.3 suites. As of recently, when you add the `-stdname` switch to the `ciphers` tool, you'll get the official suite names and OpenSSL names at the same time.

> **Note**
>
> You may notice that all TLS 1.3 suites have any under key exchange and authentication. This is because this protocol version moved these two aspects of the handshake out of the cipher suites and into the protocol itself. It also removed all insecure algorithms, so in this context any isn't bad or insecure.

Understanding Security Levels

In the previous section, we discussed how to get a complete list of supported suites—but that list is deceptive. Just because something is supported doesn't mean it's going to be enabled. Unless you tell it otherwise, the `ciphers` command outputs even the suites that will not be allowed. The trick is to use the `-s` switch, after which the number of suites will go down from 162 to only 77.

Granted, the reduction is in large part due to the removal of PSK and SRP suites, which removes 66 entries (down to 96). The remaining difference of 21 entries is due to the concept called *security level*, which OpenSSL now uses.[16]

Cipher suite configuration is complex, and most people are not experts in cryptography. For example, it's very easy to follow some outdated advice from the Internet and add an insecure element to your configuration. Additionally, there are some aspects of security that cannot be configured via cipher suites and previously couldn't be controlled at all. Security levels therefore are designed to guarantee minimum security requirements, even if incorrect configuration is requested. They are a very useful safety net.

Table 1.1. OpenSSL security levels

Security Level	Meaning
Level 0	No restrictions. Allows all features enabled at compile time. **Insecure.**
Level 1	The security level corresponds to a minimum of 80 bits of security. **Weak.**
Level 2	Security level set to 112 bits of security.
Level 3	Security level set to 128 bits of security.
Level 4	Security level set to 192 bits of security.
Level 5	Security level set to 256 bits of security.

The most important aspect of security levels is knowing what not to use, and that's the first two levels. Level 0 imposes no restrictions and is potentially insecure (depending on what features

[16] `SSL_CTX_get_security_level man page` (OpenSSL; retrieved 21 August 2020)

were enabled at compile time). You aren't very likely to need this level in practice. Level 1 is slightly better but still allows weak elements. You may need this level for interoperability purposes with legacy systems. Level 1 is the default security level in OpenSSL.

In practice, you should aim to use level 2 as your baseline. This level supports 2048-bit RSA keys, which most web sites use today. Weak protocols such as SSL 2 and SSL 3 won't be allowed, along with RC4 and SHA1. If you're not using stock OpenSSL, you may find that your distribution already made this choice for you; for example, Ubuntu 20.04 LTS chooses level 2 as the default.

Levels 3 and up are levels you should consider if you have specific security requirements and want to enforce stronger encryption. For example, if you enable level 3, 2048-bit RSA keys won't be allowed. Because keys stronger than this are quite slow, this choice of security level implicitly restricts you to ECDSA keys.

You can gain a better understanding of security levels by using the `-s` switch along with the `@SECLEVEL` keyword as part of your suite configuration. For an example, let's see how the switch from level 3 to level 4 affects one arbitrary suite configuration. At level 3, there are four suites in the output:

```
$ openssl ciphers -v -s -tls1_2 'EECDH+AESGCM @SECLEVEL=3'
ECDHE-ECDSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH Au=ECDSA Enc=AESGCM(256) Mac=AEAD
ECDHE-RSA-AES256-GCM-SHA384   TLSv1.2 Kx=ECDH Au=RSA   Enc=AESGCM(256) Mac=AEAD
ECDHE-ECDSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH Au=ECDSA Enc=AESGCM(128) Mac=AEAD
ECDHE-RSA-AES128-GCM-SHA256   TLSv1.2 Kx=ECDH Au=RSA   Enc=AESGCM(128) Mac=AEAD
```

However, at level 4 there are only two suites in the output, because the 128-bit suites were removed:

```
$ openssl ciphers -v -s -tls1_2 'EECDH+AESGCM @SECLEVEL=4'
ECDHE-ECDSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH Au=ECDSA Enc=AESGCM(256) Mac=AEAD
ECDHE-RSA-AES256-GCM-SHA384   TLSv1.2 Kx=ECDH Au=RSA   Enc=AESGCM(256) Mac=AEAD
```

You may have noticed that the previous example introduces the `-tls1_2` switch, which outputs only suites that can be negotiated with TLS 1.2. This switch, along with `-tls1_3`, `-tls1_1`, `-tls1`, and `-ssl3`, is very useful for removing unwanted output when you're interested in only one protocol.

Configuring TLS 1.3

If you're working with the `ciphers` tool and you're not familiar with how TLS 1.3 is configured (e.g., you only worked with versions of OpenSSL that did not support this protocol), you may be confused by the fact that no matter what configuration you specify, the TLS 1.3 suites are always listed at the top. This is happening because OpenSSL introduced a separate mechanism for TLS 1.3 suite configuration. At the library level, there are separate function calls for this, and there is a separate approach to use with the command-line tools.

When it comes to the `ciphers` tool, to control TLS 1.3 suites you'll need to use the `-ciphersuites` switch. To illustrate this, let's enable one TLS 1.3 suite and one SEED suite:

```
$ openssl ciphers -v -s -ciphersuites TLS_AES_256_GCM_SHA384 SEED-SHA
TLS_AES_256_GCM_SHA384  TLSv1.3 Kx=any Au=any Enc=AESGCM(256) Mac=AEAD
SEED-SHA                SSLv3   Kx=RSA Au=RSA Enc=SEED(128)   Mac=SHA1
```

When they were adding this new configuration mechanism for TLS 1.3, OpenSSL developers took an opportunity to simplify how suites are configured by removing a variety of tools and keywords that can now be called *legacy suite configuration*. The only supported approach for TLS 1.3 is to provide a colon-separated list of the suites you wish to support, in the order you wish to support them. That's all. For example:

```
$ openssl ciphers -v -s -tls1_3 \
-ciphersuites TLS_AES_128_GCM_SHA256:TLS_AES_256_GCM_SHA384
TLS_AES_128_GCM_SHA256  TLSv1.3 Kx=any Au=any Enc=AESGCM(128) Mac=AEAD
TLS_AES_256_GCM_SHA384  TLSv1.3 Kx=any Au=any Enc=AESGCM(256) Mac=AEAD
```

> **Note**
>
> Even though there is a separate configuration string for TLS 1.3 suites, the configuration is still affected by the security level configuration, which is specified in the legacy configuration string.

How does this new approach to TLS 1.3 configuration affect real life? Depending on your tools, you may now find yourself needing to use two configuration strings where previously there was only one. In the Apache web server, the `SSLCipherSuite` directive has been extended with an optional first parameter, enabling you to target the protocols you wish to configure. So you could do something like this:

```
SSLCipherSuite TLSv1.3 TLS_AES_128_GCM_SHA256
SSLCipherSuite EECDH+AES128+AESGCM
```

The result would be equivalent to the following:

```
TLS_AES_128_GCM_SHA256         TLSv1.3 Kx=any  Au=any   Enc=AESGCM(128) Mac=AEAD
ECDHE-ECDSA-AES128-GCM-SHA256  TLSv1.2 Kx=ECDH Au=ECDSA Enc=AESGCM(128) Mac=AEAD
ECDHE-RSA-AES128-GCM-SHA256    TLSv1.2 Kx=ECDH Au=RSA   Enc=AESGCM(128) Mac=AEAD
```

Not all tools have added support for TLS 1.3 suite configuration. If you prefer the Nginx web server, for example, you'll find that there is no official way to change the TLS 1.3 defaults. Instead, you'll always get the OpenSSL defaults. For most users, this not yet a real problem because all TLS 1.3 suites are strong. But if you want to do something out of the ordinary, perhaps enable the CCM suites that are currently disabled by default, you'll have to resort to using a workaround by changing the OpenSSL defaults via a configuration file, which I will cover in the next section.

Configuring OpenSSL Defaults

Occasionally, you'll run into a problem trying to configure some applications to use OpenSSL in a certain way, only to be frustrated if there are no configuration options to achieve what you need. In that situation, you can resort to changing the OpenSSL defaults.

On startup, OpenSSL will go through an initialization procedure that attempts to fetch the defaults from the filesystem. This procedure consists of the following steps:

1. Check the `OPENSSL_CONF` environment variable, which is expected to contain a path to the configuration file. This step is skipped if the binary has the `setuid` or `setguid` flag set.
2. Failing that, check the default system-wide location of the configuration directory specified at compile time. OpenSSL will look in this folder for a file called `openssl.cnf`.

This process ensures that there are a number of options available to control the defaults in a way that solves a particular need. We can change the default configuration of only one program or of all programs that run on the same server.

For the latter use case, use the `version` tool to determine the location of the default configuration file:

```
$ openssl version -d
OPENSSLDIR: "/usr/lib/ssl"
```

Now that we know how to change the defaults, the question instead becomes what to put into the configuration file. For the syntax of configuration files and detailed information, it's best that you consult the official documentation.[17] However, if you just need to reconfigure the cipher suite configuration, take a look at the following example that does just that:

```
[default_conf]
ssl_conf = ssl_section

[ssl_section]
system_default = system_default_section

[system_default_section]
MinProtocol = TLSv1.2
CipherString = DEFAULT:@SECLEVEL=2
Ciphersuites = TLS_AES_128_GCM_SHA256:TLS_CHACHA20_POLY1305_SHA256
Options = ServerPreference,PrioritizeChaCha
```

This configuration file specifies the minimum supported protocol, security level, legacy cipher suite configuration, and TLS 1.3 suite configuration, and it also enables special ChaCha20

[17] `config` man page (OpenSSL; retrieved 21 August 2020)

prioritization, which is triggered if OpenSSL detects that client prefers this cipher over AES. For the complete list of available parameters, refer to the official documentation.[18]

Recommended Suite Configuration

When it comes to cipher suite configuration, the best approach is to avoid legacy keyword-based suite configuration and instead explicitly specify the suites you want to use. By doing this, you don't have to learn about complex keyword behavior, you'll minimize mistakes, and you'll also leave behind a configuration that is self-documenting and easy to understand.

In this section, I will give you my recommendations and explain my reasoning. For simplicity, I'll show the suites as a single ordered list, even though they are configured separately for TLS 1.3 and separately for earlier protocol versions. Here is my recommended default configuration for all TLS services, given as a list of suites in the order of preference:

```
TLS_AES_128_GCM_SHA256
TLS_CHACHA20_POLY1305_SHA256
TLS_AES_256_GCM_SHA384

ECDHE-ECDSA-AES128-GCM-SHA256
ECDHE-ECDSA-CHACHA20-POLY1305
ECDHE-ECDSA-AES256-GCM-SHA384
ECDHE-ECDSA-AES128-SHA
ECDHE-ECDSA-AES256-SHA
ECDHE-ECDSA-AES128-SHA256
ECDHE-ECDSA-AES256-SHA384
ECDHE-RSA-AES128-GCM-SHA256
ECDHE-RSA-CHACHA20-POLY1305
ECDHE-RSA-AES256-GCM-SHA384
ECDHE-RSA-AES128-SHA
ECDHE-RSA-AES256-SHA
ECDHE-RSA-AES128-SHA256
ECDHE-RSA-AES256-SHA384
DHE-RSA-AES128-GCM-SHA256
DHE-RSA-CHACHA20-POLY1305
DHE-RSA-AES256-GCM-SHA384
DHE-RSA-AES128-SHA
DHE-RSA-AES256-SHA
DHE-RSA-AES128-SHA256
DHE-RSA-AES256-SHA256
```

[18] SSL_CONF_cmd man page (OpenSSL; retrieved 21 August 2020)

This configuration uses only suites that support forward secrecy and provide strong encryption.[19] The preference is for 128-bit suites, which are faster and provide strong security; ECDSA public key encryption, which is faster and more secure than the traditional RSA (at the usual key lengths); ECDHE key exchange, which is faster than DHE; and authenticated encryption, which is faster and more secure than the old CBC mode.

It's possible to have a shorter suite configuration—for example, by removing the 256-bit suites, as well as those that use DHE for the key exchange. However, I have found that having a somewhat diverse collection of suites helps avoid various edge cases with picky clients.

In terms of interoperability, all modern browsers and clients should be able to connect. Some very old clients might not, but we're talking about obsolete platforms—for example, Internet Explorer running on Windows XP. If you really need to support this, you will need to append to the list suites that use obsolete features such as 3DES or the RSA key exchange.

Finally, when it comes to performance, there is one final trick you can employ: tell OpenSSL to use ChaCha20 with clients that prefer this cipher over AES. You will notice that in my configuration, there is always a ChaCha20 suite that follows a 128-bit AES-GCM suite. For most clients, AES-GCM is the right choice, but ChaCha20 is a better option for some mobile clients because they can do it faster.[20] With ChaCha20 prioritization, you can give those mobile clients a better experience (faster loading times).

The OpenSSL option for this is called `PrioritizeChaCha`. This feature is a relatively new configuration option, and you will find that not all server software can control it. For example, at the time of writing, Apache can (using `SSLOpenSSLConfCmd`) but Nginx can't. Resorting to changing the OpenSSL defaults, as described in the previous section, should do the trick in the latter situation.

> **Note**
>
> In practice, most systems don't need to be configured to support the best possible performance or mobile client experience. If you enjoy getting your TLS configuration just right, then by all means follow all the advice from this section. Usually, however, you shouldn't spend too much time on the fine-tuning. If you find yourself with a platform that doesn't support TLS 1.3 suite configuration or that isn't able to prioritize ChaCha20, just use the defaults and move on.

[19] In TLS 1.3, key exchange and authentication are not controlled by cipher suites; negotiation of these aspects has been moved into the protocol. However, because robust forward security is a key feature in TLS 1.3, it's not something we need to worry about when it comes to cipher suite configuration.

