SECOND EDITION

# OPENSSL COOKBOOK

A Guide to the Most Frequently Used OpenSSL Features and Commands

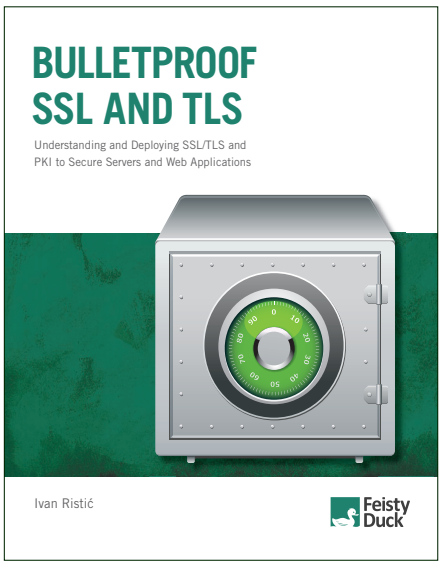

Ivan Ristić

QUALYS®
Feisty Duck

Last update: Thu Nov 10 04:26:32 GMT 2016 (build 551)

# BULLETPROOF SSL AND TLS

## Understanding and deploying SSL/TLS and PKI to secure your servers and web applications




# Available Now

**www.feistyduck.com**

For system administrators, developers, and IT security professionals, this book will teach you everything you need to know to protect your systems from eavesdropping and impersonation attacks.

***"The most comprehensive book about deploying TLS in the real world!"***

**Nasko Oskov**, Chrome Security developer and former SChannel developer

***"Meticulously researched."***

**Eric Lawrence**, Fiddler author and former Internet Explorer Program Manager

***"The most to the point and up to date book about SSL/TLS I've read."***

**Jakob Schlyter**, IT security advisor and DANE co-author

# OpenSSL Cookbook

Ivan Ristić

# OpenSSL Cookbook

by Ivan Ristić

Version 2.1-draft (build 551), published in November 2016.


First published in May 2013. Second edition published in March 2015.

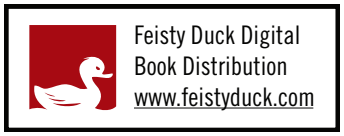
**Feisty Duck Limited**
*www.feistyduck.com*
*contact@feistyduck.com*

**Address:**
6 Acantha Court
Montpelier Road
London W5 2QP
United Kingdom

**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 SSL/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 virtually all of them rely on OpenSSL.

This book covers two ways in which OpenSSL can be used. Chapter 1, *OpenSSL*, will help users who need to perform routine tasks of key and certificate generation, and configure programs that rely on OpenSSL for SSL/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 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 andTLS*. I decided to publish the OpenSSL chapters as a separate free book because good documentation is always in great demand. This is particularly true for OpenSSL, which is not very well documented; what 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*.

## 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 couple of years, and in 2015 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*, which he publishes via Feisty Duck, his own platform for continuous writing and publishing. Ivan is an active participant in the security community, and you'll often find him speaking at security conferences such as Black Hat, RSA, OWASP AppSec, and others. He's currently Director of Application Security Research at Qualys.

# 1 OpenSSL

OpenSSL is an open source project that consists of a cryptographic library and an SSL/TLS toolkit. From the project's web site:

> *The OpenSSL Project is a collaborative effort to develop a robust, commercial-grade, full-featured, and Open Source toolkit implementing the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols as well as a full-strength general purpose cryptography library. The project is managed by a worldwide community of volunteers that use the Internet to communicate, plan, and develop the OpenSSL toolkit and its related documentation.*

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. The OpenSSL project was born in the last days of 1998, when Eric and Tim stopped their work on SSLeay to work on a commercial SSL/TLS toolkit called BSAFE SSL-C at RSA Australia.

Today, OpenSSL is ubiquitous on the server side and in many client tools. The command-line tools are also the most common choice for key and certificate management as well as testing. Interestingly, browsers have historically used other libraries, although that's now changing because Google is migrating Chrome to its own OpenSSL fork called *BoringSSL*.[2]

OpenSSL is dual-licensed under OpenSSL and SSLeay licenses. Both are BSD-like, with an advertising clause. The license has been a source of contention for a very long time, because neither of the licenses is considered compatible with the GPL family of licenses. For that reason, you will often find that GPL-licensed programs favor GnuTLS.

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

[2] BoringSSL (Chromium, retrieved 30 June 2015)

# Getting Started

If you're using one of the Unix platforms, getting started with OpenSSL is easy; you're virtually guaranteed to already have it on your system. The only problem that you might face is that you might not have the latest version. In this section, I assume that you're using a Unix platform, because that's the natural environment for OpenSSL.

Windows users tend to download binaries, which might complicate the situation slightly. In the simplest case, if you need OpenSSL only for its command-line utilities, the main OpenSSL web site links to Shining Light Productions[3] for the Windows binaries. In all other situations, you need to ensure that you're not mixing binaries compiled under different versions of OpenSSL. Otherwise, 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.[4]

## Determine OpenSSL Version and Configuration

Before you do any work, you should know which OpenSSL version you'll be using. For example, here's what I get for version information with `openssl version` on Ubuntu 12.04 LTS, which is the system that I'll be using for the examples in this chapter:

```
$ openssl version
OpenSSL 1.0.1 14 Mar 2012
```

At the time of this writing, a transition from OpenSSL 0.9.x to OpenSSL 1.0.x is in progress. The version 1.0.1 is especially significant because it is the first version to support TLS 1.1 and 1.2. The support for newer protocols is part of a global trend, so it's likely that we're going to experience a period during which interoperability issues are not uncommon.

> **Note**
>
> Various operating systems often modify the OpenSSL code, usually to fix known issues. However, the name of the project and the version number generally stay the same, and there is no indication that the code is actually a fork of the original project that will behave differently. For example, the version of OpenSSL used in Ubuntu 12.04 LTS[5] is based on OpenSSL 1.0.1c. At the time of this writing, the full name

[3] Win32 OpenSSL (Shining Light Productions, retrieved 3 July 2014)

[4] Apache 2.4 VC14 Binaries and Modules (Apache Lounge, retrieved 15 July 2015)

[5] "openssl" source package in Precise (Ubuntu, retrieved 3 July 2014)

> of the package is `openssl 1.0.1-4ubuntu5.16`, and it contains patches for the many issues that came to light over time.

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

```
$ openssl version -a
OpenSSL 1.0.1 14 Mar 2012
built on: Fri Jun 20 18:54:15 UTC 2014
platform: debian-amd64
options:  bn(64,64) rc4(8x,int) des(idx,cisc,16,int) blowfish(idx)
compiler: cc -fPIC -DOPENSSL_PIC -DZLIB -DOPENSSL_THREADS -D_REENTRANT -DDSO_DLFCN ↩
-DHAVE_DLFCN_H -m64 -DL_ENDIAN -DTERMIO -g -O2 -fstack-protector ↩
--param=ssp-buffer-size=4 -Wformat -Wformat-security -Werror=format-security -D↩
_FORTIFY_SOURCE=2 -Wl,-Bsymbolic-functions -Wl,-z,relro -Wa,--noexecstack -Wall ↩
-DOPENSSL_NO_TLS1_2_CLIENT -DOPENSSL_MAX_TLS1_2_CIPHER_LENGTH=50 -DMD32_REG_T=int ↩
-DOPENSSL_IA32_SSE2 -DOPENSSL_BN_ASM_MONT -DOPENSSL_BN_ASM_MONT5 -DOPENSSL_BN_ASM↩
_GF2m -DSHA1_ASM -DSHA256_ASM -DSHA512_ASM -DMD5_ASM -DAES_ASM -DVPAES_ASM -DBSAES↩
_ASM -DWHIRLPOOL_ASM -DGHASH_ASM
OPENSSLDIR: "/usr/lib/ssl"
```

The last line in the output (`/usr/lib/ssl`) is especially interesting because it will tell you where OpenSSL will look for its configuration and certificates. On my system, that location is essentially an alias for `/etc/ssl`, where Ubuntu keeps TLS-related files:

```
lrwxrwxrwx  1 root root   14 Apr 19 09:28 certs -> /etc/ssl/certs
drwxr-xr-x  2 root root 4096 May 28 06:04 misc
lrwxrwxrwx  1 root root   20 May 22 17:07 openssl.cnf -> /etc/ssl/openssl.cnf
lrwxrwxrwx  1 root root   16 Apr 19 09:28 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).

## Building OpenSSL

In most cases, you will be using the system–supplied version of OpenSSL, but sometimes there are good reasons to upgrade. For example, your system might be stuck with OpenSSL 0.9.x, which doesn't support newer TLS protocol versions. And even if the system OpenSSL is the right version, it might not have the features you need. For example, on Ubuntu 12.04 LTS, there's no support for SSL 2 in the `s_client` command. Although not supporting this version of SSL by default is the right decision, you'll need this feature if you're routinely testing other servers for SSL 2 support.

You can start by downloading the most recent version of OpenSSL (in my case, 1.0.1p):

```
$ wget http://www.openssl.org/source/openssl-1.0.1p.tar.gz
```

The next step is to configure OpenSSL before compilation. In most cases, you'll be leaving the system-provided version alone and installing OpenSSL in a different location. For example:

```
$ ./config \
--prefix=/opt/openssl \
--openssldir=/opt/openssl \
enable-ec_nistp_64_gcc_128
```

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.

You can then follow with:

```
$ make depend
$ make
$ 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. You can also build your own with little effort, as you'll see in the next section.

> **Note**
>
> When compiling software, it's important to be familiar with the default configuration of your compiler. System-provided packages are usually compiled using all the available hardening options, but if you compile some software yourself there is no guarantee that the same options will be used.[6]

[6] compiler hardening in Ubuntu and Debian (Kees Cook, 3 February 2014)

## Examine Available Commands

OpenSSL is a cryptographic toolkit that consists of many different utilities. I counted 46 in my version. If it were ever appropriate 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.

There isn't a specific help keyword, but help text is displayed whenever you type something OpenSSL does not recognize:

```
$ openssl help
openssl:Error: 'help' is an invalid command.

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

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.

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)
md4               md5               rmd160            sha
sha1
```

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       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        zlib
```

## Building a Trust Store

OpenSSL does not come with any trusted root certificates (also known as 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. This choice is usually fine, but default trust stores may not always be up to date. A better choice—but one that involves more work—is to turn to Mozilla, which is putting a lot of effort into maintaining a robust trust store. For example, this is what I did for my assessment tool on SSL Labs.

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

```
https://hg.mozilla.org/mozilla-central/raw-file/tip/security/nss/lib/ckfw/builtins↵
/certdata.txt
```

Unfortunately, their 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
```

But you don't have to write a conversion script if you'd rather download directly from Mozilla. Conversion scripts are available in Perl or Go. I describe both in the following sections.

> **Note**
>
> If you do end up working on your own conversion script, note that Mozilla's root certificate file actually contains two types of certificates: those that are trusted and are part of the store and also those that are explicitly distrusted. They use this mechanism to ban compromised intermediate CA certificates (e.g., DigiNotar's old certificates). Both conversion tools described here are smart enough to exclude distrusted certificates during the conversion process.

## Conversion Using Perl

The Curl project makes available a Perl script written by Guenter Knauf that can be used to convert Mozilla's trust store:

```
https://raw.github.com/bagder/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
Downloading 'certdata.txt' ...
Processing  'certdata.txt' ...
Done (156 CA certs processed, 19 untrusted skipped).
```

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

## Conversion Using Go

If you prefer the Go programming language, consider Adam Langley's conversion tool, which you can get from GitHub:

```
https://github.com/agl/extract-nss-root-certs
```

To kick off a conversion process, first download the tool itself:

```
$ wget https://raw.github.com/agl/extract-nss-root-certs/master/convert_mozilla↩
_certdata.go
```

Then download Mozilla's certificate data:

```
$ wget https://hg.mozilla.org/mozilla-central/raw-file/tip/security/nss/lib/ckfw↩
/builtins/certdata.txt --output-document certdata.txt
```

Finally, convert the file with the following command:

```
$ go run convert_mozilla_certdata.go > ca-certificates
2012/06/04 09:52:29 Failed to parse certificate starting on line 23068: negative ↩
serial number
```

In my case, there was one invalid certificate that the Go X.509 library couldn't handle, but otherwise the conversion worked as expected. Go versions from 1.6 onwards shouldn't produce this warning because they are able to handle certificates with negative serial numbers.

## 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 strong 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 for the use of public encryption is to generate a private key. Before you begin, you must make several decisions:

**Key algorithm**

OpenSSL supports RSA, DSA, and ECDSA keys, but not all types are practical for use in all scenarios. For example, for web server keys everyone uses RSA, because DSA keys are effectively limited to 1,024 bits (Internet Explorer doesn't support anything stronger) and ECDSA keys are yet to be widely supported by CAs. For SSH, DSA and RSA are widely used, whereas ECDSA might not be supported by all clients.

**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 is only 512 bits, which is simply 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 he or she could impersonate your web site. Today, 2,048-bit RSA keys are considered secure, and that's what you should use. Aim also to use 2,048 bits for DSA keys and at least 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.[7]

To generate an RSA key, use the `genrsa` command:

```
$ openssl genrsa -aes128 -out fd.key 2048
Generating RSA private key, 2048 bit long modulus
....+++
.....................................................................................↵
+++
e is 65537 (0x10001)
Enter pass phrase for fd.key: ****************
Verifying - Enter pass phrase for fd.key: ****************
```

Here, I specified that the key be protected with AES-128. You can also use AES-192 or AES-256 (switches `-aes192` and `-aes256`, respectively), but it's best to stay away from the other algorithms (DES, 3DES, and SEED).