[20] Speeding up and Strengthening HTTPS Connections for Chrome on Android (Google Security Blog; 24 April 2014)

Generating DH Parameters

The DH key exchange has fallen out of fashion, but you may still want to support it in your servers on philosophical grounds. If you do, you may find with some server software (e.g., Nginx) that you need to manually configure the desired DH parameters. This is how:

```
$ openssl dhparam -out dh-2048.pem 2048
```

In practice, only 2048-bit DH parameters make sense. Anything less is going to be weak or insecure, while anything more is going to slow you down. DH parameters need not be secret. In fact, there are some predefined groups (sometimes called *well-known groups*) that are recommended because they are known to have been securely generated.[21]

Rarely, you may encounter a situation, usually in a legacy environment, in which you need to configure a server with 1024-bit DH parameters. It's essential that you don't use a well-known group in this case. The issue is that weak DH groups are susceptible to precomputation attacks, which further downgrade their security. If you really must use a 1024-bit DH parameters, always generate your own unique group using OpenSSL.

Legacy Suite Configuration

In this section, I'll briefly cover the legacy keyword-based configuration of cipher suites that applies to TLS 1.2 and earlier protocol versions. This section is important largely only if you're interested in how the keyword approach works. Otherwise, you're better off simply specifying the suites you wish to use, as I did with the recommended configuration in the previous section.

Keywords

Cipher suite *keywords* are the basic building blocks of cipher suite configuration. Each suite name (e.g., `RC4-SHA`) is a keyword that selects exactly one suite.[22] All other keywords select groups of suites according to some criteria. Keyword names are case-sensitive. In this section, I will provide an overview of all cipher suite keywords supported by OpenSSL, one group at a time.

Group keywords are shortcuts that select frequently used cipher suites. For example, `HIGH` will select only very strong cipher suites.

[21] RFC 7919: Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for TLS (D. Gillmor, August 2016)

[22] With recent OpenSSL releases, you can use the legacy suite names that are specific to OpenSSL, but also the standard suite names.

Table 1.2. Group keywords

Keyword	Meaning
DEFAULT	The default cipher list. This is determined at compile time and must be the first cipher string specified.
COMPLEMENTOFDEFAULT	The ciphers included in ALL, but not enabled by default. Note that this rule does not cover eNULL, which is not included by ALL (use COMPLEMENTOFALL if necessary).
ALL	All cipher suites except the eNULL ciphers, which must be explicitly enabled.
COMPLEMENTOFALL	The cipher suites not enabled by ALL, currently eNULL.
HIGH	"High"-encryption cipher suites. This currently means those with key lengths larger than 128 bits, and some cipher suites with 128-bit keys.
MEDIUM	"Medium"-encryption cipher suites, currently some of those using 128-bit encryption.
LOW	"Low"-encryption cipher suites, currently those using 64- or 56-bit encryption algorithms, but excluding export cipher suites. No longer supported. **Insecure.**
EXP, EXPORT	Export encryption algorithms. Including 40- and 56-bit algorithms. No longer supported. **Insecure.**
EXPORT40	40-bit export encryption algorithms. No longer supported. **Insecure.**
EXPORT56	56-bit export encryption algorithms. No longer supported. **Insecure.**
TLSv1.2, TLSv1.0, TLSv1, SSLv3, SSLv2	Cipher suites that require the specified protocol version. There are two keywords for TLS 1.0 and no keywords for TLS 1.3 and TLS 1.1. These keywords do not affect protocol configuration, just the suites.

Digest keywords select suites that use a particular digest algorithm. For example, SHA256 selects all suites that rely on SHA256 for integrity validation.

Table 1.3. Digest algorithm keywords

Keyword	Meaning
MD5	Cipher suites using MD5. **Obsolete and insecure.**
SHA, SHA1	Cipher suites using SHA1.
SHA256	Cipher suites using SHA256.
SHA384	Cipher suites using SHA384.

> **Note**
>
> The digest algorithm keywords select only suites that validate data integrity at the protocol level. TLS 1.2 introduced support for authenticated encryption, which is a mechanism that bundles encryption with integrity validation. When the so-called AEAD (*Authenticated Encryption with Associated Data*) suites are used, the protocol doesn't need to provide additional integrity verification. For this reason, you won't be able to use the digest algorithm keywords to select AEAD suites (currently, those that have GCM in the name). The names of these suites do use SHA256 and SHA384

suffixes, but (confusing as it may be) here they refer to the hash functions used to build the *pseudorandom function* used with the suite.

Authentication keywords select suites based on the authentication method they use. Today, the RSA public key algorithm is still used by the majority of certificates, with ECDSA quickly catching up.

Table 1.4. Authentication keywords

Keyword	Meaning
aDH	Cipher suites effectively using DH authentication, i.e., the certificates carry DH keys. Removed in 1.1.0.
aDSS, DSS	Cipher suites using DSS authentication, i.e., the certificates carry DSS keys.
aECDH	Cipher suites that use ECDH authentication. Removed in 1.1.0.
aECDSA, ECDSA	Cipher suites that use ECDSA authentication.
aNULL	Cipher suites offering no authentication. This is currently the anonymous DH algorithms. **Insecure.**
aRSA	Cipher suites using RSA authentication, i.e., the certificates carry RSA keys.
aPSK	Cipher suites using PSK (Pre-Shared Key) authentication.
aSRP	Cipher suites using SRP (Secure Remote Password) authentication.

Key exchange keywords select suites based on the key exchange algorithm. When it comes to ephemeral Diffie-Hellman suites, OpenSSL is inconsistent in naming the suites and the keywords. In the suite names, ephemeral suites tend to have an E at the end of the key exchange algorithm (e.g., ECDHE-RSA-RC4-SHA and DHE-RSA-AES256-SHA), but in the keywords the E is at the beginning (e.g., EECDH and EDH). The preferred names today are DHE and ECDHE; the other keywords are supported for backward compatibility.

Table 1.5. Key exchange keywords

Keyword	Meaning
ADH	Anonymous DH cipher suites. **Insecure.**
AECDH	Anonymous ECDH cipher suites. **Insecure.**
DHE, EDH	Cipher suites using ephemeral DH key agreement only.
ECDHE, EECDH	Cipher suites using ephemeral ECDH.
kDHE, kEDH, DH	Cipher suites using ephemeral DH key agreement (includes anonymous DH).
kECDHE, kEECDH, ECDH	Cipher suites using ephemeral ECDH key agreement (includes anonymous ECDH).
kRSA, RSA	Cipher suites using RSA key exchange.
kPSK, kECDHEPSK, kDHEPSK, kRSAPSK	Cipher suites using PSK key exchange.

Cipher keywords select suites based on the cipher they use.

Table 1.6. Cipher keywords

Keyword	Meaning
AES, AESCCM, AESCCM8, AESGCM	Cipher suites using AES, AES CCM, and AES GCM.
ARIA, ARIA128, ARIA256	Cipher suites using ARIA.
CAMELLIA, CAMELLIA128, CAMELLIA256	Cipher suites using Camellia. **Obsolete.**
CHACHA20	Cipher suites using ChaCha20.
eNULL, NULL	Cipher suites that don't use encryption. **Insecure.**
IDEA	Cipher suites using IDEA. **Obsolete.**
SEED	Cipher suites using SEED. **Obsolete.**
3DES, DES, IDEA, RC2, RC4	No longer supported by default. **Obsolete and insecure.**

What remains is a number of suites that do not fit into any other category. The bulk of them are related to the GOST standards, which are relevant for the countries that are part of the Commonwealth of Independent States, formed after the breakup of the Soviet Union. The GOST suites are defined but require the GOST engine to be activated. The GOST engine is not part of the core OpenSSL since version 1.1.0.

Table 1.7. Miscellaneous keywords

Keyword	Meaning
@SECLEVEL	Configures the security level, which sets minimum security requirements.
@STRENGTH	Sorts the current cipher suite list in order of encryption algorithm key length.
aGOST	Cipher suites using GOST R 34.10 (either 2001 or 94) for authentication. Requires a GOST-capable engine.
aGOST01	Cipher suites using GOST R 34.10-2001 authentication.
aGOST94	Cipher suites using GOST R 34.10-94 authentication. **Obsolete.** Use GOST R 34.10-2001 instead.
kGOST	Cipher suites using VKO 34.10 key exchange, specified in RFC 4357.
GOST94	Cipher suites using HMAC based on GOST R 34.11-94.
GOST89MAC	Cipher suites using GOST 28147-89 MAC instead of HMAC.
PSK	Cipher suites using PSK in any capacity.

Combining Keywords

In most cases, you'll use keywords by themselves, but it's also possible to combine them to select only suites that meet several requirements, by connecting two or more keywords with the + character. In the following example, we select suites that use the ECDHE key exchange in combination with AES-GCM:

```
$ openssl ciphers -v -s -tls1_2 'EECDH+AESGCM'
ECDHE-ECDSA-AES256-GCM-SHA384  TLSv1.2 Kx=ECDH Au=ECDSA Enc=AESGCM(256) Mac=AEAD
ECDHE-RSA-AES256-GCM-SHA384    TLSv1.2 Kx=ECDH Au=RSA   Enc=AESGCM(256) Mac=AEAD
ECDHE-ECDSA-AES128-GCM-SHA256  TLSv1.2 Kx=ECDH Au=ECDSA Enc=AESGCM(128) Mac=AEAD
ECDHE-RSA-AES128-GCM-SHA256    TLSv1.2 Kx=ECDH Au=RSA   Enc=AESGCM(128) Mac=AEAD
```

Building Cipher Suite Lists

The key concept in building a cipher suite configuration is that of the *current suite list*. The list always starts empty, without any suites, but every keyword that you add to the configuration string will change the list in some way. By default, new suites are appended to the list. In the following example, the configuration starts with all suites that use the ECDHE key exchange, followed by all suites that use the DHE key exchange:

```
$ openssl ciphers -v 'ECDHE:DHE'
```

The colon character is commonly used to separate keywords, but spaces and commas are equally acceptable. The following command produces the same output as the previous example:

```
$ openssl ciphers -v 'ECDHE DHE'
```

Keyword Modifiers

Keyword modifiers are characters you can place at the beginning of each keyword in order to change the default action (adding to the list) to something else. The following actions are supported:

Append
: Add suites to the end of the list. If any of the suites are already on the list, they will remain in their present position. This is the default action, which is invoked when there is no modifier in front of the keyword.

Delete (`-`)
: Remove all matching suites from the list, potentially allowing some other keyword to reintroduce them later.

Permanently delete (`!`)
: Remove all matching suites from the list and prevent them from being added later by another keyword. This modifier is useful for specifying all the suites you never want to use, making further selection easier and preventing mistakes.

Move to the end (`+`)
: Move all matching suites to the end of the list. This works only on existing suites; it never adds new suites to the list. This modifier is useful if you want to keep some weaker suites enabled but prefer the stronger ones. For example, the string `AES:+AES256` enables all AES suites but pushes the 256-bit ones to the end.

Sorting

The `@STRENGTH` keyword serves a special purpose: it will not introduce or remove any suites, but it will sort them in order of descending cipher strength. Automatic sorting is an interesting idea, but it makes sense only in a perfect world in which cipher suites can actually be compared by cipher strength alone. In most cases, the highest-strength suites are not typically required. You often have them in your configuration only to interoperate with picky clients.

Handling Errors

There are two types of errors you might experience while working on your configuration. The first is a result of a typo or an attempt to use a keyword that does not exist:

```
$ openssl ciphers -v '@HIGH'
Error in cipher list
140460843755168:error:140E6118:SSL routines:SSL_CIPHER_PROCESS_RULESTR:invalid ↵
command:ssl_ciph.c:1317:
```

The output is cryptic, but it does contain an error message.

Another possibility is that you end up with an empty list of cipher suites, in which case you might see something similar to the following:

```
$ openssl ciphers -v 'SHA512'
Error in cipher list
140202299557536:error:1410D0B9:SSL routines:SSL_CTX_set_cipher_list:no cipher ↵
match:ssl_lib.c:1312:
```

Performance

As you're probably aware, computation speed is a significant limiting factor for any cryptographic operation. OpenSSL comes with a built-in benchmarking tool that you can use to get an idea about a system's capabilities and limits. You can invoke the benchmark using the `speed` command.