> **Warning**
>
> The `e` value that you see in the output refers to the public exponent, which is set to 65,537 by default. This is what's known as a *short public exponent*, and it significantly improves the performance of RSA verification. Using the `-3` switch, you can choose 3 as your public exponent and make verification even faster. However, there are some unpleasant historical weaknesses associated with the use of 3 as a public exponent, which is why generally everyone recommends that you stick with 65,537. The latter choice provides a safety margin that's been proven effective in the past.

Private keys are stored in the so-called PEM format, which is just text:

```
$ cat fd.key
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: AES-128-CBC,01EC21976A463CE36E9DB59FF6AF689A

vERmFJzsLeAEDqWdXX4rNwogJp+y95uTnw+bOjWRw1+O1qgGqxQXPtH3LWDUz1Ym
mkpxmIwlSidVSUuUrrUzIL+V21EJ1W9iQ71SJoPOyzX7dYX5GCAwQm9Tsb4OFhV/
[21 lines removed...]
4phGTprEnEwrffRnYrt7khQwrJhNsw6TTtthMhx/UCJdpQdaLW/TuylaJMWL1JRW
i321s5me5ej6Pr4fGccNOe7lZK+563d7v5znAx+Wo1C+F7YgF+g8LOQ8emC+6AVV
```

[7] 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.

```
-----END RSA PRIVATE KEY-----
```

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 rsa -text -in fd.key
Enter pass phrase for fd.key: ****************
Private-Key: (2048 bit)
modulus:
    00:9e:57:1c:c1:0f:45:47:22:58:1c:cf:2c:14:db:
    [...]
publicExponent: 65537 (0x10001)
privateExponent:
    1a:12:ee:41:3c:6a:84:14:3b:be:42:bf:57:8f:dc:
    [...]
prime1:
    00:c9:7e:82:e4:74:69:20:ab:80:15:99:7d:5e:49:
    [...]
prime2:
    00:c9:2c:30:95:3e:cc:a4:07:88:33:32:a5:b1:d7:
    [...]
exponent1:
    68:f4:5e:07:d3:df:42:a6:32:84:8d:bb:f0:d6:36:
    [...]
exponent2:
    5e:b8:00:b3:f4:9a:93:cc:bc:13:27:10:9e:f8:7e:
    [...]
coefficient:
    34:28:cf:72:e5:3f:52:b2:dd:44:56:84:ac:19:00:
    [...]
writing RSA key
-----BEGIN RSA PRIVATE KEY-----
[...]
-----END RSA PRIVATE KEY-----
```

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

```
$ openssl rsa -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-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAnlccwQ9FRyJYHM8sFNsY
```

```
PUHJHJzhJdwcS7kBptutf/L6OvoEAzCVHi/mOqAA4QM5BziZgnvv+FNnE3sgE5pz
iovEHJ3C959mNQmpvnedXwfcOIlbrNqdISJiPOjs6mDCzYjSO1NCQoy3UpYwvwj7
OryR1F+abARehlts/Xs/PtX3VamrljiJN6JNgFICy3ZvEhLZEKxR7oob7TnyZDrj
IHxBbqPNzeiqLCFLFPGgJPaOcH8DdovBTesvu7wr/ecsf8CYyUCdEwGkZh9DKtdU
HFa9H8tWW2mX6uwYeHCnf2HTwOE8vjtOb8oYQxlQxtL7dpFyMgrpPOoOVkZZW/PO
NQIDAQAB
-----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.

DSA key generation is a two-step process: DSA parameters are created in the first step and the key in the second. Rather than execute the steps one at a time, I tend to use the following two commands as one:

```
$ openssl dsaparam -genkey 2048 | openssl dsa -out dsa.key -aes128
Generating DSA parameters, 2048 bit long prime
This could take some time
[...]
read DSA key
writing DSA key
Enter PEM pass phrase: ****************
Verifying - Enter PEM pass phrase: ****************
```

This approach allows me to generate a password-protected key without leaving any temporary files (DSA parameters) and/or temporary keys on disk.

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 `secp256r1` named curve:

```
$ openssl ecparam -genkey -name secp256r1 | openssl ec -out ec.key -aes128
using curve name prime256v1 instead of secp256r1
read EC key
writing EC key
Enter PEM pass phrase: ****************
Verifying - Enter PEM pass phrase: ****************
```

OpenSSL supports many named curves (you can get a full list with the `-list_curves` switch), but, for web server keys, you're limited to only two curves that are supported by all major browsers: `secp256r1` (OpenSSL uses the name `prime256v1`) and `secp384r1`.

> **Note**
>
> If you're using OpenSSL 1.0.2, you can save yourself time by always generating your keys using the `genpkey` command, which has been improved to support various key types and configuration parameters. It now represents a unified interface for key generation.

## 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 []:webmaster@feistyduck.com

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,[8] *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 them 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: 0 (0x0)
        Subject: C=GB, L=London, O=Feisty Duck Ltd, CN=www.feistyduck.com↵
/emailAddress=webmaster@feistyduck.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:b7:fc:ca:1c:a6:c8:56:bb:a3:26:d1:df:e4:e3:
                    [16 more lines...]
                    d1:57
                Exponent: 65537 (0x10001)
        Attributes:
            a0:00
    Signature Algorithm: sha1WithRSAEncryption
         a7:43:56:b2:cf:ed:c7:24:3e:36:0f:6b:88:e9:49:03:a6:91:
         [13 more lines...]
         47:8b:e3:28
```

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

## 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.

## 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 (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 installing a TLS server for your own use, you probably don't want to go to a CA for a publicly trusted certificate. It's much easier to just use a self-signed certificate. If you're a Firefox user, on your first visit to the web site you can create a certificate exception, after which the site will be as secure as if it were protected with a publicly trusted certificate.

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=/CN=www.feistyduck.com/emailAddress=webmaster@feistyduck.com/O=Feisty Duck ↩
Ltd/L=London/C=GB
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 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, you'll find that it contains the SAN extension:

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

## Examining Certificates

Certificates might look a lot like random data at first glance, but they contain a great deal of information; you just need to know how to unpack it. The `x509` command does just that, so 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: 1 (0x0)
        Serial Number: 13073330765974645413 (0xb56dcd10f11aaaa5)
    Signature Algorithm: sha1WithRSAEncryption
        Issuer: CN=www.feistyduck.com/emailAddress=webmaster@feistyduck.com, ↩
O=Feisty Duck Ltd, L=London, C=GB
        Validity
            Not Before: Jun  4 17:57:34 2012 GMT
```

```
            Not After : Jun  4 17:57:34 2013 GMT
        Subject: CN=www.feistyduck.com/emailAddress=webmaster@feistyduck.com, ↩
O=Feisty Duck Ltd, L=London, C=GB
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:b7:fc:ca:1c:a6:c8:56:bb:a3:26:d1:df:e4:e3:
                    [16 more lines...]
                    d1:57
                Exponent: 65537 (0x10001)
    Signature Algorithm: sha1WithRSAEncryption
         49:70:70:41:6a:03:0f:88:1a:14:69:24:03:6a:49:10:83:20:
         [13 more lines...]
         74:a1:11:86
```

Self-signed certificates usually contain only the most basic certificate data, as seen in the previous example. 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). Let's go over them quickly.

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).

```
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, browsers 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 *extended validation* (EV) indicators can be found (as in the example that follows). The indicators are in the form of unique object identifiers (OIDs), and they are unique to the issuing CA. 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: 2.16.840.1.114414.1.7.23.3
    CPS: http://certificates.starfieldtech.com/repository/
```

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 (e.g., Internet Explorer) 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.starfieldtech.com/
    CA Issuers - URI:http://certificates.starfieldtech.com/repository/sf↩
_intermediate.crt
```

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:
    4A:AB:1C:C3:D3:4E:F7:5B:2B:59:71:AA:20:63:D6:C9:40:FB:14:F1
X509v3 Authority Key Identifier:
    keyid:49:4B:52:27:D1:1B:BC:F2:A1:21:6A:62:7B:51:42:7A:8A:D7:D5:56
```

Finally, 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
```

## 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:

**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.

**Binary (DER) key**
: Contains a private key in its raw form, using DER ASN.1 encoding. OpenSSL creates keys in its own traditional (SSLeay) format. There's also an alternative format called PKCS#8 (defined in RFC 5208), but it's not widely used. OpenSSL can convert to and from PKCS#8 format using the `pkcs8` command.

**ASCII (PEM) key**
: Contains a base64-encoded DER key, sometimes with additional metadata (e.g., the algorithm used for password protection).

**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#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.

## 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

In this section, I discuss two topics relevant for TLS deployment. The first is cipher suite configuration, in which you specify which of the many suites available in TLS you wish to use for communication. This topic is important because virtually every program that uses OpenSSL reuses its suite configuration mechanism. That means that once you learn how to configure cipher suites for one program, you can reuse the same knowledge elsewhere. The second topic is the performance measurement of raw crypto operations.

## Cipher Suite Selection

A common task in TLS server configuration is selecting which cipher suites are going to be supported. Programs that rely on OpenSSL usually adopt the same approach to suite configuration as OpenSSL does, simply passing through the configuration options. For example, in Apache `httpd`, the cipher suite configuration may look like this:

```
SSLHonorCipherOrder On
SSLCipherSuite "HIGH:!aNULL:@STRENGTH"
```

The first line controls cipher suite prioritization (and configures `httpd` to actively select suites). The second line controls which suites will be supported.

Coming up with a good suite configuration can be pretty time consuming, and there are a lot of details to consider. The best approach is to use the OpenSSL `ciphers` command to determine which suites are enabled with a particular configuration string.

### Obtaining the List of Supported Suites

Before you do anything else, you should determine which suites are supported by your OpenSSL installation. To do this, invoke the `ciphers` command with the switch `-v` and the parameter `ALL:COMPLEMENTOFALL` (clearly, `ALL` does not actually mean "all"):

```
$ openssl ciphers -v 'ALL:COMPLEMENTOFALL'
ECDHE-RSA-AES256-GCM-SHA384    TLSv1.2 Kx=ECDH Au=RSA   Enc=AESGCM(256) Mac=AEAD
```

```
ECDHE-ECDSA-AES256-GCM-SHA384  TLSv1.2 Kx=ECDH Au=ECDSA Enc=AESGCM(256) Mac=AEAD
ECDHE-RSA-AES256-SHA384        TLSv1.2 Kx=ECDH Au=RSA   Enc=AES(256)    Mac=SHA384
ECDHE-ECDSA-AES256-SHA384      TLSv1.2 Kx=ECDH Au=ECDSA Enc=AES(256)    Mac=SHA384
ECDHE-RSA-AES256-SHA           SSLv3   Kx=ECDH Au=RSA   Enc=AES(256)    Mac=SHA1
[106 more lines...]
```

> **Tip**
>
> If you're using OpenSSL 1.0.0 or later, you can also use the uppercase `-V` switch to request extra-verbose output. In this mode, the output will also contain suite IDs, which are always handy to have. For example, OpenSSL does not always use the RFC names for the suites; in such cases, you must use the IDs to cross-check.

In my case, there were 111 suites in the output. Each line contains information on one suite and the following information:

1. Suite name
2. Required minimum protocol version
3. Key exchange algorithm
4. Authentication algorithm
5. Cipher algorithm and strength
6. MAC (integrity) algorithm
7. Export suite indicator

If you change the `ciphers` parameter to something other than `ALL:COMPLEMENTOFALL`, OpenSSL will list only the suites that match that configuration. For example, you can ask it to list only cipher suites that are based on RC4, as follows:

```
$ openssl ciphers -v 'RC4'
ECDHE-RSA-RC4-SHA    SSLv3 Kx=ECDH       Au=RSA   Enc=RC4(128) Mac=SHA1
ECDHE-ECDSA-RC4-SHA  SSLv3 Kx=ECDH       Au=ECDSA Enc=RC4(128) Mac=SHA1
AECDH-RC4-SHA        SSLv3 Kx=ECDH       Au=None  Enc=RC4(128) Mac=SHA1
ADH-RC4-MD5          SSLv3 Kx=DH         Au=None  Enc=RC4(128) Mac=MD5
ECDH-RSA-RC4-SHA     SSLv3 Kx=ECDH/RSA   Au=ECDH  Enc=RC4(128) Mac=SHA1
ECDH-ECDSA-RC4-SHA   SSLv3 Kx=ECDH/ECDSA Au=ECDH  Enc=RC4(128) Mac=SHA1
RC4-SHA              SSLv3 Kx=RSA        Au=RSA   Enc=RC4(128) Mac=SHA1
RC4-MD5              SSLv3 Kx=RSA        Au=RSA   Enc=RC4(128) Mac=MD5
PSK-RC4-SHA          SSLv3 Kx=PSK        Au=PSK   Enc=RC4(128) Mac=SHA1
EXP-ADH-RC4-MD5      SSLv3 Kx=DH(512)    Au=None  Enc=RC4(40)  Mac=MD5  export
EXP-RC4-MD5          SSLv3 Kx=RSA(512)   Au=RSA   Enc=RC4(40)  Mac=MD5  export
```

The output will contain all suites that match your requirements, even if they're insecure. Clearly, you should choose your configuration strings carefully in order to activate only what's

secure. Further, the order in which suites appear in the output matters. When you configure your TLS server to actively select the cipher suite that will be used for a connection (which is the best practice and should always be done), the suites listed first are given priority.

## 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. All other keywords select groups of suites according to some criteria. Keyword names are case-sensitive. Normally, I might direct you to the OpenSSL documentation for a comprehensive list of keywords, but it turns out that the ciphers documentation is not up to date; it's missing some more recent additions. For that reason, I'll try to document all the keywords in this section.

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

**Table 1.1. Group keywords**

| Keyword | Meaning |
|---|---|
| `DEFAULT` | The default cipher list. This is determined at compile time and, as of OpenSSL 1.0.0, is normally `ALL:!aNULL:!eNULL`. This must be the first cipher string specified. |
| `COMPLEMENTOFDEFAULT` | The ciphers included in `ALL`, but not enabled by default. Currently, this is `ADH`. 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. **Insecure.** |
| `EXP, EXPORT` | Export encryption algorithms. Including 40- and 56-bit algorithms. **Insecure.** |
| `EXPORT40` | 40-bit export encryption algorithms. **Insecure.** |
| `EXPORT56` | 56-bit export encryption algorithms. **Insecure.** |
| `TLSv1, SSLv3, SSLv2` | TLS 1.0, SSL 3, or SSL 2 cipher suites, respectively. |

Digest keywords select suites that use a particular digest algorithm. For example, `MD5` selects all suites that rely on MD5 for integrity validation.

**Table 1.2. Digest algorithm keywords**

| Keyword | Meaning |
|---|---|
| MD5 | Cipher suites using MD5. **Obsolete and insecure.** |
| SHA, SHA1 | Cipher suites using SHA1. |
| SHA256 (v1.0.0+) | Cipher suites using SHA256. |
| SHA384 (v1.0.0+) | 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, virtually all public certificates use RSA for authentication. Over time, we will probably see a very slow rise in the use of Elliptic Curve (ECDSA) certificates.

**Table 1.3. Authentication keywords**

| Keyword | Meaning |
|---|---|
| aDH | Cipher suites effectively using DH authentication, i.e., the certificates carry DH keys. (v1.0.2+) |
| aDSS, DSS | Cipher suites using DSS authentication, i.e., the certificates carry DSS keys. |
| aECDH (v1.0.0+) | Cipher suites that use ECDH authentication. |
| aECDSA (v1.0.0+) | 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. |
| PSK | Cipher suites using PSK (Pre-Shared Key) authentication. |
| SRP | 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`). To make things worse, some older suites do have `E` at the beginning of the key exchange algorithm (e.g., `EDH-RSA-DES-CBC-SHA`).

**Table 1.4. Key exchange keywords**

| Keyword | Meaning |
|---|---|
| ADH | Anonymous DH cipher suites. **Insecure.** |
| AECDH (v1.0.0+) | Anonymous ECDH cipher suites. **Insecure.** |
| DH | Cipher suites using DH (includes ephemeral and anonymous DH). |
| ECDH (v1.0.0+) | Cipher suites using ECDH (includes ephemeral and anonymous ECDH). |
| EDH (v1.0.0+) | Cipher suites using ephemeral DH key agreement. |
| EECDH (v1.0.0+) | Cipher suites using ephemeral ECDH. |
| kECDH (v1.0.0+) | Cipher suites using ECDH key agreement. |
| kEDH | Cipher suites using ephemeral DH key agreements (includes anonymous DH). |
| kEECDH (v1.0.0+) | Cipher suites using ephemeral ECDH key agreement (includes anonymous ECDH). |
| kRSA, RSA | Cipher suites using RSA key exchange. |

Cipher keywords select suites based on the cipher they use.

**Table 1.5. Cipher keywords**

| Keyword | Meaning |
|---|---|
| 3DES | Cipher suites using triple DES. |
| AES | Cipher suites using AES. |
| AESGCM (v1.0.0+) | Cipher suites using AES GCM. |
| CAMELLIA | Cipher suites using Camellia. |
| DES | Cipher suites using single DES. **Obsolete and insecure.** |
| eNULL, NULL | Cipher suites that don't use encryption. **Insecure.** |
| IDEA | Cipher suites using IDEA. |
| RC2 | Cipher suites using RC2. **Obsolete and insecure.** |
| RC4 | Cipher suites using RC4. **Insecure.** |
| SEED | Cipher suites using SEED. |

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.

**Table 1.6. Miscellaneous keywords**

| Keyword | Meaning |
|---|---|
| `@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. |

## 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 RC4 and SHA:

```
$ openssl ciphers -v 'RC4+SHA'
ECDHE-RSA-RC4-SHA    SSLv3 Kx=ECDH       Au=RSA   Enc=RC4(128) Mac=SHA1
ECDHE-ECDSA-RC4-SHA  SSLv3 Kx=ECDH       Au=ECDSA Enc=RC4(128) Mac=SHA1
AECDH-RC4-SHA        SSLv3 Kx=ECDH       Au=None  Enc=RC4(128) Mac=SHA1
ECDH-RSA-RC4-SHA     SSLv3 Kx=ECDH/RSA   Au=ECDH  Enc=RC4(128) Mac=SHA1
ECDH-ECDSA-RC4-SHA   SSLv3 Kx=ECDH/ECDSA Au=ECDH  Enc=RC4(128) Mac=SHA1
RC4-SHA              SSLv3 Kx=RSA        Au=RSA   Enc=RC4(128) Mac=SHA1
PSK-RC4-SHA          SSLv3 Kx=PSK        Au=PSK   Enc=RC4(128) Mac=SHA1
```

## 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. For example, to choose all suites that use RC4 and AES ciphers:

```
$ openssl ciphers -v 'RC4:AES'
```

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 'RC4 AES'
```

## 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 to specify 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. Works only on existing suites; 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 `RC4:+MD5` enables all RC4 suites, but pushes the MD5-based ones to the end.

### Sorting

The `@STRENGTH` keyword is unlike other keywords (I assume that's why it has the `@` in the name): 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.

Take, for example, the following cipher suite configuration:

```
$ openssl ciphers -v 'DES-CBC-SHA DES-CBC3-SHA RC4-SHA AES256-SHA @STRENGTH'
AES256-SHA                  SSLv3   Kx=RSA  Au=RSA   Enc=AES(256)    Mac=SHA1
DES-CBC3-SHA                SSLv3   Kx=RSA  Au=RSA   Enc=3DES(168)   Mac=SHA1
RC4-SHA                     SSLv3   Kx=RSA  Au=RSA   Enc=RC4(128)    Mac=SHA1
DES-CBC-SHA                 SSLv3   Kx=RSA  Au=RSA   Enc=DES(56)     Mac=SHA1
```

In theory, the output is sorted in order of strength. In practice, you'll often want better control of the suite order:

- For example, `AES256-SHA` (a CBC suite) is vulnerable to the BEAST attack when used with TLS 1.0 and earlier protocols. If you want to mitigate the BEAST attack server-side, you'll prefer to prioritize the `RC4-SHA` suite, which isn't vulnerable to this problem.
- 3DES is only nominally rated at 168 bits; a so-called *meet-in-the-middle* attack reduces its strength to 112 bits,[9] and further issues make the strength as low as 108 bits.[10] This fact makes `DES-CBC3-SHA` inferior to 128-bit cipher suites. Strictly speaking, treating 3DES as a 168-bit cipher is a bug in OpenSSL that has been fixed in the more recent releases.

## 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:
```

[9] Cryptography/Meet In The Middle Attack (Wikibooks, retrieved 31 March 2014)

[10] Attacking Triple Encryption (Stefan Lucks, 1998)

## Putting It All Together

To demonstrate how various cipher suite configuration features come together, I will present one complete real-life use case. Please bear in mind that what follows is just an example. Because there are usually many aspects to consider when deciding on the configuration, there isn't such a thing as a single perfect configuration.

For that reason, before you can start to work on your configuration, you should have a clear idea of what you wish to achieve. In my case, I wish to have a reasonably secure and efficient configuration, which I define to mean the following:

1. Use only strong ciphers of 128 effective bits and up (this excludes 3DES).
2. Use only suites that provide strong authentication (this excludes anonymous and export suites).
3. Do not use any suites that rely on weak primitives (e.g., MD5).
4. Implement robust support for forward secrecy, no matter what keys and protocols are used. With this requirement comes a slight performance penalty, because I won't be able to use the fast RSA key exchange. I'll minimize the penalty by prioritizing ECDHE, which is substantially faster than DHE.
5. Prefer ECDSA over RSA. This requirement makes sense only in dual-key deployments, in which we want to use the faster ECDSA operations wherever possible, but fall back to RSA when talking to clients that do not yet support ECDSA.
6. With TLS 1.2 clients, prefer AES GCM suites, which provide the best security TLS can offer.
7. Because RC4 was recently found to be weaker than previously thought,[11] we want to push it to the end of the list. That's almost as good as disabling it. Although BEAST might still be a problem in some situations, I'll assume that it's been mitigated client-side.

Usually the best approach is to start by permanently eliminating all the components and suites that you don't wish to use; this reduces clutter and ensures that the undesired suites aren't introduced back into the configuration by mistake.

The weak suites can be identified with the following cipher strings:

- `aNULL`; no authentication

[11] On the Security of RC4 in TLS and WPA (AlFardan et al., 13 March 2013)

- `eNULL`; no encryption
- `LOW`; low-strength suites
- `3DES`; effective strength of 108 bits
- `MD5`; suites that use MD5
- `EXP`; obsolete export suites

To reduce the number of suites displayed, I'm going to eliminate all DSA, PSK, SRP, and ECDH suites, because they're used only very rarely. I am also removing the IDEA and SEED ciphers, which are obsolete but might still be supported by OpenSSL. In my configuration, I won't use CAMELLIA either, because it's slower and not as well supported as AES (e.g., no GCM or ECDHE variants in practice).

```
!aNULL !eNULL !LOW !3DES !MD5 !EXP !DSS !PSK !SRP !kECDH !CAMELLIA !IDEA !SEED
```

Now we can focus on what we want to achieve. Because forward secrecy is our priority, we can start with the `kEECDH` and `kEDH` keywords:

```
kEECDH kEDH !aNULL !eNULL !LOW !3DES !MD5 !EXP !DSS !PSK !SRP !kECDH !CAMELLIA ↩
!IDEA !SEED
```

If you test this configuration, you'll find that RSA suites are listed first, but I said I wanted ECDSA first:

```
ECDHE-RSA-AES256-GCM-SHA384    TLSv1.2 Kx=ECDH Au=RSA   Enc=AESGCM(256) Mac=AEAD
ECDHE-ECDSA-AES256-GCM-SHA384  TLSv1.2 Kx=ECDH Au=ECDSA Enc=AESGCM(256) Mac=AEAD
ECDHE-RSA-AES256-SHA384        TLSv1.2 Kx=ECDH Au=RSA   Enc=AES(256)    Mac=SHA384
ECDHE-ECDSA-AES256-SHA384      TLSv1.2 Kx=ECDH Au=ECDSA Enc=AES(256)    Mac=SHA384
ECDHE-RSA-AES256-SHA           SSLv3   Kx=ECDH Au=RSA   Enc=AES(256)    Mac=SHA1
ECDHE-ECDSA-AES256-SHA         SSLv3   Kx=ECDH Au=ECDSA Enc=AES(256)    Mac=SHA1
ECDHE-RSA-AES128-GCM-SHA256    TLSv1.2 Kx=ECDH Au=RSA   Enc=AESGCM(128) Mac=AEAD
[...]
```

In order to fix this, I'll put ECDSA suites first, by placing `kEECDH+ECDSA` at the beginning of the configuration:

```
kEECDH+ECDSA kEECDH kEDH !aNULL !eNULL !LOW !3DES !MD5 !EXP !DSS !PSK !SRP !kECDH ↩
!CAMELLIA !IDEA !SEED
```

The next problem is that older suites (SSL 3) are mixed with newer suites (TLS 1.2). In order to maximize security, I want all TLS 1.2 clients to always negotiate TLS 1.2 suites. To push older suites to the end of the list, I'll use the `+SHA` keyword (TLS 1.2 suites are all using either SHA256 or SHA384, so they won't match):

```
kEECDH+ECDSA kEECDH kEDH +SHA !aNULL !eNULL !LOW !3DES !MD5 !EXP !DSS !PSK !SRP ↩
!kECDH !CAMELLIA !IDEA !SEED
```

At this point, I'm mostly done. I only need to add the remaining secure suites to the end of the list; the `HIGH` keyword will achieve this. In addition, I'm also going to make sure RC4 suites are last, using `+RC4` (to push existing RC4 suites to the end of the list) and `RC4` (to add to the list any remaining RC4 suites that are not already on it):

```
kEECDH+ECDSA kEECDH kEDH HIGH +SHA +RC4 RC4 !aNULL !eNULL !LOW !3DES !MD5 !EXP ↩
!DSS !PSK !SRP !kECDH !CAMELLIA !IDEA !SEED
```

Let's examine the entire final output, which consists of 28 suites. In the first group are the TLS 1.2 suites:

```
ECDHE-ECDSA-AES256-GCM-SHA384  TLSv1.2 Kx=ECDH Au=ECDSA Enc=AESGCM(256) Mac=AEAD
ECDHE-ECDSA-AES256-SHA384      TLSv1.2 Kx=ECDH Au=ECDSA Enc=AES(256)    Mac=SHA384
ECDHE-ECDSA-AES128-GCM-SHA256  TLSv1.2 Kx=ECDH Au=ECDSA Enc=AESGCM(128) Mac=AEAD
ECDHE-ECDSA-AES128-SHA256      TLSv1.2 Kx=ECDH Au=ECDSA Enc=AES(128)    Mac=SHA256
ECDHE-RSA-AES256-GCM-SHA384    TLSv1.2 Kx=ECDH Au=RSA   Enc=AESGCM(256) Mac=AEAD
ECDHE-RSA-AES256-SHA384        TLSv1.2 Kx=ECDH Au=RSA   Enc=AES(256)    Mac=SHA384
ECDHE-RSA-AES128-GCM-SHA256    TLSv1.2 Kx=ECDH Au=RSA   Enc=AESGCM(128) Mac=AEAD
ECDHE-RSA-AES128-SHA256        TLSv1.2 Kx=ECDH Au=RSA   Enc=AES(128)    Mac=SHA256
DHE-RSA-AES256-GCM-SHA384      TLSv1.2 Kx=DH   Au=RSA   Enc=AESGCM(256) Mac=AEAD
DHE-RSA-AES256-SHA256          TLSv1.2 Kx=DH   Au=RSA   Enc=AES(256)    Mac=SHA256
DHE-RSA-AES128-GCM-SHA256      TLSv1.2 Kx=DH   Au=RSA   Enc=AESGCM(128) Mac=AEAD
DHE-RSA-AES128-SHA256          TLSv1.2 Kx=DH   Au=RSA   Enc=AES(128)    Mac=SHA256
AES256-GCM-SHA384              TLSv1.2 Kx=RSA  Au=RSA   Enc=AESGCM(256) Mac=AEAD
AES256-SHA256                  TLSv1.2 Kx=RSA  Au=RSA   Enc=AES(256)    Mac=SHA256
AES128-GCM-SHA256              TLSv1.2 Kx=RSA  Au=RSA   Enc=AESGCM(128) Mac=AEAD
AES128-SHA256                  TLSv1.2 Kx=RSA  Au=RSA   Enc=AES(128)    Mac=SHA256
```

ECDHE suites are first, followed by DHE suites, followed by all other TLS 1.2 suites. Within each group, ECDSA and GCM have priority.

In the second group are the suites that are going to be used by TLS 1.0 clients, using similar priorities as in the first group:

```
ECDHE-ECDSA-AES256-SHA         SSLv3   Kx=ECDH Au=ECDSA Enc=AES(256)    Mac=SHA1
ECDHE-ECDSA-AES128-SHA         SSLv3   Kx=ECDH Au=ECDSA Enc=AES(128)    Mac=SHA1
ECDHE-RSA-AES256-SHA           SSLv3   Kx=ECDH Au=RSA   Enc=AES(256)    Mac=SHA1
ECDHE-RSA-AES128-SHA           SSLv3   Kx=ECDH Au=RSA   Enc=AES(128)    Mac=SHA1
DHE-RSA-AES256-SHA             SSLv3   Kx=DH   Au=RSA   Enc=AES(256)    Mac=SHA1
DHE-RSA-AES128-SHA             SSLv3   Kx=DH   Au=RSA   Enc=AES(128)    Mac=SHA1
DHE-RSA-SEED-SHA               SSLv3   Kx=DH   Au=RSA   Enc=SEED(128  ) Mac=SHA1
AES256-SHA                     SSLv3   Kx=RSA  Au=RSA   Enc=AES(256)    Mac=SHA1
AES128-SHA                     SSLv3   Kx=RSA  Au=RSA   Enc=AES(128)    Mac=SHA1
```

Finally, the RC4 suites are at the end:

```
ECDHE-ECDSA-RC4-SHA           SSLv3   Kx=ECDH Au=ECDSA Enc=RC4(128)    Mac=SHA1
ECDHE-RSA-RC4-SHA             SSLv3   Kx=ECDH Au=RSA   Enc=RC4(128)    Mac=SHA1
RC4-SHA                       SSLv3   Kx=RSA  Au=RSA   Enc=RC4(128)    Mac=SHA1
```

## Recommended Configuration

The configuration in the previous section was designed to use as an example of cipher suite configuration using OpenSSL suite keywords, but it's not the best setup you could have. In fact, there isn't any one configuration that will satisfy everyone. In this section, I'll give you several configurations to choose from based on your preferences and risk assessment.

The design principles for all configurations here are essentially the same as those from the previous section, but I am going to make two changes to achieve better performance. First, I am going to put 128-bit suites on top of the list. Although 256-bit suites provide some increase in security, for most sites the increase is not meaningful and yet still comes with the performance penalty. Second, I am going to prefer HMAC-SHA over HMAC-SHA256 and HMAC-SHA384 suites. The latter two are much slower but also don't provide a meaningful increase in security.

In addition, I am going to change my approach from configuring suites using keywords to using suite names directly. I think that keywords, conceptually, are not a bad idea: you specify your security requirements and the library does the rest, without you having to know a lot about the suites that are going to be used. Unfortunately, this approach no longer works well in practice, as we've become quite picky about what suites we wish to have enabled and in what order.

Using suite names in a configuration is also easier: you just list the suites you want to use. And, when you're looking at someone's configuration, you now know exactly what suites are used without having to run the settings through OpenSSL.

The following is my default starting configuration, designed to offer strong security as well as good performance:

```
ECDHE-ECDSA-AES128-GCM-SHA256
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-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-AES256-GCM-SHA384
DHE-RSA-AES128-SHA
DHE-RSA-AES256-SHA
DHE-RSA-AES128-SHA256
DHE-RSA-AES256-SHA256
EDH-RSA-DES-CBC3-SHA
```

This configuration uses only suites that support forward secrecy and provide strong encryption. Most modern browsers and other clients will be able to connect, but some very old clients might not. As an example, older Internet Explorer versions running on Windows XP will fail.

If you really need to provide support for a very old range of clients—and only then—consider adding the following suites to the end of the list:

```
AES128-SHA
AES256-SHA
DES-CBC3-SHA
ECDHE-RSA-RC4-SHA
RC4-SHA
```

Most of these legacy suites use the RSA key exchange, which means that they don't provide forward secrecy. The AES cipher is preferred, but 3DES and (the insecure) RC4 are also supported for maximum compatibility with as many clients as possible. If the use of RC4 can't be avoided, the preference is to use the ECDHE suite that provides forward secrecy.

## 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 RC4, AES, RSA, ECDH, and SHA algorithms:

```
$ openssl speed rc4 aes rsa ecdh sha
```

There are three relevant parts to the output. The first part consists of the OpenSSL version number and compile-time configuration. This information is useful if you're testing several different versions of OpenSSL with varying compile-time options:

```
OpenSSL 0.9.8k 25 Mar 2009
built on: Wed May 23 00:02:00 UTC 2012
options:bn(64,64) md2(int) rc4(ptr,char) des(idx,cisc,16,int) aes(partial) ↩
blowfish(ptr2)
compiler: cc -fPIC -DOPENSSL_PIC -DZLIB -DOPENSSL_THREADS -D_REENTRANT -DDSO_DLFCN ↩
-DHAVE_DLFCN_H -m64 -DL_ENDIAN -DTERMIO -O3 -Wa,--noexecstack -g -Wall -DMD32_REG↩
_T=int -DOPENSSL_BN_ASM_MONT -DSHA1_ASM -DSHA256_ASM -DSHA512_ASM -DMD5_ASM -DAES↩
_ASM
available timing options: TIMES TIMEB HZ=100 [sysconf value]
timing function used: times
The 'numbers' are in 1000s of bytes per second processed.
```

The second part contains symmetric cryptography benchmarks (i.e., hash functions and private cryptography):

```
type             16 bytes     64 bytes    256 bytes   1024 bytes   8192 bytes
sha1             29275.44k    85281.86k   192290.28k   280526.68k   327553.12k
rc4             160087.81k   172435.03k   174264.75k   176521.50k   176700.62k
aes-128 cbc      90345.06k   140108.84k   170027.92k   179704.12k   182388.44k
aes-192 cbc     104770.95k   134601.12k   148900.05k   152662.30k   153941.11k
aes-256 cbc      95868.62k   116430.41k   124498.19k   127007.85k   127430.81k
sha256           23354.37k    54220.61k    99784.35k   126494.48k   138266.71k
sha512           16022.98k    64657.88k   113304.06k   178301.77k   214539.99k
```

Finally, the third part contains the asymmetric (public) cryptography benchmarks:

```
                  sign    verify    sign/s verify/s
rsa  512 bits 0.000120s 0.000011s   8324.9  90730.0
rsa 1024 bits 0.000569s 0.000031s   1757.0  31897.1
rsa 2048 bits 0.003606s 0.000102s    277.3   9762.0
rsa 4096 bits 0.024072s 0.000376s     41.5   2657.4
                              op      op/s
 160 bit ecdh (secp160r1)   0.0003s   2890.2
 192 bit ecdh (nistp192)    0.0006s   1702.9
 224 bit ecdh (nistp224)    0.0006s   1743.5
 256 bit ecdh (nistp256)    0.0007s   1513.3
 384 bit ecdh (nistp384)    0.0015s    689.6
 521 bit ecdh (nistp521)    0.0029s    340.3
 163 bit ecdh (nistk163)    0.0009s   1126.2
 233 bit ecdh (nistk233)    0.0012s    818.5
 283 bit ecdh (nistk283)    0.0028s    360.2
 409 bit ecdh (nistk409)    0.0060s    166.3
 571 bit ecdh (nistk571)    0.0130s     76.8
```

```
 163 bit ecdh (nistb163)   0.0009s   1061.3
 233 bit ecdh (nistb233)   0.0013s    755.2
 283 bit ecdh (nistb283)   0.0030s    329.4
 409 bit ecdh (nistb409)   0.0067s    149.7
 571 bit ecdh (nistb571)   0.0146s     68.4
```

What's this output 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. For example, the previous results are from a real-life server that's using the OpenSSL 0.9.8k (patched by the distribution vendor). I'm considering moving to OpenSSL 1.0.1h because I wish to support TLS 1.1 and TLS 1.2; will there be any performance impact? I've downloaded and compiled OpenSSL 1.0.1h for a test. Let's see:

```
$ ./openssl-1.0.1h speed rsa
[...]
OpenSSL 1.0.1h 5 Jun 2014
built on: Thu Jul  3 18:30:06 BST 2014
options:bn(64,64) rc4(8x,int) des(idx,cisc,16,int) aes(partial) idea(int) ↩
blowfish(idx)
compiler: gcc -DOPENSSL_THREADS -D_REENTRANT -DDSO_DLFCN -DHAVE_DLFCN_H ↩
-Wa,--noexecstack -m64 -DL_ENDIAN -DTERMIO -O3 -Wall -DOPENSSL_IA32_SSE2 -DOPENSSL↩
_BN_ASM_MONT -DOPENSSL_BN_ASM_MONT5 -DOPENSSL_BN_ASM_GF2m -DSHA1_ASM -DSHA256_ASM ↩
-DSHA512_ASM -DMD5_ASM -DAES_ASM -DVPAES_ASM -DBSAES_ASM -DWHIRLPOOL_ASM -DGHASH↩
_ASM
                  sign    verify    sign/s verify/s
rsa  512 bits 0.000102s 0.000008s   9818.0 133081.7
rsa 1024 bits 0.000326s 0.000020s   3067.2  50086.9
rsa 2048 bits 0.002209s 0.000068s    452.8  14693.6
rsa 4096 bits 0.015748s 0.000255s     63.5   3919.4
```

Apparently, OpenSSL 1.0.1h is almost twice as fast on this server for my use case (2,048-bit RSA key): The performance went from 277 signatures/s to 450 signatures/s. This means that I'll get better performance if I upgrade. Always good news!

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 four cores, so that's what I'm going to use:

```
$ openssl speed -multi 4 rsa
[...]
OpenSSL 0.9.8k 25 Mar 2009
built on: Wed May 23 00:02:00 UTC 2012
options:bn(64,64) md2(int) rc4(ptr,char) des(idx,cisc,16,int) aes(partial) ↩
blowfish(ptr2)
compiler: cc -fPIC -DOPENSSL_PIC -DZLIB -DOPENSSL_THREADS -D_REENTRANT -DDSO_DLFCN ↩
-DHAVE_DLFCN_H -m64 -DL_ENDIAN -DTERMIO -O3 -Wa,--noexecstack -g -Wall -DMD32_REG↩
_T=int -DOPENSSL_BN_ASM_MONT -DSHA1_ASM -DSHA256_ASM -DSHA512_ASM -DMD5_ASM -DAES↩
_ASM
available timing options: TIMES TIMEB HZ=100 [sysconf value]
timing function used:
                  sign    verify    sign/s verify/s
rsa  512 bits 0.000030s 0.000003s  33264.5 363636.4
rsa 1024 bits 0.000143s 0.000008s   6977.9 125000.0
rsa 2048 bits 0.000917s 0.000027s   1090.7  37068.1
rsa 4096 bits 0.006123s 0.000094s    163.3  10652.6
```

As expected, the performance is almost four times better than before. I'm again looking at how many RSA signatures can be executed 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 1,090 signatures/second tells us that this server can handle about 1,000 brand-new TLS connections per second. In my case, that's sufficient—with a very healthy safety margin. Because I also have session resumption enabled on the server, I know that I can support many more than 1,000 TLS connections per second. I wish I had enough traffic on that server to worry about the performance of TLS.

Another reason why you shouldn't believe the output of the `speed` command too much is because it doesn't use the fastest available cipher implementations by default. In some ways, the default output is a lie. For example, on servers that support the AES-NI instruction set to accelerate AES computations, this feature won't be used by default when testing:

```
$ 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      67546.70k    74183.00k    69278.82k   155942.87k   156486.38k
```

To activate hardware acceleration, you have to use the `-evp` switch on the command line:

```
$ 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     188523.36k   223595.37k   229763.58k   203658.58k   206452.14k
```

# 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.

## 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 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 `basicConstraint` extension indicates that the certificate is a CA, and the `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.[12]

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

[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]
```

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

```
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 for the subordinate CA, start with the file we used for the root CA and make the changes listed here. 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 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're implemented, 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. But, when you really need to be certain of something, the only way is to get your hands dirty with OpenSSL.

## Connecting to SSL 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 SSL/TLS layer 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 -connect www.feistyduck.com:443
```

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: Tue, 10 Mar 2015 17:13:23 GMT
Server: Apache
Strict-Transport-Security: max-age=31536000
Cache-control: no-cache, must-revalidate
Content-Type: text/html;charset=UTF-8
Transfer-Encoding: chunked
Set-Cookie: JSESSIONID=7F3D840B9C2FDB1FF7E5731590BD9C99; Path=/; Secure; HttpOnly
Connection: close

read:errno=0
```

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=3 L = ValiCert Validation Network, O = "ValiCert, Inc.", OU = ValiCert Class ↵
2 Policy Validation Authority, CN = http://www.valicert.com/, emailAddress = ↵
info@valicert.com
verify error:num=19:self signed certificate in certificate chain
verify return:0
```

On my system (and possibly on yours), `s_client` doesn't pick up the default trusted certificates; it complains that there is a self-signed certificate in the certificate chain. In most cases, you won't care about certificate validation; but if you do, you will need to point `s_client` to the trusted certificates, like this:

```
$ openssl s_client -connect www.feistyduck.com:443 -CAfile /etc/ssl/certs↵
/ca-certificates.crt
CONNECTED(00000003)
depth=3 L = ValiCert Validation Network, O = "ValiCert, Inc.", OU = ValiCert Class ↵
2 > Policy Validation Authority, CN = http://www.valicert.com/, emailAddress = ↵
info@valicert.com
verify return:1
depth=2 C = US, O = "Starfield Technologies, Inc.", OU = Starfield Class 2 ↵
Certification Authority
verify return:1
depth=1 C = US, ST = Arizona, L = Scottsdale, O = "Starfield Technologies, Inc.", ↵
OU = http://certificates.starfieldtech.com/repository, CN = Starfield Secure ↵
Certification Authority, serialNumber = 10688435
verify return:1
depth=0 1.3.6.1.4.1.311.60.2.1.3 = GB, businessCategory = Private Organization, ↵
serialNumber = 06694169, C = GB, ST = London, L = London, O = Feisty Duck Ltd, CN ↵
= www.feistyduck.com
verify return:1
```

Instead of `s_client` complaining, you now see it verifying each of the certificates from the chain. For the verification to work, you must have access to a good selection of CA certificates. The path I used in the example (`/etc/ssl/certs/ca-certificates.crt`) is valid on Ubuntu 12.04 LTS but might not be valid on your system. If you don't want to use the system-provided CA certificates for this purpose, you can rely on those provided by Mozilla, as discussed in the section called "Building a Trust Store" in Chapter 1.

> **Warning**
>
> Apple's operating system OS X ships with a modified version of OpenSSL that sometimes overrides certificate validation. In other words, the `-CAfile` switch might not work as expected. You can fix this by setting the `OPENSSL_X509_TEA_DISABLE` environment variable before you invoke `s_client`.[1] Given that the default version of OpenSSL on OS X is from the 0.9.x branch and thus obsolete, it's best that you upgrade to the latest version—for example, using Homebrew or MacPorts.

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:/1.3.6.1.4.1.311.60.2.1.3=GB/businessCategory=Private Organization↵
/serialNumber=06694169/C=GB/ST=London/L=London/O=Feisty Duck Ltd↵
/CN=www.feistyduck.com
   i:/C=US/ST=Arizona/L=Scottsdale/O=Starfield Technologies, Inc./OU=http:/↵
/certificates.starfieldtech.com/repository/CN=Starfield Secure Certification ↵
Authority/serialNumber=10688435
 1 s:/C=US/ST=Arizona/L=Scottsdale/O=Starfield Technologies, Inc./OU=http:/↵
/certificates.starfieldtech.com/repository/CN=Starfield Secure Certification ↵
Authority/serialNumber=10688435
   i:/C=US/O=Starfield Technologies, Inc./OU=Starfield Class 2 Certification ↵
Authority
 2 s:/C=US/O=Starfield Technologies, Inc./OU=Starfield Class 2 Certification ↵
Authority
   i:/L=ValiCert Validation Network/O=ValiCert, Inc./OU=ValiCert Class 2 Policy ↵
Validation Authority/CN=http://www.valicert.com//emailAddress=info@valicert.com
 3 s:/L=ValiCert Validation Network/O=ValiCert, Inc./OU=ValiCert Class 2 Policy ↵
Validation Authority/CN=http://www.valicert.com//emailAddress=info@valicert.com
   i:/L=ValiCert Validation Network/O=ValiCert, Inc./OU=ValiCert Class 2 Policy ↵
Validation Authority/CN=http://www.valicert.com//emailAddress=info@valicert.com
```

[1] Apple OpenSSL Verification Surprises (Hynek Schlawack, 3 March 2014)

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-----
MIIF5zCCBM+gAwIBAgIHBG9JXlv9vTANBgkqhkiG9w0BAQUFADCB3DELMAkGA1UE
[30 lines removed...]
os5LW3PhHz8y9YFep2SV4c7+NrlZISHOZVzN
-----END CERTIFICATE-----
subject=/1.3.6.1.4.1.311.60.2.1.3=GB/businessCategory=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/serialNumber=10688435
```

> **Note**
>
> Whenever you see a long string of numbers instead of a name in a subject, it means that OpenSSL does not know the *object identifier* (OID) in question. OIDs are globally unique and unambiguous identifiers that are used to refer to "things." For example, in the previous output, the OID `1.3.6.1.4.1.311.60.2.1.3` should have been replaced with `jurisdictionOfIncorporationCountryName`, which is used in *extended validation* (EV) certificates.

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 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
```

```
---
SSL handshake has read 3043 bytes and written 375 bytes
---
New, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES256-SHA
Server public key is 2048 bit
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
SSL-Session:
    Protocol  : TLSv1.1
    Cipher    : ECDHE-RSA-AES256-SHA
    Session-ID: 032554E059DB27BF8CD87EBC53E9FF29376265F0BBFDBBFB7773D2277E5559F5
    Session-ID-ctx:
    Master-Key: 1A55823368DB6EFC397DEE2DC3382B5BB416A061C19CEE162362158E90F1FB0846E↩
EFDB2CCF564A18764F1A98F79A768
    Key-Arg   : None
    PSK identity: None
    PSK identity hint: None
    SRP username: None
    TLS session ticket lifetime hint: 300 (seconds)
    TLS session ticket:
    0000 - 77 c3 47 09 c4 45 e4 65-90 25 8b fd 77 4c 12 da   w.G..E.e.%..wL..
    0010 - 38 f0 43 09 08 a1 ec f0-8d 86 f8 b1 f0 7e 4b a9   8.C..........~K.
    0020 - fe 9f 14 8e 66 d7 5a dc-0f d0 0c 25 fc 99 b8 aa   ....f.Z....%....
    0030 - 8f 93 56 5a ac cd f8 66-ac 94 00 8b d1 02 63 91   ..VZ...f......c.
    0040 - 05 47 af 98 11 81 65 d9-48 5b 44 bb 41 d8 24 e8   .G....e.H[D.A.$.
    0050 - 2e 08 2d bb 25 59 f0 8f-bf aa 5c b6 fa 9c 12 a6   ..-.%Y....\.....
    0060 - a1 66 3f 84 2c f6 0f 06-51 c0 64 24 7a 9a 48 96   .f?.,...Q.d$z.H.
    0070 - a7 f6 a9 6e 94 f2 71 10-ff 00 4d 7a 97 e3 f5 8b   ...n..q...Mz....
    0080 - 2d 1a 19 9c 1a 8d e0 9c-e5 55 cd be d7 24 2e 24   -........U...$.$
    0090 - fc 59 54 b0 f8 f1 0a 5f-03 08 52 0d 90 99 c4 78   .YT...._..R....x
    00a0 - d2 93 61 d8 eb 76 15 27-03 5e a4 db 0c 05 bb 51   ..a..v.'.^.....Q
    00b0 - 6c 65 76 9b 4e 6b 6c 19-69 33 2a bd 02 1f 71 14   lev.Nkl.i3*...q.

    Start Time: 1390553737
    Timeout   : 300 (sec)
    Verify return code: 0 (ok)
---
```

The most important information here is the protocol version (TLS 1.1) and cipher suite used (`ECDHE-RSA-AES256-SHA`). 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. Once you understand what all of this output contains, you will rarely look at it.

> **Warning**
>
> Operating system distributions often ship tools that are different from the stock versions. We have another example of that here: the previous command negotiated TLS 1.1, even though the server supports TLS 1.2. Why? As it turns out, some OpenSSL versions shipped with Ubuntu 12.04 LTS disable TLS 1.2 for client connections in order to avoid certain interoperability issues. To avoid problems like these, I recommend that you always test with a version of OpenSSL that you configured and compiled.

## Testing Protocols that Upgrade to SSL

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 are `smtp`, `pop3`, `imap`, `ftp`, and `xmpp`.

## Using Different Handshake Formats

Sometimes, when you are trying to test a server using OpenSSL, your attempts to communicate with the server may fail even though you know the server supports TLS (e.g., you can see that TLS is working when you attempt to use a browser). One possible reason this might occur is that the server does not support the older SSL 2 handshake.

Because OpenSSL attempts to negotiate all protocols it understands and because SSL 2 can be negotiated only using the old SSL 2 handshake, it uses this handshake as the default. Even though it is associated with a very old and insecure protocol version, the old handshake format is not technically insecure. It supports upgrades, which means that a better protocol can be negotiated. However, this handshake format does not support many connection negotiation features that were designed after SSL 2.

Therefore, if something is not working and you're not sure what it is exactly, you can try to force OpenSSL to use the newer handshake format. You can do that by disabling SSL 2:

```
$ openssl s_client -connect www.feistyduck.com:443 -no_ssl2
```

Another way to achieve the same effect is to specify the desired server name on the command line:

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

In order to specify the server name, OpenSSL needs to use a feature of the newer handshake format (the feature is called *Server Name Indication* [SNI]), and that will force it to abandon the old format.

## 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' > www.feistyduck.com.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.

## 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.

```
    Protocol  : TLSv1.1
```

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`, or `-tls1_2` switches. Alternatively, you can choose which protocols you don't want to test by using one or many of the following: `-no_ssl2`, `-no_ssl3`, `-no_tls1`, `-no_tls1_1`, or `-no_tls1_2`.

> **Note**
>
> Not all versions of OpenSSL support all protocol versions. For example, the older versions of OpenSSL will not support TLS 1.1 and TLS 1.2, and the newer versions might not support older protocols, such as SSL 2.

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
---
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)
---
```

# Testing Cipher Suite Support

A little trick is required if you wish to use OpenSSL to determine if a remote server supports a particular cipher suite. The cipher configuration string is designed to select which suites you wish to use, but if you specify only one suite and successfully handshake with a server, then

you know that the server supports the suite. If the handshake fails, you know the support is not there.

As an example, to test if a server supports `RC4-SHA`, type:

```
$ openssl s_client -connect www.feistyduck.com:443 -cipher RC4-SHA
```

If you want to determine all suites supported by a particular server, start by invoking `openssl ciphers ALL` to obtain a list of all suites supported by your version of OpenSSL. Then submit them to the server one by one to test them individually. I am not suggesting that you do this manually; this is a situation in which a little automation goes a long way. In fact, this is a situation in which looking around for a good tool might be appropriate.

There is a disadvantage to testing this way, however. You can only test the suites that OpenSSL supports. This used to be a much bigger problem; before version 1.0, OpenSSL supported a much smaller number of suites (e.g., 32 on my server with version 0.9.8k). With a version from the 1.0.1 branch, you can test over 100 suites and probably most of the relevant ones.

No single SSL/TLS library supports all cipher suites, and that makes comprehensive testing difficult. For SSL Labs, I resorted to using partial handshakes for this purpose, with a custom client that pretends to support arbitrary suites. It actually can't negotiate even a single suite, but just proposing to negotiate is enough for servers to tell you if they support a suite or not. Not only can you test all the suites this way, but you can also do it very efficiently.

## Testing Servers that Require SNI

Initially, SSL and TLS were designed to support only one web site per IP endpoint (address and port combination). SNI is a TLS extension that enables use of more than one certificate on the same IP endpoint. TLS clients use the extension to send the desired name, and TLS servers use it to select the correct certificate to respond with. In a nutshell, SNI makes virtual secure hosting possible.