If you invoke `speed` without any parameters, OpenSSL produces a lot of output, little of which will be of interest. A better approach is to test only those algorithms that are directly relevant to you. For example, for usage in a secure web server, you might care about the performance of RSA and ECDSA and will do something like this:

```
$ openssl speed rsa ecdsa
```

The first part of the resulting output consists of the OpenSSL version number and compile-time configuration. This information is useful for record-keeping and if you're testing different versions of OpenSSL:

```
OpenSSL 1.1.1f  31 Mar 2020
built on: Mon Apr 20 11:53:50 2020 UTC
options:bn(64,64) rc4(16x,int) des(int) aes(partial) blowfish(ptr)
compiler: gcc -fPIC -pthread -m64 -Wa,--noexecstack -Wall -Wa,--noexecstack -g -O2 ↵
-fdebug-prefix-map=/build/openssl-P_ODHM/openssl-1.1.1f=. -fstack-protector-strong ↵
-Wformat -Werror=format-security -DOPENSSL_TLS_SECURITY_LEVEL=2 -DOPENSSL_USE↵
_NODELETE -DL_ENDIAN -DOPENSSL_PIC -DOPENSSL_CPUID_OBJ -DOPENSSL_IA32_SSE2 ↵
-DOPENSSL_BN_ASM_MONT -DOPENSSL_BN_ASM_MONT5 -DOPENSSL_BN_ASM_GF2m -DSHA1_ASM ↵
-DSHA256_ASM -DSHA512_ASM -DKECCAK1600_ASM -DRC4_ASM -DMD5_ASM -DAESNI_ASM -DVPAES↵
_ASM -DGHASH_ASM -DECP_NISTZ256_ASM -DX25519_ASM -DPOLY1305_ASM -DNDEBUG ↵
-Wdate-time -D_FORTIFY_SOURCE=2
```

The rest of the output contains the benchmark results. Let's first take a look at the RSA key operations:

```
                  sign       verify     sign/s   verify/s
rsa   512 bits  0.000073s  0.000005s  13736.4  187091.4
rsa  1024 bits  0.000207s  0.000014s   4828.4   71797.6
rsa  2048 bits  0.000991s  0.000045s   1009.1   22220.4
rsa  3072 bits  0.004796s  0.000096s    208.5   10463.5
rsa  4096 bits  0.011073s  0.000165s     90.3    6054.5
rsa  7680 bits  0.090541s  0.000565s     11.0    1769.7
rsa 15360 bits  0.521500s  0.002204s      1.9     453.7
```

RSA is most commonly used at 2048 bits. In my results, one CPU of the tested server can perform about 1,000 sign (server) operations and 22,000 verify (client) operations every second. As for ECDSA, it's typically only used at 256 bits. We can see that at this length, ECDSA can do 10 times as many signatures. On the other hand, it's slower when it comes to the verifications, at barely 6,500 operations per second:

```
                              sign      verify   sign/s   verify/s
 256 bits ecdsa (nistp256)   0.0000s   0.0002s  20508.1  6566.2
 384 bits ecdsa (nistp384)   0.0017s   0.0013s    580.4   755.0
 521 bits ecdsa (nistp521)   0.0006s   0.0012s   1711.5   840.8
```

In practice, you care more about the sign operations because servers are designed to provide services to a great many clients. The clients, on the other hand, are typically communicating with only a small number of servers at the same time. The fact that ECDSA is slower in this scenario doesn't matter much.

What's this output of `speed` useful for? You should be able to compare how compile-time options affect speed or how different versions of OpenSSL compare on the same platform. If you're thinking of switching servers, benchmarking OpenSSL can give you an idea of the differences in computing power. You can also verify that the hardware acceleration is in place.

Using the benchmark results to estimate deployment performance is not straightforward because of the great number of factors that influence performance in real life. Further, many of

those factors lie outside TLS (e.g., HTTP keep alive settings, caching, etc.). At best, you can use these numbers only for a rough estimate.

But before you can do that, you need to consider something else. By default, the `speed` command will use only a single process. Most servers have multiple cores, so to find out how many TLS operations are supported by the entire server, you must instruct `speed` to use several instances in parallel. You can achieve this with the `-multi` switch. My server has two cores, so that's what I'm going to specify:

```
$ openssl speed -multi 2 rsa
[...]
                  sign       verify     sign/s   verify/s
rsa   512 bits  0.000037s  0.000003s  27196.5  367409.6
rsa  1024 bits  0.000106s  0.000007s   9467.8  144188.0
rsa  2048 bits  0.000503s  0.000023s   1988.1   43838.4
rsa  3072 bits  0.002415s  0.000050s    414.1   20152.2
rsa  4096 bits  0.005589s  0.000084s    178.9   11880.8
rsa  7680 bits  0.045659s  0.000285s     21.9    3506.1
rsa 15360 bits  0.264904s  0.001130s      3.8     884.8
```

As expected, the performance is about two times better. I'm again looking at how many RSA signatures can be completed per second, because this is the most CPU-intensive cryptographic operation performed on a server and is thus always the first bottleneck. The result of 1,988 signatures/second (with a 2048-bit key) tells us that this small server will most definitely handle hundreds of brand-new TLS connections per second. (We have to assume that the server will do other things, not only TLS handshakes.) In my case, that's sufficient—with a very healthy safety margin. Because I also have session resumption enabled on the server—and that bypasses public encryption—I know that the performance will be even better.

When testing speed, it's important to always enable hardware acceleration using the `-evp` switch. If you don't, the results can be vastly different. As an illustration, take a look at the performance differences on a server that supports AES-NI hardware acceleration. I got the following with a software-only implementation:

```
$ openssl speed aes-128-cbc
[...]
The 'numbers' are in 1000s of bytes per second processed.
type            16 bytes     64 bytes     256 bytes    1024 bytes   8192 bytes
aes-128 cbc     131377.50k   135401.41k   134796.12k   133931.35k   134778.95k
```

The performance is more than three times better with hardware acceleration:

```
$ openssl speed -evp aes-128-cbc
[...]
The 'numbers' are in 1000s of bytes per second processed.
type            16 bytes     64 bytes     256 bytes    1024 bytes   8192 bytes
aes-128-cbc     421949.23k   451223.42k   460066.13k   463651.84k   462883.50k
```

When you're looking at the speed of cryptographic operations, you should focus on the primitives you will actually deploy. For example, CBC is obsolete, so you want to use AES in GCM mode instead. And here we see how the GCM performance is three to four times better:

```
$ openssl speed -evp aes-128-gcm
[...]
The 'numbers' are in 1000s of bytes per second processed.
type            16 bytes     64 bytes    256 bytes    1024 bytes   8192 bytes
aes-128-gcm     219599.85k   588822.40k  1313242.97k  1680529.75k  1989388.97k
```

Then there is ChaCha20-Poly1305, which is a relatively recent addition. Its performance can't compete with hardware-accelerated AES, but it doesn't need to; this authenticated cipher is designed to be fast on mobile phones. Compare its speed to nonaccelerated AES-128-CBC instead.

```
$ openssl speed -evp chacha20-poly1305
[...]
The 'numbers' are in 1000s of bytes per second processed.
type                16 bytes     64 bytes     256 bytes   1024 bytes   8192 bytes
chacha20-poly1305   148729.65k   273026.35k   590953.90k  1027021.82k  1092427.78k
```

Creating a Private Certification Authority

If you want to set up your own CA, everything you need is already included in OpenSSL. The user interface is purely command line–based and thus not very user friendly, but that's possibly for the better. Going through the process is very educational, because it forces you to think about every aspect, even the smallest details.

The educational aspect of setting a private CA is the main reason why I would recommend doing it, but there are others. An OpenSSL-based CA, crude as it might be, can well serve the needs of an individual or a small group. For example, it's much better to use a private CA in a development environment than to use self-signed certificates everywhere. Similarly, client certificates—which provide two-factor authentication—can significantly increase the security of your sensitive web applications.

The biggest challenge in running a private CA is not setting everything up but keeping the infrastructure secure. For example, the root key must be kept offline because all security depends on it. On the other hand, CRLs and OCSP responder certificates must be refreshed on a regular basis, which requires bringing the root online.

As you go through this section you will create two configuration files: one to control the root CA (`root-ca.conf`) and another to control the subordinate CA (`sub-ca.conf`). Although you should be able to do everything from scratch just by following my instructions, you can also download the configuration file templates from my GitHub account.[23] The latter option will

[23] OpenSSL CA configuration templates (Bulletproof SSL and TLS GitHub repository, retrieved 31 March 2017)

save you some time, but the former approach will give you a better understanding of the work involved.

Features and Limitations

In the rest of this section, we're going to create a private CA that's similar in structure to public CAs. There's going to be one root CA from which other subordinate CAs can be created. We'll provide revocation information via CRLs and OCSP responders. To keep the root CA offline, OCSP responders are going to have their own identities. This isn't the simplest private CA you could have, but it's one that can be secured properly. As a bonus, the subordinate CA will be *technically constrained*, which means that it will be allowed to issue certificates only for the allowed hostnames.

After the setup is complete, the root certificate will have to be securely distributed to all intended clients. Once the root is in place, you can begin issuing client and server certificates. The main limitation of this setup is that the OCSP responder is chiefly designed for testing and can be used only for lighter loads.

Creating a Root CA

Creating a new CA involves several steps: configuration, creation of a directory structure and initialization of the key files, and finally generation of the root key and certificate. This section describes the process as well as the common CA operations.

Root CA Configuration

Before we can actually create a CA, we need to prepare a configuration file (`root-ca.conf`) that will tell OpenSSL exactly how we want things set up. Configuration files aren't needed most of the time, during normal usage, but they are essential when it comes to complex operations, such as root CA creation. OpenSSL configuration files are powerful; before you proceed I suggest that you familiarize yourself with their capabilities (`man config` on the command line).

The first part of the configuration file contains some basic CA information, such as the name and the base URL, and the components of the CA's distinguished name. Because the syntax is flexible, information needs to be provided only once:

```
[default]
name                    = root-ca
domain_suffix           = example.com
aia_url                 = http://$name.$domain_suffix/$name.crt
crl_url                 = http://$name.$domain_suffix/$name.crl
ocsp_url                = http://ocsp.$name.$domain_suffix:9080
default_ca              = ca_default
name_opt                = utf8,esc_ctrl,multiline,lname,align
```

```
[ca_dn]
countryName             = "GB"
organizationName        = "Example"
commonName              = "Root CA"
```

The second part directly controls the CA's operation. For full information on each setting, consult the documentation for the `ca` command (`man ca` on the command line). Most of the settings are self-explanatory; we mostly tell OpenSSL where we want to keep our files. Because this root CA is going to be used only for the issuance of subordinate CAs, I chose to have the certificates valid for 10 years. For the signature algorithm, the secure SHA256 is used by default.

The default policy (`policy_c_o_match`) is configured so that all certificates issued from this CA have the `countryName` and `organizationName` fields that match that of the CA. This wouldn't be normally done by a public CA, but it's appropriate for a private CA:

```
[ca_default]
home                    = .
database                = $home/db/index
serial                  = $home/db/serial
crlnumber               = $home/db/crlnumber
certificate             = $home/$name.crt
private_key             = $home/private/$name.key
RANDFILE                = $home/private/random
new_certs_dir           = $home/certs
unique_subject          = no
copy_extensions         = none
default_days            = 3650
default_crl_days        = 365
default_md              = sha256
policy                  = policy_c_o_match

[policy_c_o_match]
countryName             = match
stateOrProvinceName     = optional
organizationName        = match
organizationalUnitName  = optional
commonName              = supplied
emailAddress            = optional
```

The third part contains the configuration for the `req` command, which is going to be used only once, during the creation of the self-signed root certificate. The most important parts are in the extensions: the `basicConstraints` extension indicates that the certificate is a CA, and `keyUsage` contains the appropriate settings for this scenario:

```
[req]
```

```
default_bits            = 4096
encrypt_key             = yes
default_md              = sha256
utf8                    = yes
string_mask             = utf8only
prompt                  = no
distinguished_name      = ca_dn
req_extensions          = ca_ext

[ca_ext]
basicConstraints        = critical,CA:true
keyUsage                = critical,keyCertSign,cRLSign
subjectKeyIdentifier    = hash
```

The fourth part of the configuration file contains information that will be used during the construction of certificates issued by the root CA. All certificates will be CAs, as indicated by the `basicConstraints` extension, but we set `pathlen` to zero, which means that further subordinate CAs are not allowed.

All subordinate CAs are going to be constrained, which means that the certificates they issue will be valid only for a subset of domain names and restricted uses. First, the `extendedKeyUsage` extension specifies only `clientAuth` and `serverAuth`, which is TLS client and server authentication. Second, the `nameConstraints` extension limits the allowed hostnames only to *example.com* and *example.org* domain names. In theory, this setup enables you to give control over the subordinate CAs to someone else but still be safe in knowing that they can't issue certificates for arbitrary hostnames. If you wanted, you could restrict each subordinate CA to a small domain namespace. The requirement to exclude the two IP address ranges comes from the CA/Browser Forum's Baseline Requirements, which have a definition for technically constrained subordinate CAs.[24]

In practice, name constraints are not entirely practical, because some major platforms don't currently recognize the `nameConstraints` extension. If you mark this extension as critical, such platforms will reject your certificates. You won't have such problems if you don't mark it as critical (as in the example), but then some other platforms won't enforce it.

```
[sub_ca_ext]
authorityInfoAccess     = @issuer_info
authorityKeyIdentifier  = keyid:always
basicConstraints        = critical,CA:true,pathlen:0
crlDistributionPoints   = @crl_info
extendedKeyUsage        = clientAuth,serverAuth
keyUsage                = critical,keyCertSign,cRLSign
nameConstraints         = @name_constraints
subjectKeyIdentifier    = hash
```

[24] Baseline Requirements (The CA/Browser Forum, retrieved 9 July 2014)

```
[crl_info]
URI.0                   = $crl_url

[issuer_info]
caIssuers;URI.0         = $aia_url
OCSP;URI.0              = $ocsp_url

[name_constraints]
permitted;DNS.0=example.com
permitted;DNS.1=example.org
excluded;IP.0=0.0.0.0/0.0.0.0
excluded;IP.1=0:0:0:0:0:0:0:0/0:0:0:0:0:0:0:0
```

The fifth and final part of the configuration specifies the extensions to be used with the certificate for OCSP response signing. In order to be able to run an OCSP responder, we generate a special certificate and delegate the OCSP signing capability to it. This certificate is not a CA, which you can see from the extensions:

```
[ocsp_ext]
authorityKeyIdentifier  = keyid:always
basicConstraints        = critical,CA:false
extendedKeyUsage        = OCSPSigning
keyUsage                = critical,digitalSignature
subjectKeyIdentifier    = hash
```

Root CA Directory Structure

The next step is to create the directory structure specified in the previous section and initialize some of the files that will be used during the CA operation:

```
$ mkdir root-ca
$ cd root-ca
$ mkdir certs db private
$ chmod 700 private
$ touch db/index
$ openssl rand -hex 16  > db/serial
$ echo 1001 > db/crlnumber
```

The following subdirectories are used:

certs/
: Certificate storage; new certificates will be placed here as they are issued.

db/
: This directory is used for the certificate database (index) and the files that hold the next certificate and CRL serial numbers. OpenSSL will create some additional files as needed.

private/
: This directory will store the private keys, one for the CA and the other for the OCSP responder. It's important that no other user has access to it. (In fact, if you're going to be serious about the CA, the machine on which the root material is stored should have only a minimal number of user accounts.)