Because SNI is not yet very widely used by servers, in most cases you won't need to specify it on the `s_client` command line. But when you encounter an SNI-enabled system, one of three things can happen:

- Most often, you will get the same certificate you would get as if SNI information had not been supplied.
- The server might respond with the certificate for some site other than the one you wish to test.
- Very rarely, the server might abort the handshake and refuse the connection.

You can enable SNI in `s_client` with the `-servername` switch:

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

You can determine if a site requires SNI by testing with and without the SNI switch and checking if the certificates are the same. If they are not, SNI is required.

Sometimes, if the requested server name is not available, the server says so with a TLS warning. Even though this warning is not fatal as far as the server is concerned, the client might decide to close the connection. For example, with an older OpenSSL version (i.e., before 1.0.0), you will get the following error message:

```
$ /opt/openssl-0.9.8k/bin/openssl s_client -connect www.feistyduck.com:443 ↵
-servername xyz.com
CONNECTED(00000003)
1255:error:14077458:SSL routines:SSL23_GET_SERVER_HELLO:reason(1112):s23↵
_clnt.c:596:
```

## Testing Session Reuse

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
```

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:

```
New, TLSv1/SSLv3, Cipher is RC4-SHA
```

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

```
Reused, TLSv1/SSLv3, Cipher is RC4-SHA
```

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 -no_ssl2 2> ↵
/dev/null | grep 'New\|Reuse'
New, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES256-GCM-SHA384
Reused, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES256-GCM-SHA384
Reused, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES256-GCM-SHA384
Reused, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES256-GCM-SHA384
```

```
Reused, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES256-GCM-SHA384
Reused, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES256-GCM-SHA384
```

Here's what the command does:

- The `-reconnect` switch activates the session reuse mode.
- The `-no_ssl2` switch indicates that we do not wish to attempt an SSL 2 connection, which changes the handshake of the first connection to that of SSL 3 and better. The older, SSL 2 handshake format doesn't support TLS extensions and interferes with the session-reuse mechanism on servers that support session tickets.
- 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 it with the `-no_ticket` switch.

## 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 and the subject of the second match:

```
---
Certificate chain
 0 s:/1.3.6.1.4.1.311.60.2.1.3=GB/businessCategory=Private Organization↩
/serialNumber=06694169/C=GB/ST=London/L=London/O=Feisty Duck Ltd↩
```

```
/CN=www.feistyduck.com
   i:/C=US/ST=Arizona/L=Scottsdale/O=Starfield Technologies, Inc./OU=http:/↵
/certificates.starfieldtech.com/repository/CN=Starfield Secure Certification ↵
Authority/serialNumber=10688435
-----BEGIN CERTIFICATE-----
MIIF5zCCBM+gAwIBAgIHBG9JXlv9vTANBgkqhkiG9w0BAQUFADCB3DELMAkGA1UE
[30 lines of text removed]
os5LW3PhHz8y9YFep2SV4c7+NrlZISHOZVzN
-----END CERTIFICATE-----
 1 s:/C=US/ST=Arizona/L=Scottsdale/O=Starfield Technologies, Inc./OU=http:/↵
/certificates.starfieldtech.com/repository/CN=Starfield Secure Certification ↵
Authority/serialNumber=10688435
   i:/C=US/O=Starfield Technologies, Inc./OU=Starfield Class 2 Certification ↵
Authority
-----BEGIN CERTIFICATE-----
MIIFBzCCA++gAwIBAgICAgEwDQYJKoZIhvcNAQEFBQAwaDELMAkGA1UEBhMCVVMx
[...]
```

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:
        OCSP - URI:http://ocsp.starfieldtech.com/
        CA Issuers - URI:http://certificates.starfieldtech.com/repository/sf↵
_intermediate.crt
[...]
```

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.starfieldtech.com/
```

Now you can submit the OCSP request:

```
$ openssl ocsp -issuer issuer.crt -cert fd.crt -url http://ocsp.starfieldtech.com/ ↵
-CAfile issuer.crt
WARNING: no nonce in response
```

```
Response verify OK
fd.crt: good
        This Update: Feb 18 17:59:10 2013 GMT
        Next Update: Feb 18 23:59:10 2013 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 if you are using OpenSSL 1.0.0 or later by using the undocumented `-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.starfieldtech.com/ ↵
-CAfile issuer.crt -no_nonce -header Host ocsp.starfieldtech.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: C = US, O = "GeoTrust, Inc.", CN = RapidSSL OCSP-TGV Responder
    Produced At: Jan 22 17:48:55 2014 GMT
    Responses:
    Certificate ID:
      Hash Algorithm: sha1
      Issuer Name Hash: 834F7C75EAC6542FED58B2BD2B15802865301E0E
      Issuer Key Hash: 6B693D6A18424ADD8F026539FD35248678911630
      Serial Number: 0FE760
    Cert Status: good
    This Update: Jan 22 17:48:55 2014 GMT
    Next Update: Jan 29 17:48:55 2014 GMT
[...]
```

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; you can extract it with the following command:

```
$ openssl x509 -in fd.crt -noout -text | grep crl
                  URI:http://rapidssl-crl.geotrust.com/crls/rapidssl.crl
```

Then fetch the CRL from the CA:

```
$ wget http://rapidssl-crl.geotrust.com/crls/rapidssl.crl
```

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

```
$ openssl crl -in rapidssl.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=0FE760
```

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

```
$ openssl crl -in rapidssl.crl -inform DER -text -noout
Certificate Revocation List (CRL):
        Version 2 (0x1)
    Signature Algorithm: sha1WithRSAEncryption
        Issuer: /C=US/O=GeoTrust, Inc./CN=RapidSSL CA
        Last Update: Jan 25 11:03:00 2014 GMT
        Next Update: Feb  4 11:03:00 2014 GMT
        CRL extensions:
            X509v3 Authority Key Identifier:
                keyid:6B:69:3D:6A:18:42:4A:DD:8F:02:65:39:FD:35:24:86:78:91:16:30

            X509v3 CRL Number:
                92103
Revoked Certificates:
    Serial Number: 0F38D7
        Revocation Date: Nov 26 20:07:51 2013 GMT
    Serial Number: 6F29
        Revocation Date: Aug 15 20:48:57 2011 GMT
[...]
    Serial Number: 0C184E
        Revocation Date: Jun 13 23:00:12 2013 GMT
    Signature Algorithm: sha1WithRSAEncryption
```

```
95:df:e5:59:bc:95:e8:2f:bb:0a:4f:20:ad:ca:8f:78:16:54:
35:32:55:b0:c9:be:5b:89:da:ba:ae:67:19:6e:07:23:4d:5f:
16:18:5c:f3:91:15:da:9e:68:b0:81:da:68:26:a0:33:9d:34:
2d:5c:84:4b:70:fa:76:27:3a:fc:15:27:e8:4b:3a:6e:2e:1c:
2c:71:58:15:8e:c2:7a:ac:9f:04:c0:f6:3c:f5:ee:e5:77:10:
e7:88:83:00:44:c4:75:c4:2b:d3:09:55:b9:46:bf:fd:09:22:
de:ab:07:64:3b:82:c0:4c:2e:10:9b:ab:dd:d2:cb:0c:a9:b0:
51:7b:46:98:15:83:97:e5:ed:3d:ea:b9:65:d4:10:05:10:66:
09:5c:c9:d3:88:c6:fb:28:0e:92:1e:35:b0:e0:25:35:65:b9:
98:92:c7:fd:e2:c7:cc:e3:b5:48:08:27:1c:e5:fc:7f:31:8f:
0a:be:b2:62:dd:45:3b:fb:4f:25:62:66:45:34:eb:63:44:43:
cb:3b:40:77:b3:7f:6c:83:5c:99:4b:93:d9:39:62:48:5d:8c:
63:e2:a8:26:64:5d:08:e5:c3:08:e2:09:b0:d1:44:7b:92:96:
aa:45:9f:ed:36:f8:62:60:66:42:1c:ea:e9:9a:06:25:c4:85:
fc:77:f2:71
```

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 is correct (e.g., if necessary, remove the `0x` prefix, omit any leading zeros, and convert all letters to uppercase). For example:

```
$ openssl crl -in rapidssl.crl -inform DER -text -noout | grep FE760
```

## Testing Renegotiation

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 (emphasis mine):

```
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
```

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 is not needed 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, you 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:

```
HEAD / HTTP/1.0
R
RENEGOTIATING
depth=3 C = US, O = "VeriSign, Inc.", OU = Class 3 Public Primary Certification ↩
Authority
verify return:1
depth=2 C = US, O = "VeriSign, Inc.", OU = VeriSign Trust Network, OU = "(c) 2006 ↩
VeriSign, Inc. - For authorized use only", CN = VeriSign Class 3 Public Primary ↩
Certification Authority - G5
verify return:1
depth=1 C = US, O = "VeriSign, Inc.", OU = VeriSign Trust Network, OU = Terms of ↩
use at https://www.verisign.com/rpa (c)06, CN = VeriSign Class 3 Extended ↩
Validation SSL CA
verify return:1
depth=0 1.3.6.1.4.1.311.60.2.1.3 = US, 1.3.6.1.4.1.311.60.2.1.2 = California, ↩
businessCategory = Private Organization, serialNumber = C2759208, C = US, ST = ↩
California, L = Mountain View, O = Mozilla Corporation, OU = Terms of use at ↩
www.verisign.com/rpa (c)05, OU = Terms of use at www.verisign.com/rpa (c)05, CN = ↩
addons.mozilla.org
verify return:1
Host: addons.mozilla.org

HTTP/1.1 301 MOVED PERMANENTLY
Content-Type: text/html; charset=utf-8
Date: Tue, 05 Jun 2012 16:42:51 GMT
Location: https://addons.mozilla.org/en-US/firefox/
Keep-Alive: timeout=5, max=998
Transfer-Encoding: chunked
Connection: close

read:errno=0
```

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 prefer the secure option.

## Testing for the BEAST Vulnerability

The BEAST attack exploits a weakness that exists in all versions of SSL, and TLS protocols before TLS 1.1. The weakness affects all CBC suites and both client and server data streams; however, the BEAST attack works only against the client side. Most modern browsers use the so-called 1/n-1 split as a workaround to prevent exploitation, but some servers continue to deploy mitigations on their end, especially if they have a user base that relies on older (and unpatched) browsers.

The ideal mitigation approach is to rely only on TLS 1.1 and better, but these newer protocols are not yet sufficiently widely supported. The situation is complicated by the fact that RC4 itself is now considered insecure. If you think BEAST is more dangerous than RC4 weaknesses, you might deploy TLS 1.2 for use with up-to-date clients, but force RC4 with everyone else.

**Strict mitigation**

Do not support any CBC suites when protocols TLS 1.0 and earlier are used, leaving only RC4 suites enabled. Clients that don't support RC4 won't be able to negotiate a secure connection. This mode excludes some potential web site users, but it's required by some PCI assessors.

**RC4 prioritization**

Because only a very small number of clients do not support RC4, the second approach is to leave CBC suites enabled, but enforce RC4 with all clients that support it. This approach provides protection to all but a very small number of visitors.

How you are going to test depends on what behavior you expect of the server. With both approaches, we want to ensure that only insecure protocols are used by using the `-no_ssl2`, `-no_tls_1_1`, and `-no_tls_1_2` switches.

To test for strict mitigation, attempt to connect while disabling all RC4 suites on your end:

```
$  echo | openssl s_client -connect www.feistyduck.com:443 \
-cipher 'ALL:!RC4' -no_ssl2 -no_tls1_1 -no_tls1_2
```

If the connection is successful (which is possible only if a vulnerable CBC suite is used), you know that strict mitigation is not in place.

To test for RC4 prioritization, attempt to connect with all RC4 suites moved to the end of the cipher suite list:

```
$ echo | openssl s_client -connect www.feistyduck.com:443 \
-cipher 'ALL:+RC4' -no_ssl2 -no_tls1_1 -no_tls1_2
```

A server that prioritizes RC4 will choose one of RC4 suites for the connection, ignoring all the CBC suites that were also offered. If you see anything else, you know that the server does not have any BEAST mitigations in place.