> **Note**
>
> When creating a new CA certificate, it's important to initialize the certificate serial numbers with a random number generator, as I do in this section. This is very useful if you ever end up creating and deploying multiple CA certificates with the same distinguished name (common if you make a mistake and need to start over); conflicts will be avoided, because the certificates will have different serial numbers.

Root CA Generation

We take two steps to create the root CA. First, we generate the key and the CSR. All the necessary information will be picked up from the configuration file when we use the `-config` switch:

```
$ openssl req -new \
    -config root-ca.conf \
    -out root-ca.csr \
    -keyout private/root-ca.key
```

In the second step, we create a self-signed certificate. The `-extensions` switch points to the `ca_ext` section in the configuration file, which activates the extensions that are appropriate for a root CA:

```
$ openssl ca -selfsign \
    -config root-ca.conf \
    -in root-ca.csr \
    -out root-ca.crt \
    -extensions ca_ext
```

Structure of the Database File

The database in `db/index` is a plaintext file that contains certificate information, one certificate per line. Immediately after the root CA creation, it should contain only one line:

```
V    240706115345Z        1001    unknown    /C=GB/O=Example/CN=Root CA
```

Each line contains six values separated by tabs:

1. Status flag (`V` for valid, `R` for revoked, `E` for expired)
2. Expiration date (in `YYMMDDHHMMSSZ` format)
3. Revocation date or empty if not revoked

4. Serial number (hexadecimal)
5. File location or `unknown` if not known
6. Distinguished name

Root CA Operations

To generate a CRL from the new CA, use the `-gencrl` switch of the `ca` command:

```
$ openssl ca -gencrl \
    -config root-ca.conf \
    -out root-ca.crl
```

To issue a certificate, invoke the `ca` command with the desired parameters. It's important that the `-extensions` switch points to the correct section in the configuration file (e.g., you don't want to create another root CA).

```
$ openssl ca \
    -config root-ca.conf \
    -in sub-ca.csr \
    -out sub-ca.crt \
    -extensions sub_ca_ext
```

To revoke a certificate, use the `-revoke` switch of the `ca` command; you'll need to have a copy of the certificate you wish to revoke. Because all certificates are stored in the `certs/` directory, you only need to know the serial number. If you have a distinguished name, you can look for the serial number in the database.

Choose the correct reason for the value in the `-crl_reason` switch. The value can be one of the following: `unspecified`, `keyCompromise`, `CACompromise`, `affiliationChanged`, `superseded`, `cessationOfOperation`, `certificateHold`, and `removeFromCRL`.

```
$ openssl ca \
    -config root-ca.conf \
    -revoke certs/1002.pem \
    -crl_reason keyCompromise
```

Create a Certificate for OCSP Signing

First, we create a key and CSR for the OCSP responder. These two operations are done as for any non-CA certificate, which is why we don't specify a configuration file:

```
$ openssl req -new \
    -newkey rsa:2048 \
    -subj "/C=GB/O=Example/CN=OCSP Root Responder" \
    -keyout private/root-ocsp.key \
    -out root-ocsp.csr
```

Second, use the root CA to issue a certificate. The value of the `-extensions` switch specifies `ocsp_ext`, which ensures that extensions appropriate for OCSP signing are set. I reduced the lifetime of the new certificate to 365 days (from the default of 3,650). Because these OCSP certificates don't contain revocation information, they can't be revoked. For that reason, you want to keep the lifetime as short as possible. A good choice is 30 days, provided you are prepared to generate a fresh certificate that often:

```
$ openssl ca \
    -config root-ca.conf \
    -in root-ocsp.csr \
    -out root-ocsp.crt \
    -extensions ocsp_ext \
    -days 30
```

Now you have everything ready to start the OCSP responder. For testing, you can do it from the same machine on which the root CA resides. However, for production you must move the OCSP responder key and certificate elsewhere:

```
$ openssl ocsp \
    -port 9080
    -index db/index \
    -rsigner root-ocsp.crt \
    -rkey private/root-ocsp.key \
    -CA root-ca.crt \
    -text
```

You can test the operation of the OCSP responder using the following command line:

```
$ openssl ocsp \
    -issuer root-ca.crt \
    -CAfile root-ca.crt \
    -cert root-ocsp.crt \
    -url http://127.0.0.1:9080
```

In the output, `verify OK` means that the signatures were correctly verified, and `good` means that the certificate hasn't been revoked.

```
Response verify OK
root-ocsp.crt: good
        This Update: Jul  9 18:45:34 2014 GMT
```

Creating a Subordinate CA

The process of subordinate CA generation largely mirrors the root CA process. In this section, I will only highlight the differences where appropriate. For everything else, refer to the previous section.

Subordinate CA Configuration

To generate a configuration file (`sub-ca.conf`) for the subordinate CA, start with the file we used for the root CA and make the changes listed in this section. We'll change the name to `sub-ca` and use a different distinguished name. We'll put the OCSP responder on a different port, but only because the `ocsp` command doesn't understand virtual hosts. If you used a proper web server for the OCSP responder, you could avoid using special ports altogether. The default lifetime of new certificates will be 365 days, and we'll generate a fresh CRL once every 30 days.

The change of `copy_extensions` to `copy` means that extensions from the CSR will be copied into the certificate, but only if they are not already set in our configuration. With this change, whoever is preparing the CSR can put the required alternative names in it, and the information from there will be picked up and placed in the certificate. This feature is somewhat dangerous (you're allowing someone else to have limited direct control over what goes into a certificate), but I think it's fine for smaller environments:

```
[default]
name                    = sub-ca
ocsp_url                = http://ocsp.$name.$domain_suffix:9081

[ca_dn]
countryName             = "GB"
organizationName        = "Example"
commonName              = "Sub CA"

[ca_default]
default_days            = 365
default_crl_days        = 30
copy_extensions         = copy
```

At the end of the configuration file, we'll add two new profiles, one each for client and server certificates. The only difference is in the `keyUsage` and `extendedKeyUsage` extensions. Note that we specify the `basicConstraints` extension but set it to `false`. We're doing this because we're copying extensions from the CSR. If we left this extension out, we might end up using one specified in the CSR:

```
[server_ext]
authorityInfoAccess     = @issuer_info
authorityKeyIdentifier  = keyid:always
basicConstraints        = critical,CA:false
crlDistributionPoints   = @crl_info
extendedKeyUsage        = clientAuth,serverAuth
keyUsage                = critical,digitalSignature,keyEncipherment
subjectKeyIdentifier    = hash

[client_ext]
```

```
authorityInfoAccess     = @issuer_info
authorityKeyIdentifier  = keyid:always
basicConstraints        = critical,CA:false
crlDistributionPoints   = @crl_info
extendedKeyUsage        = clientAuth
keyUsage                = critical,digitalSignature
subjectKeyIdentifier    = hash
```

After you're happy with the configuration file, create a directory structure following the same process as for the root CA. Just use a different directory name, for example, `sub-ca`.

Subordinate CA Generation

As before, we take two steps to create the subordinate CA. First, we generate the key and the CSR. All the necessary information will be picked up from the configuration file when we use the `-config` switch.

```
$ openssl req -new \
    -config sub-ca.conf \
    -out sub-ca.csr \
    -keyout private/sub-ca.key
```

In the second step, we get the root CA to issue a certificate. The `-extensions` switch points to the `sub_ca_ext` section in the configuration file, which activates the extensions that are appropriate for the subordinate CA.

```
$ openssl ca \
    -config root-ca.conf \
    -in sub-ca.csr \
    -out sub-ca.crt \
    -extensions sub_ca_ext
```

Subordinate CA Operations

To issue a server certificate, process a CSR while specifying `server_ext` in the `-extensions` switch:

```
$ openssl ca \
    -config sub-ca.conf \
    -in server.csr \
    -out server.crt \
    -extensions server_ext
```

To issue a client certificate, process a CSR while specifying `client_ext` in the `-extensions` switch:

```
$ openssl ca \
```

```
-config sub-ca.conf \
-in client.csr \
-out client.crt \
-extensions client_ext
```

> **Note**
>
> When a new certificate is requested, all its information will be presented to you for verification before the operation is completed. You should always ensure that everything is in order, but especially if you're working with a CSR that someone else prepared. Pay special attention to the certificate distinguished name and the `basicConstraints` and `subjectAlternativeName` extensions.

CRL generation and certificate revocation are the same as for the root CA. The only thing different about the OCSP responder is the port; the subordinate CA should use `9081` instead. It's recommended that the responder uses its own certificate, which avoids keeping the subordinate CA on a public server.

2 Testing TLS with OpenSSL

Due to the large number of protocol features and implementation quirks, it's sometimes difficult to determine the exact configuration and features of secure servers. Although many tools exist for this purpose, it's often difficult to know exactly how they work, and that sometimes makes it difficult to fully trust their results. Even though I spent years testing secure servers and have access to good tools, when I really want to understand what is going on, I resort to using OpenSSL and Wireshark. I am not saying that you should use OpenSSL for everyday testing; on the contrary, you should find an automated tool that you trust. For online testing, I recommend Hardenize;[1] for offline work, consider `testssl.sh`.[2] But when you really need to be certain of something, the only way is to get your hands dirty with OpenSSL.

Custom-Compile OpenSSL for Testing

Using OpenSSL for testing purposes has become more difficult recently because, paradoxically, OpenSSL itself got better. In the aftermath of Heartbleed, the OpenSSL developers undertook a great overhaul, one aspect of which was removal of obsolete cryptography. That is great news for everyone, of course, but does make *our* lives more difficult. To test for a wide variety of conditions, we may need to use two versions: one recent and one old. The recent one is useful to test modern features (e.g., TLS 1.3), but the old one is what you need to test obsolete functionality.

At the time of writing, the new version will most definitely be from the 1.1.1 branch. As for the old, after some research, I settled on OpenSSL 1.0.2g, configured so that the removal of some obsolete features is reverted:

```
$ ./config \
--prefix=/opt/openssl-1.0.2g \
--openssldir=/opt/openssl-1.0.2g \
no-shared \
```

[1] Hardenize (retrieved 31 August 2020)

[2] testssl.sh (retrieved 29 August 2020)

```
enable-ssl2 \
enable-ssl3 \
enable-weak-ssl-ciphers
```

Throughout this chapter, I will refer to these two versions of OpenSSL as *new* and *old*. That's how you'll know which version to use for the testing. Refer to the previous chapter for more information on how to configure and install OpenSSL.

Connecting to TLS Services

OpenSSL comes with a client tool that you can use to connect to a secure server. The tool is similar to `telnet` or `nc` in the sense that it handles the encryption aspect but allows you to fully control the layer that comes next.

To connect to a server, you need to supply a hostname and a port. For example:

```
$ openssl s_client -crlf \
-connect www.feistyduck.com:443 \
-servername www.feistyduck.com
```

Notice that you had to supply the hostname twice. The `-connect` switch is used to establish the TCP connection, but `-servername` is used to specify the hostname sent at the TLS level. Starting with OpenSSL 1.1.1, the `s_client` tool automatically configures the latter. You'll still need to use the `-servername` switch if *(1)* you're using an earlier version of OpenSSL, *(2)* you're connecting to an IP address, or *(3)* the TLS host needs to be different. Use the `-noservername` switch to avoid sending hostname information in the TLS handshake.

Once you type the command, you're going to see a lot of diagnostic output (more about that in a moment) followed by an opportunity to type whatever you want. Because we're talking to an HTTP server, the most sensible thing to do is to submit an HTTP request. In the following example, I use a `HEAD` request because it instructs the server not to send the response body:

```
HEAD / HTTP/1.0
Host: www.feistyduck.com

HTTP/1.1 200 OK
Date: Mon, 24 Aug 2020 16:38:02 GMT
Server: Apache
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
Cache-control: no-cache, must-revalidate
Content-Type: text/html;charset=UTF-8
Transfer-Encoding: chunked
Set-Cookie: JSESSIONID=882D48C8842EA82E3F3AFACC4425A695; Path=/; Secure; HttpOnly
Connection: close

read:errno=0
```

> **Note**
>
> If, when connecting to a remote server in this way, the TLS handshake completes but you're getting disconnected after the first HTTP request line, check that you've specified the `-crlf` switch on the command line. This switch ensures that the newlines you type are translated to a carriage return plus line feed combo to ensure string HTTP compliance.

Now we know that the TLS communication layer is working: we got through to the HTTP server, submitted a request, and received a response back. Let's go back to the diagnostic output. The first couple of lines will show the information about the server certificate:

```
CONNECTED(00000003)
depth=2 C = GB, ST = Greater Manchester, L = Salford, O = COMODO CA Limited, CN = ↩
COMODO RSA Certification Authority
verify return:1
depth=1 C = GB, ST = Greater Manchester, L = Salford, O = COMODO CA Limited, CN = ↩
COMODO RSA Domain Validation Secure Server CA
verify return:1
depth=0 OU = Domain Control Validated, OU = PositiveSSL, CN = www.feistyduck.com
verify return:1
```

The next section in the output lists all the certificates presented by the server in the order in which they were delivered:

```
Certificate chain
 0 s:OU = Domain Control Validated, OU = PositiveSSL, CN = www.feistyduck.com
   i:C = GB, ST = Greater Manchester, L = Salford, O = COMODO CA Limited, CN = ↩
COMODO RSA Domain Validation Secure Server CA
 1 s:C = GB, ST = Greater Manchester, L = Salford, O = COMODO CA Limited, CN = ↩
COMODO RSA Domain Validation Secure Server CA
   i:C = GB, ST = Greater Manchester, L = Salford, O = COMODO CA Limited, CN = ↩
COMODO RSA Certification Authority
```

For each certificate, the first line shows the subject and the second line shows the issuer information.

This part is very useful when you need to see exactly what certificates are sent; browser certificate viewers typically display reconstructed certificate chains that can be almost completely different from the presented ones. To determine if the chain is nominally correct, you might wish to verify that the subjects and issuers match. You start with the leaf (web server) certificate at the top, and then you go down the list, matching the issuer of the current certificate to the subject of the next. The last issuer you see can point to some root certificate that is not in the chain, or—if the self-signed root is included—it can point to itself.

The next item in the output is the server certificate; it's a lot of text, but I'm going to remove most of it for brevity:

```
Server certificate
-----BEGIN CERTIFICATE-----
MIIFUzCCBDugAwIBAgIRAPR/CbWZEksfCIRqxNcesPIwDQYJKoZIhvcNAQELBQAw
gZAxCzAJBgNVBAYTAkdCMRswGQYDVQQIExJHcmVhdGVyIE1hbmNoZXN0ZXIxEDAO
BgNVBAcTB1NhbGZvcmQxGjAYBgNVBAoTEUNPTU9ETyBDQSBMaW1pdGVkMTYwNAYD
[...]
L1MPjFiB5pyvf9jDBxv8TmG4Q6TnDDhw2t2Qil6lhsPAMZ9odP22W3uaLE1y7aB6
zbQXjVsc3E1THfFZWRzDPsU4fN/1iGlbrcAWa2sFfhJXrCDfAowFJ8A1n9jMiNEG
WfQfGgA2ar2xUtsqA7Re6XlXOlwBPuQ=
-----END CERTIFICATE-----
subject=OU = Domain Control Validated, OU = PositiveSSL, CN = www.feistyduck.com
issuer=C = GB, ST = Greater Manchester, L = Salford, O = COMODO CA Limited, CN = ↩
COMODO RSA Domain Validation Secure Server CA
```

> **Note**
>
> By default, the `s_client` tool shows just the leaf certificate. If you wish to obtain the entire chain, use the `-showcerts` switch.

If you want to have a better look at the certificate, you'll first need to copy it from the output and store it in a separate file. I'll discuss how to do that in the next section.

The following is a lot of information about the TLS connection, most of which is self-explanatory:

```
---
No client certificate CA names sent
Peer signing digest: SHA512
Peer signature type: RSA
Server Temp Key: ECDH, P-256, 256 bits
---
SSL handshake has read 3624 bytes and written 446 bytes
Verification: OK
---
New, TLSv1.2, Cipher is ECDHE-RSA-AES128-GCM-SHA256
Server public key is 2048 bit
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
SSL-Session:
    Protocol  : TLSv1.2
    Cipher    : ECDHE-RSA-AES128-GCM-SHA256
    Session-ID: 73FC4831AF053C46291C2D8CC90BF7F1D5B12178E488FBB4DC49A302B870E8DE
    Session-ID-ctx:
    Master-Key: E60DA9C6669C2C7DFFD8A3AD2CD17405CC0B9B69C4184469D779A9BA19A6FD4B3D6↩
02A023BD8B23F8D9A9FF2CBB5DDF7
    PSK identity: None
```

```
PSK identity hint: None
SRP username: None
TLS session ticket lifetime hint: 300 (seconds)
TLS session ticket:
0000 - 31 95 ff d0 4c 42 dd d0-24 64 03 5c fc 55 1d 17   1...LB..$d.\.U..
0010 - 05 c4 61 1f b8 ba fd fe-f7 6c 6c e9 ae a2 49 3f   ..a......ll...I?
0020 - c5 19 d4 e9 69 a5 79 d5-af 13 26 c8 2c e7 f0 01   ....i.y...&.,...
0030 - 3b 42 d8 c0 29 4c fa 7e-88 aa 8d c8 0b 30 96 ce   ;B..)L.~.....0..
0040 - 43 40 2c 09 0b aa 2e d5-61 e3 34 7a a3 78 2f 93   C@,.....a.4z.x/.
0050 - 67 5a b9 96 78 f5 e7 69-b7 b6 2d 8c 00 8f 04 ab   gZ..x..i..-.....
0060 - 42 1d 26 db 92 ec 2d 2f-ba 1c c6 61 87 64 0e d5   B.&...-/...a.d..
0070 - f2 ce 20 d0 07 a5 e2 6d-c6 45 50 c2 45 14 a8 ee   .. ....m.EP.E...
0080 - 59 7c 63 e1 d7 d8 b0 b6-76 21 d2 13 97 eb bd 97   Y|c.....v!......
0090 - a1 d3 e8 5c 61 da da 2d-85 80 db ae de 56 97 e1   ...\a..-.....V..
00a0 - e8 7a 25 f9 bf cf b6 18-48 5b b0 03 a5 e6 ec 0a   .z%.....H[......
00b0 - bf 2f 0d 1a 6b ae 79 10-80 9c cf 4d 66 8f 90 43   ./..k.y....Mf..C
00c0 - 69 54 32 be 0c 89 57 e8-6d 81 b5 3e 5b cb 5e 8e   iT2...W.m..>[.^.

Start Time: 1598288068
Timeout   : 7200 (sec)
Verify return code: 0 (ok)
Extended master secret: no
```

The most important information here is the protocol version (TLS 1.2) and cipher suite used (`ECDHE-RSA-AES128-GCM-SHA256`). Do note that protocol information appears in two locations, which is potentially confusing when different versions are shown. The first location describes the minimum protocol requirement with the negotiated cipher suite, while the second location points to the actual protocol version currently being negotiated. You will see a difference in protocol versions with some older cipher suites—for example:

```
New, TLSv1/SSLv3, Cipher is DHE-RSA-AES128-SHA
```

The selected suite could be used with SSL 3.0, but it's used with TLS 1.2 on this connection:

```
Protocol  : TLSv1.2
Cipher    : DHE-RSA-AES128-SHAs
```

You can also determine that the server has issued to you a session ID and a TLS session ticket (a way of resuming sessions without having the server maintain state) and that secure renegotiation is supported.

> **Note**
>
> If you're connecting to a TLS 1.3 server, the output may be different. Sometimes you will observe less information initially, with additional information arriving later in bursts. This behavior depends on the implementation and reflects the changes in TLS 1.3, which transmits session tickets as separate protocol messages that are sent only

after the handshake is complete. Additionally, multiple session tickets are usually sent on the same connection.

Certificate Verification

Just because you are able to connect to a TLS server, that doesn't mean that the service is configured correctly, even if the server supports all the right protocols and cipher suites. It is equally important that the configured certificate matches the correct DNS names.

By default, the `s_client` tool reports but otherwise ignores certificate issues. Further, before you begin to trust its judgment you need to be confident that it can recognize a valid certificate when it sees one. This is especially true when you're using a custom-compiled binary.

In the example from the previous section, the verification status code (shown on the penultimate line) was `0`, which means that the verification has been successful. If you're connecting to a server that has a valid public certificate but you see status `20` instead, that probably means that trusted roots haven't been correctly configured:

```
Verify return code: 20 (unable to get local issuer certificate)
```

At this point, if you don't wish to fix your OpenSSL installation, you can instead use the `-CApath` switch to point to the location where the roots are kept. For example:

```
$ openssl s_client -connect www.feistyduck.com:443 -CApath /etc/ssl/certs/
```

If you instead have a single file with the roots in it, use the `-CAfile` switch:

```
$ openssl s_client -connect www.feistyduck.com:443 \
-CAfile /etc/ssl/certs/ca-certificates.crt
```

Even if you get a successful status code at this point, that doesn't mean that the certificate is correctly configured. That's because the `s_client` tool doesn't check that the certificate is correct for the given hostname; you have to tell it to do that manually and tell it which hostname to use:

```
$ openssl s_client -connect www.feistyduck.com:443 -verify_hostname ↵
www.feistyduck.com
```

If there is a mismatch, you might see status code 62:

```
Verify return code: 62 (Hostname mismatch)
```

Otherwise, you'll see the familiar status code `0`. In the rare instance that you need to verify a certificate that has been issued for an IP address instead of a hostname, you'll need to use the `-verify_ip` switch for the verification.

Testing Protocols that Upgrade to TLS

When used with HTTP, TLS wraps the entire plain-text communication channel to form HTTPS. Some other protocols start off as plaintext, but then they upgrade to encryption. If you want to test such a protocol, you'll have to tell OpenSSL which protocol it is so that it can upgrade on your behalf. Provide the protocol information using the `-starttls` switch. For example:

```
$ openssl s_client -connect gmail-smtp-in.l.google.com:25 -starttls smtp
```

At the time of writing, the supported protocols in recent OpenSSL releases are `smtp`, `pop3`, `imap`, `ftp`, `xmpp`, `xmpp-server`, `irc`, `postgres`, `mysql`, `lmtp`, `nntp`, `sieve`, and `ldap`. There is less choice with OpenSSL 1.0.2g: `smtp`, `pop3`, `imap`, `ftp`, and `xmpp`.

Some protocols require the client to provide their names. For example, for SMTP, OpenSSL will use `mail.example.com` by default, but you can specify the correct value with the `-name` switch. If you're testing XMPP, you may need to specify the correct server name; you can do this with the `-xmpphost` switch.

Extracting Remote Certificates

When you connect to a remote secure server using `s_client`, it will dump the server's PEM-encoded certificate to standard output. If you need the certificate for any reason, you can copy it from the scroll-back buffer. If you know in advance you only want to retrieve the certificate, you can use this command line as a shortcut:

```
$ echo | openssl s_client -connect www.feistyduck.com:443 2>&1 | sed --quiet '↵
/-BEGIN CERTIFICATE-/,/-END CERTIFICATE-/p' > feistyduck.crt
```

The purpose of the `echo` command at the beginning is to separate your shell from `s_client`. If you don't do that, `s_client` will wait for your input until the server times out (which may potentially take a very long time).

By default, `s_client` will print only the leaf certificate; if you want to print the entire chain, give it the `-showcerts` switch. With that switch enabled, the previous command line will place all the certificates in the same file.

```
$ echo | openssl s_client -showcerts -connect www.feistyduck.com:443 2>&1 | sed ↵
--quiet '/-BEGIN CERTIFICATE-/,/-END CERTIFICATE-/p' > feistyduck.chain
```

Another useful trick is to pipe the output of `s_client` directly to the `x509` tool. The following command shows detailed server information, along with its SHA256 fingerprint:

```
$ echo | openssl s_client -connect www.feistyduck.com:443 2>&1 | openssl x509 ↵
-noout -text -fingerprint -sha256
```

Sometimes you will need to take the certificate fingerprint and use it with other tools. Unfortunately, OpenSSL outputs certificates in a format that shows individual bytes and separates them using colons. This handy command line normalizes certificate fingerprints by removing the colons and converting the hexadecimal characters to lowercase:

```
$ echo | openssl s_client -connect www.feistyduck.com:443 2>&1 | openssl x509 ↩
-noout -fingerprint -sha256 | sed 's/://g' | tr '[:upper:]' '[:lower:]' | sed 's↩
/sha256 fingerprint=//g'
```

> **Note**
>
> Connecting to remote TLS servers and reviewing their certificates is a pretty common operation, but you shouldn't spend your time remembering and typing these long commands. Instead, invest into writing a couple of shell functions that will package this functionality into easy-to-use commands.

Testing Protocol Support

By default, `s_client` will try to use the best protocol to talk to the remote server and report the negotiated version in output. As mentioned earlier, you will find the protocol version in the output twice, and you want the line that explicitly talks about the protocol:[3]

```
    Protocol  : TLSv1.2
```

If you need to test support for specific protocol versions, you have two options. You can explicitly choose one protocol to test by supplying one of the `-ssl2`, `-ssl3`, `-tls1`, `-tls1_1`, `-tls1_2`, or `tls1_3` switches. Naturally, each switch requires support for a specific protocol version in the testing tool. If you want to exclude a particular protocol from the testing, there is a family of switches that disable protocols (e.g., `-no_tls_1_2` for TLS 1.2). Sometimes that may be the better approach. Starting with OpenSSL 1.1.0, there are two new options, `-min_protocol` and `-max_protocol`, which control the minimum and maximum protocol version, respectively.

For example, here's the output you might get when testing a server that doesn't support a certain protocol version:

```
$ openssl s_client -connect www.example.com:443 -tls1_2
CONNECTED(00000003)
140455015261856:error:1408F10B:SSL routines:SSL3_GET_RECORD:wrong version ↩
number:s3_pkt.c:340:
---
no peer certificate available
---
```

[3] Do note that when connecting to a TLS v1.3 server, the protocol information may not appear immediately on the connection. Instead, it will be printed when the session ticket is received, which may take a couple of seconds with some servers.

```
No client certificate CA names sent
---
SSL handshake has read 5 bytes and written 7 bytes
---
New, (NONE), Cipher is (NONE)
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
SSL-Session:
    Protocol  : TLSv1.2
    Cipher    : 0000
    Session-ID:
    Session-ID-ctx:
    Master-Key:
    Key-Arg   : None
    PSK identity: None
    PSK identity hint: None
    SRP username: None
    Start Time: 1339231204
    Timeout   : 7200 (sec)
    Verify return code: 0 (ok)
---
```

Understanding if a server supports SSL 2.0 may sometimes require more work, due to the fact that this old and very insecure version of the SSL protocol uses a different handshake from that used from SSL 3.0 onwards. Although servers that support only SSL 2.0 should now be very rare, to check this eventuality, you'll need to submit a separate check using the `-ssl2` switch.