## Testing for Heartbleed

You can test for Heartbleed manually or by using one of the available tools. (There are many tools, 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.[2] Given the seriousness of Heartbleed, it's best to either test manually or by using a tool that

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

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 (version 1.0.1 and newer). 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
[...]
```

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, then connect to the server, 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 pair. 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 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, enter the `B` command again.

## Determining the Strength of Diffie-Hellman Parameters

In OpenSSL 1.0.2 and newer, 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 some 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-1.0.2 s_client -connnect www.feistyduck.com:443 -cipher kEDH
[...]
---
```

```
No client certificate CA names sent
Peer signing digest: SHA512
Server Temp Key: DH, 2048 bits
---
[...]
```

Servers that support export suites might actually offer even weaker DH parameters. To check for that possibility, connect while offering only export DHE suites:

```
$ openssl-1.0.2 s_client -connnect 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.

# A SSL/TLS Deployment Best Practices

*This appendix contains SSL/TLS Deployment Best Practices, which is an SSL Labs publication I began to work on in 2012 and continue to maintain. It's included here with permission from Qualys, Inc.*

Obtaining a comprehensive understanding of the SSL/TLS and PKI landscape requires a lot of time and dedication. In my experience, most people don't need to know everything, but it's tricky to find the small bits that they do need to know. For these individuals, I maintain this guide: the smallest possible document that could reasonably claim to give a complete picture of the problems facing someone who wishes to deploy a secure SSL/TLS server.

## 1 Private Key and Certificate

In TLS, all security starts with the server's cryptographic identity; a strong private key is needed to prevent attackers from carrying out impersonation attacks. Equally important is to have a valid and strong certificate, which grants the private key the right to represent a particular hostname. Without these two fundamental building blocks, nothing else can be secure.

### 1.1 Use 2048-Bit Private Keys

For most web sites, security provided by 2,048-bit RSA keys is sufficient. The RSA public key algorithm is widely supported, which makes keys of this type a safe default choice. At 2,048 bits, such keys provide about 112 bits of security. If you want more security than this, note that RSA keys don't scale very well. To get 128 bits of security, you need 3,072-bit RSA keys, which are noticeably slower. ECDSA keys provide an alternative that offers better security and better performance. At 256 bits, ECDSA keys provide 128 bits of security. A small number of older clients don't support ECDSA, but modern clients do. It's possible to get the best of both worlds and deploy with RSA and ECDSA keys simultaneously if you don't mind the overhead of managing such a setup.

## 1.2 Protect Private Keys

Treat your private keys as an important asset, restricting access to the smallest possible group of employees while still keeping your arrangements practical. Recommended policies include the following:

- Generate private keys on a trusted computer with sufficient entropy. Some CAs offer to generate private keys for you; run away from them.
- Password-protect keys from the start to prevent compromise when they are stored in backup systems. Private key passwords don't help much in production because a knowledgeable attacker can always retrieve the keys from process memory. There are hardware devices (called *Hardware Security Modules*, or HSMs) that can protect private keys even in the case of server compromise, but they are expensive and thus justifiable only for organizations with strict security requirements.
- After compromise, revoke old certificates and generate new keys.
- Renew certificates yearly, and more often if you can automate the process. Most sites should assume that a compromised certificate will be impossible to revoke reliably; certificates with shorter lifespans are therefore more secure in practice.
- Unless keeping the same keys is important for public key pinning, you should also generate new private keys whenever you're getting a new certificate.

## 1.3 Ensure Sufficient Hostname Coverage

Ensure that your certificates cover all the names you wish to use with a site. Your goal is to avoid invalid certificate warnings, which confuse users and weaken their confidence.

Even when you expect to use only one domain name, remember that you cannot control how your users arrive at the site or how others link to it. In most cases, you should ensure that the certificate works with and without the *www* prefix (e.g., that it works for both *example.com* and *www.example.com*). The rule of thumb is that a secure web server should have a certificate that is valid for every DNS name configured to point to it.

Wildcard certificates have their uses, but avoid using them if it means exposing the underlying keys to a much larger group of people, and especially if doing so crosses team or department boundaries. In other words, the fewer people there are with access to the private keys, the better. Also be aware that certificate sharing creates a bond that can be abused to transfer vulnerabilities from one web site or server to all other sites and servers that use the same certificate (even when the underlying private keys are different).

## 1.4 Obtain Certificates from a Reliable CA

Select a *Certification Authority* (CA) that is reliable and serious about its certificate business and security. Consider the following criteria when selecting your CA:

**Security posture**

All CAs undergo regular audits, but some are more serious about security than others. Figuring out which ones are better in this respect is not easy, but one option is to examine their security history, and, more important, how they have reacted to compromises and if they have learned from their mistakes.

**Business focus**

CAs whose activities constitute a substantial part of their business have everything to lose if something goes terribly wrong, and they probably won't neglect their certificate division by chasing potentially more lucrative opportunities elsewhere.

**Services offered**

At a minimum, your selected CA should provide support for both *Certificate Revocation List* (CRL) and *Online Certificate Status Protocol* (OCSP) revocation methods, with rock-solid network availability and performance. Many sites are happy with domain-validated certificates, but you also should consider if you'll ever require *Extended Validation* (EV) certificates. In either case, you should have a choice of public key algorithm. Most web sites use RSA today, but ECDSA may become important in the future because of its performance advantages.

**Certificate management options**

If you need a large number of certificates and operate in a complex environment, choose a CA that will give you good tools to manage them.

**Support**

Choose a CA that will give you good support if and when you need it.

> **Note**
>
> For best results, deploy new certificates to production about one week after getting them from your CA. This practice *(1)* helps avoid certificate warnings for some users who don't have the correct time on their computers and *(2)* helps avoid failed revocation checks with CAs who need extra time to propagate new certificates to their OCSP responders.

### 1.5 Use Strong Certificate Signature Algorithms

Certificate security depends *(1)* on the strength of the private key that was used to sign the certificate and *(2)* the strength of the hashing function used in the signature. Until recently, most certificates relied on the SHA1 hashing function, which is now considered insecure. As a result, we're currently in transition to SHA256. As of January 2016, you shouldn't be able to get a SHA1 certificate from a public CA. The existing SHA1 certificates will continue to work (with warnings in some browsers), but only until the end of 2016.

## 2 Configuration

With correct TLS server configuration, you ensure that your credentials are properly presented to the site's visitors, that only secure cryptographic primitives are used, and that all known weaknesses are mitigated.

### 2.1 Use Complete Certificate Chains

In most deployments, the server certificate alone is insufficient; two or more certificates are needed to build a complete chain of trust. A common configuration problem occurs when deploying a server with a valid certificate, but without all the necessary intermediate certificates. To avoid this situation, simply use all the certificates provided to you by your CA.

An invalid certificate chain effectively renders the server certificate invalid and results in browser warnings. In practice, this problem is sometimes difficult to diagnose because some browsers can reconstruct incomplete chains and some can't. All browsers tend to cache and reuse intermediate certificates.

### 2.2 Use Secure Protocols

There are five protocols in the SSL/TLS family: SSL v2, SSL v3, TLS v1.0, TLS v1.1, and TLS v1.2:

- SSL v2 is insecure and must not be used. This protocol version is so bad that it can be used to attack RSA keys and sites with the same name even if they are on an entirely different servers (the DROWN attack).
- SSL v3 is insecure when used with HTTP (the POODLE attack) and weak when used with other protocols. It's also obsolete and shouldn't be used.
- TLS v1.0 is also a legacy protocol that shouldn't be used, but it's typically still necessary in practice. Its major weakness (BEAST) has been mitigated in modern browsers, but other problems remain.

- TLS v1.1 and v1.2 are both without known security issues, but only v1.2 provides modern cryptographic algorithms.

TLS v1.2 should be your main protocol because it's the only version that offers modern *authenticated encryption* (also known as AEAD). If you don't support TLS v1.2 today, your security is lacking.

In order to support older clients, you may need to continue to support TLS v1.0 and TLS v1.1 for now. However, you should plan to retire TLS v1.0 in the near future. For example, the PCI DSS standard will require all sites that accept credit card payments to remove support for TLS v1.0 by June 2018.

Work is currently under way to design TLS v1.3, with the aims to remove all obsolete and insecure features and to make improvements that will keep our communication secure in the following decades.

## 2.3 Use Secure Cipher Suites

To communicate securely, you must first ascertain that you are communicating directly with the desired party (and not through someone else who will eavesdrop) and exchanging data securely. In SSL and TLS, cipher suites define how secure communication takes place. They are composed from varying building blocks with the idea of achieving security through diversity. If one of the building blocks is found to be weak or insecure, you should be able to switch to another.

You should rely chiefly on the AEAD suites that provide strong authentication and key exchange, forward secrecy, and encryption of at least 128 bits. Some other, weaker suites may still be supported, provided they are negotiated only with older clients that don't support anything better.

There are several obsolete cryptographic primitives that *must* be avoided:

- Anonymous Diffie-Hellman (ADH) suites do not provide authentication.
- NULL cipher suites provide no encryption.
- Export cipher suites are insecure when negotiated in a connection, but they can also be used against a server that prefers stronger suites (the FREAK attack).
- Suites with weak ciphers (typically of 40 and 56 bits) use encryption that can easily be broken.
- RC4 is insecure.
- 3DES is slow and weak.

Use the following suite configuration, designed for both RSA and ECDSA keys, as your starting point:

```
ECDHE-ECDSA-AES128-GCM-SHA256
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-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-AES256-GCM-SHA384
DHE-RSA-AES128-SHA
DHE-RSA-AES256-SHA
DHE-RSA-AES128-SHA256
DHE-RSA-AES256-SHA256
EDH-RSA-DES-CBC3-SHA
```

> **Note**
>
> This example configuration uses nonstandard suite names required by OpenSSL. For deployment in other environments (e.g., IIS), you'll need to use the standard TLS suite names.[1]

## 2.4 Select Best Cipher Suites

In SSL v3 and later protocol versions, clients submit a list of cipher suites that they support, and servers choose one suite from the list to use for the connection. Not all servers do this well, however; some will select the first supported suite from the client's list. Having servers actively select the best available cipher suite is critical for achieving the best security.

## 2.5 Use Forward Secrecy

*Forward secrecy* (sometimes also called *perfect forward secrecy*) is a protocol feature that enables secure conversations that are not dependent on the server's private key. With cipher suites that do not provide forward secrecy, someone who can recover a server's private key can decrypt

[1] Transport Layer Security (TLS) Parameters (IANA, retrieved 18 March 2016)

*all* earlier recorded encrypted conversations. You need to support and prefer ECDHE suites in order to enable forward secrecy with modern web browsers. To support a wider range of clients, you should also use DHE suites as fallback after ECDHE. Avoid the RSA key exchange unless absolutely necessary. My proposed default configuration in Section 2.3 contains only suites that provide forward secrecy.

### 2.6 Use Strong Key Exchange

For the key exchange, public sites can typically choose between the classic *ephemeral Diffie-Hellman* key exchange (DHE) and its elliptic curve variant, ECDHE. There are other key exchange algorithms, but they're generally insecure in one way or another. The RSA key exchange is still very popular, but it doesn't provide forward secrecy.

In 2015, a group of researchers published new attacks against DHE; their work is known as the Logjam attack.[2] The researchers discovered that lower-strength DH key exchanges (e.g., 768 bits) can easily be broken and that some well-known 1,024-bit DH groups can be broken by state agencies. To be on the safe side, if deploying DHE, configure it with at least 2,048 bits of security. Some older clients (e.g., Java 6) might not support this level of strength. For performance reasons, most servers should prefer ECDHE, which is both stronger and faster. The `secp256r1` named curve (also known as `P-256`) is a good choice in this case.

### 2.7 Mitigate Known Problems

There have been several serious attacks against SSL and TLS in recent years, but they should generally not concern you if you're running up-to-date software and following the advice in this guide. (If you're not, I'd advise testing your systems using SSL Labs and taking it from there.) However, nothing is perfectly secure, which is why it is a good practice to keep an eye on what happens in security. Promptly apply vendor patches if and when they become available; otherwise, rely on workarounds for mitigation.

## 3 Performance

Security is our main focus in this guide, but we must also pay attention to performance; a secure service that does not satisfy performance criteria will no doubt be dropped. With proper configuration, TLS can be quite fast. With modern protocols—for example, HTTP/2—it might even be faster than plaintext communication.

[2] Weak Diffie-Hellman and the Logjam Attack (retrieved 16 March 2016)

## 3.1 Avoid Too Much Security

The cryptographic handshake, which is used to establish secure connections, is an operation for which the cost is highly influenced by private key size. Using a key that is too short is insecure, but using a key that is too long will result in "too much" security and slow operation. For most web sites, using RSA keys stronger than 2,048 bits and ECDSA keys stronger than 256 bits is a waste of CPU power and might impair user experience. Similarly, there is little benefit to increasing the strength of the ephemeral key exchange beyond 2,048 bits for DHE and 256 bits for ECDHE. There are no clear benefits of using encryption above 128 bits.

## 3.2 Use Session Resumption

Session resumption is a performance-optimization technique that makes it possible to save the results of costly cryptographic operations and to reuse them for a period of time. A disabled or nonfunctional session resumption mechanism may introduce a significant performance penalty.

## 3.3 Use WAN Optimization and HTTP/2

These days, TLS overhead doesn't come from CPU-hungry cryptographic operations, but from network latency. A TLS handshake, which can start only after the TCP handshake completes, requires a further exchange of packets and is more expensive the further away you are from the server. The best way to minimize latency is to avoid creating new connections—in other words, to keep existing connections open for a long time (keep-alives). Other techniques that provide good results include supporting modern protocols such as HTTP/2 and using WAN optimization (usually via content delivery networks).

## 3.4 Cache Public Content

When communicating over TLS, browsers might assume that all traffic is sensitive. They will typically use the memory to cache certain resources, but once you close the browser, all the content may be lost. To gain a performance boost and enable long-term caching of some resources, mark public resources (e.g., images) as public.

## 3.5 Use OCSP Stapling

*OCSP stapling* is an extension of the OCSP protocol that delivers revocation information as part of the TLS handshake, directly from the server. As a result, the client does not need to contact OCSP servers for out-of-band validation and the overall TLS connection time is

significantly reduced. OCSP stapling is an important optimization technique, but you should be aware that not all web servers provide solid OCSP stapling implementations. Combined with a CA that has a slow or unreliable OCSP responder, such web servers might create performance issues. For best results, simulate failure conditions to see if they might impact your availability.

### 3.6 Use Fast Cryptographic Primitives

In addition to providing the best security, my recommended cipher suite configuration also provides the best performance. Whenever possible, use CPUs that support hardware-accelerated AES. After that, if you really want a further performance edge (probably not needed for most sites), consider using ECDSA keys.

## 4 HTTP and Application Security

The HTTP protocol and the surrounding platform for web application delivery continued to evolve rapidly after SSL was born. As a result of that evolution, the platform now contains features that can be used to defeat encryption. In this section, we list those features, along with ways to use them securely.

### 4.1 Encrypt Everything

The fact that encryption is optional is probably one of the biggest security problems today. We see the following problems:

- No TLS on sites that need it
- Sites that have TLS but that do not enforce it
- Sites that mix TLS and non-TLS content, sometimes even within the same page
- Sites with programming errors that subvert TLS

Although many of these problems can be mitigated if you know exactly what you're doing, the only way to reliably protect web site communication is to enforce encryption throughout—without exception.

### 4.2 Eliminate Mixed Content

Mixed-content pages are those that are transmitted over TLS but include resources (e.g., JavaScript files, images, CSS files) that are not transmitted over TLS. Such pages are not secure. An active man-in-the-middle (MITM) attacker can piggyback on a single unprotected

JavaScript resource, for example, and hijack the entire user session. Even if you follow the advice from the previous section and encrypt your entire web site, you might still end up retrieving some resources unencrypted from third-party web sites.

### 4.3 Understand and Acknowledge Third-Party Trust

Web sites often use third-party services activated via JavaScript code downloaded from another server. A good example of such a service is Google Analytics, which is used on large parts of the Web. Such inclusion of third-party code creates an implicit trust connection that effectively gives the other party full control over your web site. The third party may not be malicious, but large providers of such services are increasingly seen as targets. The reasoning is simple: if a large provider is compromised, the attacker is automatically given access to all the sites that depend on the service.

If you follow the advice from Section 4.2, at least your third-party links will be encrypted and thus safe from MITM attacks. However, you should go a step further than that: learn what services you use and remove them, replace them with safer alternatives, or accept the risk of their continued use. A new technology called *subresource integrity* (SRI) could be used to reduce the potential exposure via third-party resources.[3]

### 4.4 Secure Cookies

To be properly secure, a web site requires TLS, but also that all its cookies are explicitly marked as secure when they are created. Failure to secure the cookies makes it possible for an active MITM attacker to tease some information out through clever tricks, even on web sites that are 100% encrypted. For best results, consider adding cryptographic integrity validation or even encryption to your cookies.

## 4.5 Secure HTTP Compression

The 2012 CRIME attack showed that TLS compression can't be implemented securely. The only solution was to disable TLS compression altogether. The following year, two further attack variations followed. TIME and BREACH focused on secrets in HTTP response bodies compressed using HTTP compression. Unlike TLS compression, HTTP compression is a necessity and can't be turned off. Thus, to address these attacks, changes to application code need to be made.[4]

---

[3] Subresource Integrity (Mozilla Developer Network, retrieved 16 March 2016)

[4] Defending against the BREACH Attack (Qualys Security Labs; 7 August 2013)

TIME and BREACH attacks are not easy to carry out, but if someone is motivated enough to use them, the impact is roughly equivalent to a successful *Cross-Site Request Forgery* (CSRF) attack.

## 4.6 Deploy HTTP Strict Transport Security

*HTTP Strict Transport Security* (HSTS) is a safety net for TLS. It was designed to ensure that security remains intact even in the case of configuration problems and implementation errors. To activate HSTS protection, you add a new response header to your web sites. After that, browsers that support HSTS (all modern browsers at this time) enforce it.

The goal of HSTS is simple: after activation, it does not allow any insecure communication with the web site that uses it. It achieves this goal by automatically converting all plaintext links to secure ones. As a bonus, it also disables click-through certificate warnings. (Certificate warnings are an indicator of an active MITM attack. Studies have shown that most users click through these warnings, so it is in your best interest to never allow them.)

Adding support for HSTS is the single most important improvement you can make for the TLS security of your web sites. New sites should always be designed with HSTS in mind and the old sites converted to support it wherever possible and as soon as possible. For best security, consider using *HSTS preloading*,[5] which embeds your HSTS configuration in modern browsers, making even the first connection to your site secure.

The following configuration example activates HSTS on the main hostname and all its subdomains for a period of one year, while also allowing preloading:

```
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
```

## 4.7 Deploy Content Security Policy

*Content Security Policy* (CSP) is a security mechanism that web sites can use to restrict browser operation. Although initially designed to address *Cross-Site Scripting* (XSS), CSP is constantly evolving and supports features that are useful for enhancing TLS security. In particular, it can be used to restrict mixed content when it comes to third-party web sites, for which HSTS doesn't help.

To deploy CSP to prevent third-party mixed content, use the following configuration:

```
Content-Security-Policy: default-src https: 'unsafe-inline' 'unsafe-eval';
```

[5] HSTS Preload List (Google developers, retrieved 16 March 2016)

```
connect-src https: wss:
```

> **Note**
>
> This is not the best way to deploy CSP. In order to provide an example that doesn't break anything except mixed content, I had to disable some of the default security features. Over time, as you learn more about CSP, you should change your policy to bring them back.

### 4.8 Do Not Cache Sensitive Content

All sensitive content must be communicated only to the intended parties and treated accordingly by all devices. Although proxies do not see encrypted traffic and cannot share content among users, the use of cloud-based application delivery platforms is increasing, which is why you need to be very careful when specifying what is public and what is not.

### 4.9 Consider Other Threats

TLS is designed to address only one aspect of security—confidentiality and integrity of the communication between you and your users—but there are many other threats that you need to deal with. In most cases, that means ensuring that your web site does not have other weaknesses.

## 5 Validation

With many configuration parameters available for tweaking, it is difficult to know in advance what impact certain changes will have. Further, changes are sometimes made accidentally; software upgrades can introduce changes silently. For that reason, we advise that you use a comprehensive SSL/TLS assessment tool initially to verify your configuration to ensure that you start out secure, and then periodically to ensure that you stay secure. For public web sites, we recommend the free SSL Labs server test.[6]

## 6 Advanced Topics

The following advanced topics are currently outside the scope of our guide. They require a deeper understanding of SSL/TLS and *Public Key Infrastructure* (PKI), and they are still being debated by experts.

[6] SSL Labs (retrieved 16 March 2016)

**Public Key Pinning**

*Public key pinning* is designed to give web site operators the means to restrict which CAs can issue certificates for their web sites. This feature has been deployed by Google for some time now (hardcoded into their browser, Chrome) and has proven to be very useful in preventing attacks and making the public aware of them. In 2014, Firefox also added support for hardcoded pinning. A standard called *Public Key Pinning Extension for HTTP*[7] is now available. Public key pinning addresses the biggest weakness of PKI (the fact that any CA can issue a certificate for any web site), but it comes at a cost; deploying requires significant effort and expertise, and creates risk of losing control of your site (if you end up with invalid pinning configuration). You should consider pinning largely only if you're managing a site that might be realistically attacked via a fraudulent certificate.

**DNSSEC and DANE**

*Domain Name System Security Extensions* (DNSSEC) is a set of technologies that add integrity to the domain name system. Today, an active network attacker can easily hijack any DNS request and forge arbitrary responses. With DNSSEC, all responses can be cryptographically tracked back to the DNS root. *DNS-based Authentication of Named Entities* (DANE) is a separate standard that builds on top of DNSSEC to provide bindings between DNS and TLS. DANE could be used to augment the security of the existing CA-based PKI ecosystem or bypass it altogether.

Even though not everyone agrees that DNSSEC is a good direction for the Internet, support for it continues to improve. Browsers don't yet support either DNSSEC or DANE (preferring similar features provided by HSTS and HPKP instead), but there is some indication that they are starting to be used to improve the security of email delivery.

# 7 Changes

The first release of this guide was on 24 February 2012. This section tracks the document changes over time, starting with version 1.3.

## Version 1.3 (17 September 2013)

The following changes were made in this version:

[7] RFC 7469: Public Key Pinning Extension for HTTP (Evans et al, April 2015)

- Recommend replacing 1024-bit certificates straight away.
- Recommend against supporting SSL v3.
- Remove the recommendation to use RC4 to mitigate the BEAST attack server-side.
- Recommend that RC4 is disabled.
- Recommend that 3DES is disabled in the near future.
- Warn about the CRIME attack variations (TIME and BREACH).
- Recommend supporting forward secrecy.
- Add discussion of ECDSA certificates.

## Version 1.4 (8 December 2014)

The following changes were made in this version:

- Discuss SHA1 deprecation and recommend migrating to the SHA2 family.
- Recommend that SSL v3 is disabled and mention the POODLE attack.
- Expand Section 3.1 to cover the strength of the DHE and ECDHE key exchanges.
- Recommend OCSP Stapling as a performance-improvement measure, promoting it to Section 3.5.

## Version 1.5 (8 June 2016)

The following changes were made in this version:

- Refreshed the entire document to keep up with the times.
- Recommended use of authenticated cipher suites.
- Spent more time discussing key exchange strength and the Logjam attack.
- Removed the recommendation to disable client-initiated renegotiation. Modern software does this anyway, and it might be impossible or difficult to disable it with something older. At the same time, the DoS vector isn't particularly strong. Overall, I feel it's better to spend available resources fixing something else.
- Added a warning about flaky OCSP stapling implementations.
- Added mention of subresource integrity enforcement.
- Added mention of cookie integrity validation and encryption.

- Added mention of HSTS preloading.
- Recommended using CSP for better handling of third-party mixed content.
- Mentioned FREAK, Logjam, and DROWN attacks.
- Removed the section that discussed mitigation of various TLS attacks, which are largely obsolete by now, especially if the advice presented here is followed. Moved discussion of CRIME variants into a new section.
- Added a brief discussion of DNSSEC and DANE to the Advanced section.

## Acknowledgments

Special thanks to Marsh Ray, Nasko Oskov, Adrian F. Dimcev, and Ryan Hurst for their valuable feedback and help in crafting the initial version of this document. Also thanks to many others who generously share their knowledge of security and cryptography with the world. The guidelines presented here draw on the work of the entire security community.

## About SSL Labs

SSL Labs (*www.ssllabs.com*) is Qualys's research effort to understand SSL/TLS and PKI as well as to provide tools and documentation to assist with assessment and configuration. Since 2009, when SSL Labs was launched, hundreds of thousands of assessments have been performed using the free online assessment tool. Other projects run by SSL Labs include periodic Internet-wide surveys of TLS configuration and SSL Pulse, a monthly scan of about 150,000 most popular TLS-enabled web sites in the world.

## About Qualys

Qualys, Inc. (NASDAQ: QLYS), is a pioneer and leading provider of cloud security and compliance solutions with over 8,800 customers in more than 100 countries, including a majority of each of the Forbes Global 100 and Fortune 100. The QualysGuard Cloud Platform and integrated suite of solutions help organizations simplify security operations and lower the cost of compliance by delivering critical security intelligence on demand and automating the full spectrum of auditing, compliance, and protection for IT systems and web applications. Founded in 1999, Qualys has established strategic partnerships with leading managed service providers and consulting organizations, including Accenture, BT, Cognizant Technology Solutions, Dell SecureWorks, Fujitsu, HCL Comnet, Infosys, Optiv, NTT, Tata Communications,

Verizon and Wipro. The company is also a founding member of the Cloud Security Alliance (CSA).

# B Changes

This appendix tracks the evolution of *OpenSSL Cookbook* over time. If all you need is a quick overview of the changes, you will find here everything you need to know.

## v1.0 (May 2013)

First release.

## v1.1 (October 2013)

Changes in this version:

- Updated *SSL/TLS Deployment Best Practices* to v1.3. This version brings several significant changes: *(1)* RC4 is deprecated, *(2)* the BEAST attack is considered mitigated server-side, *(3)* Forward Secrecy has been promoted to its own category. There are many other smaller improvements throughout.
- Reworked the cipher suite configuration example to increase focus on Forward Security, making it more relevant.
- Discussed all three key types (RSA, DSA, and ECDSA) and explained when the use of each type is appropriate. Added new text to explain how to generate DSA and ECDSA keys.
- Marked cipher suite configuration keywords that were introduced in the OpenSSL 1.x branch.

Thanks to Michael Reschly, Brian Howson, Christian Folini, Karsten Weiss, and Martin Carpenter for their feedback.

## v2.0 (March 2015)

Changes in this version:

- Added Chapter 2, *Testing with OpenSSL*, another one taken from *Bulletproof SSL and TLS*. This chapter focuses on secure server assessment.
- Added the section called "Recommended Configuration", which contains a list of recommended cipher suites. I now prefer to configure OpenSSL by explicitly listing all the suites I wish to enable.
- Added the section called "Creating a Private Certification Authority", which contains a step-by-step guide to creating and deploying a private CA.
- Updated *SSL/TLS Deployment Best Practices* to v1.4. Important changes in this version include SHA1 deprecation and SSL 3 weaknesses (POODLE).

Thanks to Stephen N. Henson, Jeff Kayser, and Olivier Levillain for their feedback.

## v2.1 (March 2016)

Changes in this version:

- Comprehensive update of Appendix A, *SSL/TLS Deployment Best Practices*.