Another protocol difference is that SSL 2.0 servers are sometimes seen without any configured cipher suites. In that case, although SSL 2.0 is supported, technically speaking, any handshake attempts will still fail. You should treat this situation as misconfiguration.

Testing Cipher Suite Configuration

It's not very likely that you will be spending a lot of time testing cipher suite configuration using OpenSSL on the command line. This is because you can effectively test for only one suite at a time; testing for more than 300 cipher suites that are supported by TLS 1.2 and earlier protocol revisions would take a considerable amount of time. This is a perfect opportunity to use those handy tools that automate the process.

Still, there will be times when you will need to probe servers to determine if they support a particular suite or a cryptographic primitive, or if the preference is correctly configured.

The introduction of TLS 1.3 made testing in this area slightly more complicated, but it's still manageable. Because of the differences between this protocol version and all other revisions, it's usually best to split your tests into two groups. When testing TLS 1.3, always use the -

ciphersuites switch in combination with -tls1_3. The usual approach is to specify only one suite to determine if it's supported:

```
$ echo | openssl s_client -connect www.hardenize.com:443 -tls1_3 -ciphersuites TLS↩
_AES_128_GCM_SHA256 2>/dev/null| grep New
New, TLSv1.3, Cipher is TLS_AES_128_GCM_SHA256
```

The output will naturally be different if you pick a suite that is not supported:

```
$ echo | openssl s_client -connect www.hardenize.com:443 -tls1_3 -ciphersuites TLS↩
_AES_128_CCM_SHA256 2>/dev/null | grep New
New, (NONE), Cipher is (NONE)
```

When you're testing the configuration of TLS 1.2 and earlier protocol versions, use the -cipher switch in combination with -no_tls1_3 (assuming you're using a version of OpenSSL that supports TLS 1.3):

```
$ echo | openssl s_client -connect www.hardenize.com:443 -no_tls1_3 -cipher AESGCM ↩
2>/dev/null | grep New
New, TLSv1.2, Cipher is ECDHE-ECDSA-AES128-GCM-SHA256
```

As you can see in the previous example, when testing TLS 1.2 and earlier you don't have to specify only one cipher suite, but in that case you will need to observe what has been negotiated. If you want to probe further, you can always tweak the command line to remove the previously negotiated suite:

```
$ echo | openssl s_client -connect www.hardenize.com:443 -no_tls1_3 -cipher ↩
'AESGCM:!ECDHE-ECDSA-AES128-GCM-SHA256' 2>/dev/null | grep New
New, TLSv1.2, Cipher is ECDHE-ECDSA-AES256-GCM-SHA384
```

Even though you won't be testing for a great many suites manually, there is a quick way to determine if a particular server supports any of the many bad cryptographic primitives. To do this, use your *old* OpenSSL version and list all the bad cipher suite keywords, like this:

```
$ echo | openssl s_client -connect example.com:443 -cipher '3DES DES RC2 RC4 IDEA ↩
SEED CAMELLIA MD5 aNULL eNULL EXPORT LOW' 2>/dev/null | grep New
New, TLSv1/SSLv3, Cipher is DHE-RSA-CAMELLIA256-SHA
```

Another good test is to see if a server supports the RSA key exchange that doesn't support forward secrecy:

```
$ echo | /opt/openssl-1.0.2g/bin/openssl s_client -connect example.com:443 -cipher ↩
kRSA 2>/dev/null | grep New
New, TLSv1/SSLv3, Cipher is AES128-GCM-SHA256
```

Ideally, you'd get a handshake failure here, but it's not terrible if you don't, provided the server uses the RSA key exchange only as a matter of last resort. You can check this by offering suites with forward secrecy as your least preferred option:

```
$ echo | /opt/openssl-1.0.2g/bin/openssl s_client -connect example.com:443 -cipher ↩
'DHE ECDHE kRSA +kECDHE +kDHE' 2>/dev/null | grep New
New, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES128-GCM-SHA256
```

Testing Cipher Suite Preference

As a general rule, TLS servers should always be configured to enforce their cipher suite preferences, ensuring that they negotiate their preferred cipher suite with every client. This feature is essential with TLS 1.2 and earlier protocol revisions, which support many cipher suites, most of them undesirable. It's a different story with TLS 1.3: it only has a handful of suites available at this time and all of them are secure, so enforcing server preference doesn't matter that much.[4]

> **Note**
>
> Because cipher suite preference doesn't matter much with TLS 1.3, some stacks don't even support it with this protocol, even if they do with earlier protocol versions. Thus, for the best results, you will want to test separately for TLS 1.3 and everything else—or separately for every supported protocol. This is another case in which automation is the better choice.

To test for server suite preference, you first need to have some idea of what suites are supported. For example, you could have the complete list of supported suites. Alternatively, you can probe the server with different suite types—for example, those that use ECDHE versus DHE or RSA key exchange.

With two suites in hand, you need to initiate two connections, first offering one of the suites as your first choice, then the other:

```
$ echo | openssl s_client -connect www.hardenize.com:443 -tls1_3 -ciphersuites ↩
'TLS_AES_128_GCM_SHA256:TLS_AES_256_GCM_SHA384' 2>/dev/null | grep New
New, TLSv1.3, Cipher is TLS_AES_128_GCM_SHA256
zoom:~ ivanr$ echo | openssl s_client -connect www.hardenize.com:443 -tls1_3 ↩
-ciphersuites 'TLS_AES_256_GCM_SHA384:TLS_AES_128_GCM_SHA256' 2>/dev/null | grep ↩
New
New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384
```

If you see the same suite negotiated on both connections, that means that the server is configured to actively select negotiated suites. Otherwise, it isn't. The server in the previous example is one of those TLS 1.3 servers that doesn't enforce preference. That very same server

[4] Out of five cipher suites that TLS v1.3 supports, OpenSSL enables only three by default. The remaining two are CCM suites, which are intended for use with embedded systems, in which every little bit of performance and battery life matters. These suites are not worth using for other use cases—especially the CCM_8 suite, which reduces the strength of authentication.

does have a preference with TLS 1.2; we can see that it always selects a better suite, even when we push it to the end of our list:

```
$ echo | openssl s_client -connect www.hardenize.com:443 -no_tls1_3 -cipher ↩
'ECDHE+AESGCM RSA' 2>/dev/null | grep New
New, TLSv1.2, Cipher is ECDHE-ECDSA-AES128-GCM-SHA256
$ echo | openssl s_client -connect www.hardenize.com:443 -no_tls1_3 -cipher ↩
'ECDHE+AESGCM RSA +ECDHE' 2>/dev/null | grep New
New, TLSv1.2, Cipher is ECDHE-ECDSA-AES128-GCM-SHA256
```

When it comes to server suite preference testing, the ChaCha20 suites are best avoided. This is because some servers support another type of preference, where they treat AES-GCM and ChaCha20 suites as equal in terms of security and respect client preference as a special case. The idea is that the client will prefer the faster cipher suite, which is typically ChaCha20 for mobile devices and AES-GCM for desktops.

That said, with servers that support this type of preference, you may want to test if it's working correctly. To do that, you'll need to use three supported cipher suites and three tests. The purpose of the first two tests is to establish that the server selects its favorite suite when ChaCha20 is not involved:

```
$ echo | openssl s_client -connect www.hardenize.com:443 -no_tls1_3 -cipher ↩
'ECDHE-RSA-AES128-GCM-SHA256 ECDHE-ECDSA-AES128-GCM-SHA256 ECDHE-ECDSA-CHACHA20-POL↩
Y1305'  2>/dev/null | grep New
New, TLSv1.2, Cipher is ECDHE-ECDSA-AES128-GCM-SHA256
$ echo | openssl s_client -connect www.hardenize.com:443 -no_tls1_3 -cipher ↩
'ECDHE-ECDSA-AES128-GCM-SHA256 ECDHE-RSA-AES128-GCM-SHA256  ECDHE-ECDSA-CHACHA20-PO↩
LY1305'  2>/dev/null | grep New
New, TLSv1.2, Cipher is ECDHE-ECDSA-AES128-GCM-SHA256
```

If you see that the server responds with the same suite in both cases, you can submit another test with a supported ChaCha20 suite first. If you see the server selecting it, you know it's configured to support the client-preferred suite:

```
$ echo | openssl s_client -connect www.hardenize.com:443 -no_tls1_3 -cipher ↩
'ECDHE-ECDSA-CHACHA20-POLY1305 ECDHE-RSA-AES128-GCM-SHA256 ECDHE-ECDSA-AES128-GCM-S↩
HA256'  2>/dev/null | grep New
New, TLSv1.2, Cipher is ECDHE-ECDSA-CHACHA20-POLY1305
```

Testing Named Groups

Named groups are predefined cryptographic parameters that are used for key exchange. In TLS 1.3, named groups include both elliptic curve (EC) and finite field (DH) parameters. TLS 1.2 and earlier generally use only predefined elliptic curves; the server provides DH parameters

on every connection.[5] In a handshake, the client and server have to agree on a common named group over which the key exchange will take place, and it's important that the selected group satisfies desired security requirements.

In practice, there is seldom a need to test servers for named groups. Although there's a fair number of named groups in various RFCs, OpenSSL is probably the only major client to have extensive support. Historically, you could only use NIST's P-256 and P-384 EC groups because these were the only widely supported curves. Relatively recently, X25519 and X448 groups were added as an alternative. Because all these curves are strong, there is little need to spend time thinking about them.

You may find yourself testing named group configuration usually to understand what your web server is doing. For example, you may care about X25519 and want to ensure it's available and preferred. To test for this, use the `s_client` tool and the `-curves` switch. For example, here's how to determine if a single named group is supported:

```
$ echo | openssl s_client -connect hardenize.com:443 -curves X25519 2>/dev/null | ↩
grep "Server Temp Key"
Server Temp Key: X25519, 253 bits
```

On success, you will see the named group in the output, because that's the group that was selected for the handshake. On failure, you may see no output, which means that the handshake failed. Alternatively, the server, unable to negotiate an ECDHE suite, may fall back to a DHE suite, indicated by the following output:

```
Server Temp Key: DH, 2048 bits
```

If you need to test for named group preference, you need to offer two or more named groups, with your preferred one last. If you see it negotiated, that will mean that the server actively chooses the group it considers most appropriate. Use colons to separate the groups and be aware that the names are case-sensitive.

```
$ echo | openssl s_client -connect hardenize.com:443 -curves prime256v1:X25519 ↩
2>/dev/null | grep "Server Temp Key"
Server Temp Key: X25519, 253 bits
```

> **Note**
>
> You can get the complete list of elliptic curves supported by OpenSSL using the `ecparam` tool and the `-list_curves` switch. To that list, add X25519 and X448. Support for finite field groups is currently not available but should arrive with OpenSSL 3.0.

[5] RFC 7919, which came out in 2016, redefined the `elliptic_curves` TLS extension to support finite field groups and changed the extension name to `supported_groups`. Although this extension applies to TLS 1.2, support for it is not widespread.

Testing DANE

DNS-based Authentication of Named Entities (DANE) is set of standards that enables you to endorse the TLS certificates you use via DNS configuration. For this to work, DANE requires DNS itself to be secure, which means that DNSSEC is necessary. Therefore, DANE is essentially a mechanism for pinning; only the certificates you approve will be accepted as valid by DANE-enabled clients. DANE itself is not controversial, but DNSSEC, on which it relies, is a very divisive topic, with the world split between those who love it and those who hate it. As a result, DANE is currently not universally supported. It's more commonly used to secure SMTP servers; there is no support at the browser level.

Supporting DANE adds some complexity to your TLS deployments because of the way DNS configuration is propagated and cached. Before you use a new certificate you need to ensure that your new DNS configuration (endorsing that certificate) is fully propagated. Thus, you would typically first publish your DNS changes, wait for a period time sufficient for the caches to clear, and only then deploy the certificates.[6]

The testing itself is straightforward; you use the `s_client` tool while feeding it DANE data. This is handy because it enables you to test a connection even before making DNS changes. First, let's see what DANE configuration looks like.

DANE stores configuration in TLSA resource records, using two prefix labels to indicate the protocol and port:

```
$ host -t TLSA _25._tcp.mail.protonmail.ch
_25._tcp.mail.protonmail.ch has TLSA record 3 1 1 76BB66711DA416433CA890A5B2E5A0533↩
C6006478F7D10A4469A947A CC8399E1
_25._tcp.mail.protonmail.ch has TLSA record 3 1 1 6111A5698D23C89E09C36FF833C1487ED↩
C1B0C841F87C49DAE8F7A09 E11E979E
```

This output contains two endorsements, one per certificate. Having two endorsements is not unusual. For example, perhaps you might have a service that uses two certificates (e.g., one with an RSA key and another with an ECDSA key), or you have a backup certificate, or you're simply in a transitional period when you're switching certificates. The three numbers at the beginning indicate that the endorsement targets the certificate directly (3) via its public key (1) and a SHA256 hash (1). The rest of the data is the hash itself.

To test, you connect to the SMTP service while providing the DANE data using the `-dane_tlsa_domain` and `-dane_tlsa_rrdata` switches:

```
$ openssl s_client -starttls smtp \
-connect mail.protonmail.ch:25 \
-dane_tlsa_domain mail.protonmail.ch \
-dane_tlsa_rrdata "3 1 1 76BB66711DA416433CA890A5B2E5A0533C6006478F7D10A4469A947ACC↩
```

[6] New Adventures in DNSSEC and DANE (Jan Schaumann, retrieved 2 October 2020)

```
8399E1"
```

If the verification is successful, you will see something like this in the output:

```
---
SSL handshake has read 5209 bytes and written 433 bytes
Verification: OK
Verified peername: *.protonmail.ch
DANE TLSA 3 1 1 ...8f7d10a4469a947acc8399e1 matched EE certificate at depth 0
---
```

If you'd like to test for validation failure, just break the supplied hash. The result will be similar to the following output:

```
---
SSL handshake has read 5209 bytes and written 433 bytes
Verification error: No matching DANE TLSA records
---
```

For the best results, when testing DANE in this way, always provide all known TLSA records (one per `-dane_tlsa_rrdata` switch). If you do, services that use multiple certificates simultaneously will check out no matter what certificate is negotiated. For TLS 1.2 and earlier, it's possible to force a particular certificate via a choice of client-supported cipher suites (the `-cipher` switch). TLS 1.3 suites are different, and for this protocol version you would need to use the `-sigalgs` switch with a value such as `ecdsa_secp256r1_sha256` or `rsa_pss_rsae_sha256`.

Testing Session Resumption

When coupled with the `-reconnect` switch, the `s_client` command can be used to test session reuse. In this mode, `s_client` will connect to the target server six times. It will create a new session on the first connection, then try to reuse the same session in the subsequent five connections:

```
$ echo | openssl s_client -connect www.feistyduck.com:443 -reconnect
```

> **Note**
>
> Due to a bug in OpenSSL, at the time of writing session resumption testing doesn't work in combination with TLS 1.3. Until the bug is resolved,[7] the best you can do is test the earlier protocol versions. Use the `-no_tls1_3` switch.

The previous command will produce a sea of output, most of which you won't care about. The key parts are the information about new and reused sessions. There should be only one new session at the beginning, indicated by the following line:

[7] s_client -reconnect Option Is Broken with TLSv1.3 (OpenSSL; retrieved 31 August 2020)

```
New, TLSv1.2, Cipher is ECDHE-RSA-AES128-GCM-SHA256
```

This is followed by five session reuses, indicated by lines like this:

```
Reused, TLSv1.2, Cipher is ECDHE-RSA-AES128-GCM-SHA256
```

Most of the time, you don't want to look at all that output and want an answer quickly. You can get it using the following command line:

```
$ echo | openssl s_client -connect www.feistyduck.com:443 -reconnect 2> /dev/null ↵
| grep 'New\|Reuse'
New, TLSv1.2, Cipher is ECDHE-RSA-AES128-GCM-SHA256
Reused, TLSv1.2, Cipher is ECDHE-RSA-AES128-GCM-SHA256
Reused, TLSv1.2, Cipher is ECDHE-RSA-AES128-GCM-SHA256
Reused, TLSv1.2, Cipher is ECDHE-RSA-AES128-GCM-SHA256
Reused, TLSv1.2, Cipher is ECDHE-RSA-AES128-GCM-SHA256
Reused, TLSv1.2, Cipher is ECDHE-RSA-AES128-GCM-SHA256
```

Here's what the command does:

- The `-reconnect` switch activates the session reuse mode.
- The `2> /dev/null` part hides `stderr` output, which you don't care about.
- Finally, the piped `grep` command filters out the rest of the fluff and lets through only the lines that you care about.

> **Note**
>
> If you don't want to include session tickets in the test—for example, because not all clients support this feature yet—you can disable this method of resumption using the `-no_ticket` switch. This option doesn't apply to TLS 1.3.

Keeping Session State across Connections

If you need better control over resumption, the `s_client` tool provides options to persist the connection state to a file. On your first connection, use the `-sess_out` switch to record the state:

```
$ openssl s_client -connect www.feistyduck.com:443 -sess_out sess.pem
```

To view the recorded state, use the `sess_id` tool:

```
$ openssl sess_id -in sess.pem -noout -text
SSL-Session:
    Protocol  : TLSv1.2
    Cipher    : ECDHE-ECDSA-AES128-GCM-SHA256
    Session-ID: F7384C2C4BE621F66045ECE12A89821FEE789C2E75B78C90C428BE37E0FE4599
    Session-ID-ctx:
    Master-Key: 9D39C582D9AA1618B2F16C7911C4BFFB61D6D1FD578A93B1145FD2B4DBFDE76EB22↵
```

```
79BA50AEFFCD95320BEEBC9489FAF
    PSK identity: None
    PSK identity hint: None
    SRP username: None
    TLS session ticket lifetime hint: 64800 (seconds)
    TLS session ticket:
    0000 - a2 d3 e3 04 03 21 85 6d-1a 4f 9c 82 fc 4e 15 e0   .....!.m.O...N..
    0010 - 9b b8 b1 24 0d 95 a3 0a-b8 24 d4 f5 d2 be b8 56   ...$.....$.....V
    0020 - b2 f0 e9 c5 e5 53 31 b5-24 74 96 ba e4 56 32 68   .....S1.$t...V2h
    0030 - fe bb 7a 7f 28 d7 c4 19-6a c5 ca 22 3a a7 2d 45   ..z.(...j.."":.-E
    0040 - 52 91 74 f7 a8 fa 75 40-02 b9 84 9c 84 0d a8 06   R.t...u@........
    0050 - c7 a1 65 af 8b 54 19 74-52 e8 c4 f4 47 1c 3f f0   ..e..T.tR...G.?.
    0060 - 46 35 1a 3c a9 a5 73 30-33 b7 20 bd dc 8a b8 f9   F5.<..s03. .....
    0070 - 79 20 4a de b3 60 83 53-c7 a7 62 e1 a2 9e 55 8c   y J..`.S..b...U.
    0080 - 24 0a f5 4c ab 81 a5 d9-36 ae 52 61 a1 4e b7 99   $..L....6.Ra.N..
    0090 - 20 9e ca 67 49 ea 80 a4-14 ce ac 36 aa 20 0e 53    ..gI......6. .S
    00a0 - d7 9f 14 a6 7c b9 88 4c-6b 69 93 d4 62 fb 02 50   ....|..Lki..b..P

    Start Time: 1602414785
    Timeout   : 300 (sec)
    Verify return code: 20 (unable to get local issuer certificate)
    Extended master secret: no
```

Finally, to connect again using the same session state, use the `-sess_in` switch:

```
$ openssl s_client -connect www.feistyduck.com:443 -sess_in sess.pem
```

Keeping the state across connections in this way gives you more control and enables you to completely change connection parameters from one connection to another. For example, you could connect to one server on your first attempt, then another server on your second. This may be of use when you need to test if session resumption is correctly implemented on a web server cluster. Manual control of your connections allows you to spread them over time, perhaps testing for session timeouts and ticket key rotation.

Checking OCSP Revocation

If an OCSP responder is malfunctioning, sometimes it's difficult to understand exactly why. Checking certificate revocation status from the command line is possible, but it's not quite straightforward. You need to perform the following steps:

1. Obtain the certificate that you wish to check for revocation.
2. Obtain the issuing certificate.
3. Determine the URL of the OCSP responder.
4. Submit an OCSP request and observe the response.

For the first two steps, connect to the server with the `-showcerts` switch specified:

```
$ openssl s_client -connect www.feistyduck.com:443 -showcerts
```

The first certificate in the output will be the one belonging to the server. If the certificate chain is properly configured, the second certificate will be that of the issuer. To confirm, check that the issuer of the first certificate matches the subject of the second:

```
Certificate chain
 0 s:OU = Domain Control Validated, OU = PositiveSSL, CN = www.feistyduck.com
   i:C = GB, ST = Greater Manchester, L = Salford, O = COMODO CA Limited, CN = ↩
COMODO RSA Domain Validation Secure Server CA
-----BEGIN CERTIFICATE-----
MIIFUzCCBDugAwIBAgIRAPR/CbWZEksfCIRqxNcesPIwDQYJKoZIhvcNAQELBQAw
[...]
zbQXjVsc3E1THfFZWRzDPsU4fN/1iGlbrcAWa2sFfhJXrCDfAowFJ8A1n9jMiNEG
WfQfGgA2ar2xUtsqA7Re6XlXOlwBPuQ=
-----END CERTIFICATE-----
 1 s:C = GB, ST = Greater Manchester, L = Salford, O = COMODO CA Limited, CN = ↩
COMODO RSA Domain Validation Secure Server CA
   i:C = GB, ST = Greater Manchester, L = Salford, O = COMODO CA Limited, CN = ↩
COMODO RSA Certification Authority
-----BEGIN CERTIFICATE-----
MIIGCDCCA/CgAwIBAgIQKy5u6tl1NmwUim7bo3yMBzANBgkqhkiG9w0BAQwFADCB
hTELMAkGA1UEBhMCR0IxGzAZBgNVBAgTEkdyZWF0ZXIgTWFuY2hlc3RlcjEQMA4G
[...]
```

If the second certificate isn't the right one, check the rest of the chain; some servers don't serve the chain in the correct order. If you can't find the issuer certificate in the chain, you'll have to find it somewhere else. One way to do that is to look for the *Authority Information Access* extension in the leaf certificate:

```
$ openssl x509 -in fd.crt -noout -text
[...]
    Authority Information Access:
        CA Issuers - URI:http://crt.comodoca.com/COMODORSADomainValidationSecureSer↩
verCA.crt
        OCSP - URI:http://ocsp.comodoca.com
```

If the *CA Issuers* information is present, it should contain the URL of the issuer certificate. If the issuer certificate information isn't available, you can try to open the site in a browser, let it reconstruct the chain, and download the issuing certificate from its certificate viewer. If all that fails, you can look for the certificate in your trust store or visit the CA's web site.

If you already have the certificates and just need to know the address of the OCSP responder, use the `-ocsp_uri` switch with the `x509` command as a shortcut:

```
$ openssl x509 -in fd.crt -noout -ocsp_uri
http://ocsp.comodoca.com
```

Now you can submit the OCSP request:

```
$ openssl ocsp -issuer issuer.crt -cert fd.crt -url http://ocsp.comodoca.com ↩
-CAfile issuer.crt
WARNING: no nonce in response
Response verify OK
fd.crt: good
        This Update: Aug 30 22:35:12 2020 GMT
  Next Update: Sep  6 22:35:12 2020 GMT
```

You want to look for two things in the response. First, check that the response itself is valid (`Response verify OK` in the previous example), and second, check what the response said. When you see `good` as the status, that means that the certificate hasn't been revoked. The status will be `revoked` for revoked certificates.

> **Note**
>
> The warning message about the missing nonce is telling you that OpenSSL wanted to use a nonce as a protection against replay attacks, but the server in question did not reply with one. This generally happens because CAs want to improve the performance of their OCSP responders. When they disable the nonce protection (the standard allows it), OCSP responses can be produced (usually in batch), cached, and reused for a period of time.

You may encounter OCSP responders that do not respond successfully to the previous command line. The following suggestions may help in such situations.

Do not request a nonce
: Some servers cannot handle nonce requests and respond with errors. OpenSSL will request a nonce by default. To disable nonces, use the `-no_nonce` command-line switch.

Supply a Host request header
: Although most OCSP servers respond to HTTP requests that don't specify the correct hostname in the `Host` header, some don't. If you encounter an error message that includes an HTTP error code (e.g., 404), try adding the hostname to your OCSP request. You can do this with the help of the `-header` switch.

With the previous two points in mind, the final command to use is the following:

```
$ openssl ocsp -issuer issuer.crt -cert fd.crt -url http://ocsp.comodoca.com ↩
-CAfile issuer.crt -no_nonce -header Host ocsp.comodoca.com
```

Testing OCSP Stapling

OCSP stapling is an optional feature that allows a server certificate to be accompanied by an OCSP response that proves its validity. Because the OCSP response is delivered over an already existing connection, the client does not have to fetch it separately.

OCSP stapling is used only if requested by a client, which submits the `status_request` extension in the handshake request. A server that supports OCSP stapling will respond by including an OCSP response as part of the handshake.

When using the `s_client tool`, OCSP stapling is requested with the `-status` switch:

```
$ echo | openssl s_client -connect www.feistyduck.com:443 -status
```

The OCSP-related information will be displayed at the very beginning of the connection output. For example, with a server that does not support stapling you will see this line near the top of the output:

```
CONNECTED(00000003)
OCSP response: no response sent
```

With a server that does support stapling, you will see the entire OCSP response in the output:

```
OCSP Response Data:
    OCSP Response Status: successful (0x0)
    Response Type: Basic OCSP Response
    Version: 1 (0x0)
    Responder Id: 90AF6A3A945A0BD890EA125673DF43B43A28DAE7
    Produced At: Aug 30 22:35:12 2020 GMT
    Responses:
    Certificate ID:
      Hash Algorithm: sha1
      Issuer Name Hash: 7AE13EE8A0C42A2CB428CBE7A605461940E2A1E9
      Issuer Key Hash: 90AF6A3A945A0BD890EA125673DF43B43A28DAE7
      Serial Number: F47F09B599124B1F08846AC4D71EB0F2
    Cert Status: good
    This Update: Aug 30 22:35:12 2020 GMT
    Next Update: Sep  6 22:35:12 2020 GMT

    Signature Algorithm: sha256WithRSAEncryption
         1b:9d:be:3e:e6:b2:9a:e6:22:fe:69:cc:55:a9:62:5d:29:79:
         [...]
```

The certificate status good means that the certificate has not been revoked.

Checking CRL Revocation

Checking certificate verification with a *Certificate Revocation List* (CRL) is even more involved than doing the same via OCSP. The process is as follows:

1. Obtain the certificate you wish to check for revocation.
2. Obtain the issuing certificate.
3. Download and verify the CRL.

4. Look for the certificate serial number in the CRL.

The first steps overlap with OCSP checking; to complete them follow the instructions in the section called "Checking OCSP Revocation".

The location of the CRL is encoded in the server certificate; look for the "X509v3 CRL Distribution Points" section in the text output. For example:

```
$ openssl x509 -in fd.crt -noout -text | grep -A 5 CRL
[...]
                  URI:http://crl.comodoca.com/COMODORSADomainValidationSecureServer↩
CA.crl
```

Then fetch the CRL from the CA:

```
$ wget http://crl.comodoca.com/COMODORSADomainValidationSecureServerCA.crl -O ↩
comodo.crl
```

Verify that the CRL is valid (i.e., signed by the issuer certificate):

```
$ openssl crl -in comodo.crl -inform DER -CAfile issuer.crt -noout
verify OK
```

Now, determine the serial number of the certificate you wish to check:

```
$ openssl x509 -in fd.crt -noout -serial
serial=F47F09B599124B1F08846AC4D71EB0F2
```

At this point, you can convert the CRL into a human-readable format and inspect it manually:

```
$ openssl crl -in comodo.crl -inform DER -text -noout
Certificate Revocation List (CRL):
        Version 2 (0x1)
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = GB, ST = Greater Manchester, L = Salford, O = COMODO CA ↩
Limited, CN = COMODO RSA Domain Validation Secure Server CA
        Last Update: Aug 31 07:52:03 2020 GMT
        Next Update: Sep  7 07:52:03 2020 GMT
        CRL extensions:
            X509v3 Authority Key Identifier:
                keyid:90:AF:6A:3A:94:5A:0B:D8:90:EA:12:56:73:DF:43:B4:3A:28:DA:E7

            X509v3 CRL Number:
                2149
            1.3.6.1.4.1.311.21.4:
200903195203Z   .
Revoked Certificates:
    Serial Number: 70DAB4B3229280F04364BC58DB2AB922
        Revocation Date: May 29 12:18:27 2017 GMT
```

```
Serial Number: 51894D40389CDAB84A7A6F3374E1D893
    Revocation Date: May 30 23:20:55 2017 GMT
[...]
Signature Algorithm: sha256WithRSAEncryption
     5a:7c:6e:6e:98:05:c4:24:2b:84:7a:28:6f:45:26:33:6b:88:
     4d:dd:61:22:e4:23:47:76:c7:8a:55:ec:f9:72:29:47:21:73:
     [...]
```

The CRL starts with some metadata, which is followed by a list of revoked certificates, and it ends with a signature (which we verified in the previous step). If the serial number of the server certificate is on the list, that means it had been revoked.

If you don't want to look for the serial number visually (some CRLs can be quite long), grep for it, but be careful that your formatting matches that used by the `crl` tool. For example:

```
$ openssl crl -in comodo.crl -inform DER -text -noout | grep F47F09B599124B1F08846A↩
C4D71EB0F2
```

Testing Renegotiation

In TLS, renegotiation is a failed feature that was responsible for several protocol weaknesses, some of which are quite easy to exploit. TLS 1.3 no longer supports renegotiation, but there are still older servers out there that support it with earlier protocol revisions.

The `s_client` tool has a couple of features that can assist you with manual testing of renegotiation. First of all, when you connect, the tool will report if the remote server supports secure renegotiation. This is because a server that supports secure renegotiation indicates its support for it via a special TLS extension that is exchanged during the handshake phase. When support is available, the output may look like this:

```
New, TLSv1/SSLv3, Cipher is AES256-SHA
Server public key is 2048 bit
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
SSL-Session:
    [...]
```

If secure renegotiation is not supported, the output will be slightly different:

```
Secure Renegotiation IS NOT supported
```

> **Note**
>
> Because TLS 1.3 doesn't support renegotiation, the `s_client` tool will always give a negative answer if this protocol version is negotiated. To ensure reliable results, use the `-no_tls1_3` switch to force negotiation of an earlier protocol version.

Even if the server indicates support for secure renegotiation, you may wish to test whether it also allows clients to initiate renegotiation. *Client-initiated renegotiation* is a protocol feature that doesn't serve any purpose in practice (because the server can always initiate renegotiation when it is needed) and makes the server more susceptible to denial of service attacks.

To initiate renegotiation, after the TLS handshake is complete, type an `R` character on a line by itself. For example, assuming we're talking to an HTTP server, you can type the first line of a request, initiate renegotiation, and then finish the request. Here's what that looks like when talking to a web server that supports client-initiated renegotiation:

```
GET / HTTP/1.0
R
RENEGOTIATING
depth=2 C = US, O = DigiCert Inc, OU = www.digicert.com, CN = DigiCert High ↩
Assurance EV Root CA
verify return:1
depth=1 C = US, O = DigiCert Inc, OU = www.digicert.com, CN = DigiCert SHA2 ↩
Extended Validation Server CA
verify return:1
depth=0 businessCategory = Private Organization, jurisdictionC = US, ↩
jurisdictionST = California, serialNumber = C2543436, C = US, ST = California, L = ↩
Mountain View, O = Mozilla Foundation, OU = Cloud Services, CN = addons.mozilla.org
verify return:1
Host: addons.mozilla.org

HTTP/1.1 301 Moved Permanently
Content-Type: text/plain; charset=utf-8
Date: Mon, 31 Aug 2020 12:40:49 GMT
Location: /en-US/firefox/
Strict-Transport-Security: max-age=31536000
Content-Length: 49
Connection: Close

Moved Permanently. Redirecting to /en-US/firefox/closed
```

When renegotiation is taking place, the server will send its certificates to the client again. You can see the verification of the certificate chain in the output. The next line after that continues with the `Host` request header. Seeing the web server's response is the proof that renegotiation is supported. Because of the various ways the renegotiation issue was addressed in various versions of SSL/TLS libraries, servers that do not support renegotiation may break the connection or may keep it open but refuse to continue to talk over it (which usually results in a timeout).

A server that does not support renegotiation will flatly refuse the second handshake on the connection:

```
HEAD / HTTP/1.0
```

```
R
RENEGOTIATING
140003560109728:error:1409E0E5:SSL routines:SSL3_WRITE_BYTES:ssl handshake ↵
failure:s3_pkt.c:592:
```

At the time of writing, the default behavior for OpenSSL is to connect to servers that don't support secure renegotiation; it will also accept both secure and insecure renegotiation, opting for whatever the server is able to do. If renegotiation is successful with a server that doesn't support secure renegotiation, you will know that the server supports insecure client-initiated renegotiation.

> **Note**
>
> The most reliable way to test for insecure renegotiation is to use the method described in this section, but with a version of OpenSSL that was released before the discovery of insecure renegotiation (e.g., 0.9.8k). I mention this because there is a small number of servers that support both secure and insecure renegotiation. This vulnerability is difficult to detect with modern versions of OpenSSL, which always prefer the secure option.

Testing for Heartbleed

You can test for Heartbleed manually with OpenSSL or by using one of the tools designed for this purpose. There are now many utilities available, because Heartbleed is very easy to exploit. But, as usual with such tools, there is a question of their accuracy. There is evidence that some tools fail to detect vulnerable servers.[8] Given the seriousness of Heartbleed, it's best to either test manually or by using a tool that gives you full visibility of the process. I am going to describe an approach you can use with only a modified version of OpenSSL.

Some parts of the test don't require modifications to OpenSSL, assuming you have a version that supports the Heartbeat protocol (starting with 1.0.1, but before 1.1.0). For example, to determine if the remote server supports the Heartbeat protocol, use the `-tlsextdebug` switch to display server extensions when connecting:

```
$ openssl s_client -connect www.feistyduck.com:443 -tlsextdebug
CONNECTED(00000003)
TLS server extension "renegotiation info" (id=65281), len=1
0001 - <SPACES/NULS>
TLS server extension "EC point formats" (id=11), len=4
0000 - 03 00 01 02                                        ....
TLS server extension "session ticket" (id=35), len=0
TLS server extension "heartbeat" (id=15), len=1
0000 - 01
```

[8] Bugs in Heartbleed detection scripts (Shannon Simpson and Adrian Hayter, 14 April 2014)

```
[...]
```

A server that does not return the heartbeat extension is not vulnerable to Heartbleed. To test if a server responds to heartbeat requests, use the `-msg` switch to request that protocol messages are shown, connect to the server, wait until the handshake completes, then type `B` and press return:

```
$ openssl s_client -connect www.feistyduck.com:443 -tlsextdebug -msg
[...]
---
B
HEARTBEATING
>>> TLS 1.2  [length 0025], HeartbeatRequest
    01 00 12 00 00 3c 83 1a 9f 1a 5c 84 aa 86 9e 20
    c7 a2 ac d7 6f f0 c9 63 9b d5 85 bf 9a 47 61 27
    d5 22 4c 70 75
<<< TLS 1.2  [length 0025], HeartbeatResponse
    02 00 12 00 00 3c 83 1a 9f 1a 5c 84 aa 86 9e 20
    c7 a2 ac d7 6f 52 4c ee b3 d8 a1 75 9a 6b bd 74
    f8 60 32 99 1c
read R BLOCK
```

This output shows a complete heartbeat request and response. The second and third bytes in both heartbeat messages specify payload length. We submitted a payload of 18 bytes (12 hexadecimal) and the server responded with a payload of the same size. In both cases there were also additional 16 bytes of padding. The first two bytes in the payload make the sequence number, which OpenSSL uses to match responses to requests. The remaining payload bytes and the padding are just random data.

To detect a vulnerable server, you'll have to prepare a special version of OpenSSL that sends an incorrect payload length. Vulnerable servers take the declared payload length and respond with that many bytes irrespective of the length of the actual payload provided.

At this point, you have to decide if you want to build an invasive test (which exploits the server by retrieving some data from the process) or a noninvasive test. This will depend on your circumstances. If you have permission for your testing activities, use the invasive test. With it, you'll be able to see exactly what is returned, and there won't be room for errors. For example, some versions of GnuTLS support Heartbeat and will respond to requests with incorrect payload length, but they will not actually return server data. A noninvasive test can't reliably diagnose that situation.

The following patch against OpenSSL 1.0.1h creates a noninvasive version of the test:

```
--- t1_lib.c.original   2014-07-04 17:29:35.092000000 +0100
+++ t1_lib.c    2014-07-04 17:31:44.528000000 +0100
@@ -2583,6 +2583,7 @@
 #endif
```

```
 #ifndef OPENSSL_NO_HEARTBEATS
+#define PAYLOAD_EXTRA 16
 int
 tls1_process_heartbeat(SSL *s)
        {
@@ -2646,7 +2647,7 @@
                 * sequence number */
                n2s(pl, seq);

-               if (payload == 18 && seq == s->tlsext_hb_seq)
+               if ((payload == (18 + PAYLOAD_EXTRA)) && seq == s->tlsext_hb_seq)
                        {
                        s->tlsext_hb_seq++;
                        s->tlsext_hb_pending = 0;
@@ -2705,7 +2706,7 @@
        /* Message Type */
        *p++ = TLS1_HB_REQUEST;
        /* Payload length (18 bytes here) */
-       s2n(payload, p);
+       s2n(payload + PAYLOAD_EXTRA, p);
        /* Sequence number */
        s2n(s->tlsext_hb_seq, p);
        /* 16 random bytes */
```

To build a noninvasive test, increase payload length by up to 16 bytes, or the length of the padding. When a vulnerable server responds to such a request, it will return the padding but nothing else. To build an invasive test, increase the payload length by, say, 32 bytes. A vulnerable server will respond with a payload of 50 bytes (18 bytes sent by OpenSSL by default, plus your 32 bytes) and send 16 bytes of padding. By increasing the declared length of the payload in this way, a vulnerable server will return up to 64 KB of data. A server not vulnerable to Heartbleed will not respond.

To produce your own Heartbleed testing tool, unpack a fresh copy of OpenSSL source code, edit `ssl/t1_lib.c` to make the change as in the patch, compile as usual, but don't install. The resulting `openssl` binary will be placed in the `apps/` subdirectory. Because it is statically compiled, you can rename it to something like `openssl-heartbleed` and move it to its permanent location.

Here's an example of the output you'd get with a vulnerable server that returns 16 bytes of server data (in bold):

```
B
HEARTBEATING
>>> TLS 1.2  [length 0025], HeartbeatRequest
    01 00 32 00 00 7c e8 f5 62 35 03 bb 00 34 19 4d
    57 7e f1 e5 90 6e 71 a9 26 85 96 1c c4 2b eb d5
```

```
    93 e2 d7 bb 5f
<<< TLS 1.2  [length 0045], HeartbeatResponse
    02 00 32 00 00 7c e8 f5 62 35 03 bb 00 34 19 4d
    57 7e f1 e5 90 6e 71 a9 26 85 96 1c c4 2b eb d5
    93 e2 d7 bb 5f 6f 81 0f aa dc e0 47 62 3f 7e dc
    60 95 c6 ba df c9 f6 9d 2b c8 66 f8 a5 45 64 0b
    d2 f5 3d a9 ad
read R BLOCK
```

If you want to see more data retrieved in a single response, increase the payload length, recompile, and test again. Alternatively, to retrieve another batch of the same size, use the `B` command again.

Determining the Strength of Diffie-Hellman Parameters

Starting from OpenSSL 1.0.2, when you connect to a server, the `s_client` command prints the strength of the ephemeral Diffie-Hellman key if one is used. Thus, to determine the strength of server's DH parameters, all you need to do is connect to it while offering only suites that use the DH key exchange. For example:

```
$ openssl s_client -connect www.feistyduck.com:443 -cipher kEDH
[...]
---s
No client certificate CA names sent
Peer signing digest: SHA512
Peer signature type: RSA
Server Temp Key: DH, 2048 bits
---
[...]
```

Servers that support export suites might actually offer even weaker DH parameters. To check for that possibility, connect using your old OpenSSL[9] while offering only export DHE suites:

```
$ openssl s_client -connect www.feistyduck.com:443 -cipher kEDH+EXPORT
```

This command should fail with well-configured servers. Otherwise, you'll probably see the server offering to negotiate insecure 512-bit DH parameters.

[9] Support for EXPORT cipher suites was removed in OpenSSL 1.1.0.