# OPENSSL COOKBOOK

A Guide to the Most Frequently Used OpenSSL Features and Commands

From the book

Bulletproof SSL/TLS and PKI

Ivan Ristić

Last update: Sat Oct 05 19:50:52 BST 2013 (build 118)

# OpenSSL Cookbook

Ivan Ristić

Feisty Duck
LONDON

**OpenSSL Cookbook**

by Ivan Ristić

Version 1.1, published in October 2013.
Copyright © 2013 Feisty Duck Limited. All rights reserved.

First published in May 2013.

ISBN: 978-1907117053

**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.

This book is intended primarily for OpenSSL users who need to perform routine tasks of key and certificate generation and configure programs that rely on OpenSSL for SSL/TLS functionality. The majority of the Internet is powered by open source products, and virtually all of those projects rely on OpenSSL. Apache `httpd` has long been a favorite, but it's now being pursued by `nginx`, which is increasingly gaining in popularity. And, even in the Java camp, Apache Tomcat performs better when coupled with OpenSSL, replacing the native Java implementation of SSL/TLS.

This book is—at least at the moment—built around a chapter from my third book, *Bulletproof SSL/TLS and PKI*. I'm still writing it. I've decided to make the OpenSSL chapter free because this very successful open source project is not very well documented and because the documentation that you can find on the Internet is often wrong and outdated. Thus, good documentation is in great demand.

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 it by not only making the OpenSSL chapter free, but also by committing to continue to maintain and improve it over time. I hope that the fates of *OpenSSL Cookbook* and *Bulletproof SSL/TLS and PKI* become closely intertwined, leading to more free content.

## 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 at *@ivanristic*.

## About Bulletproof SSL/TLS and PKI

*Bulletproof SSL/TLS and PKI*, the book that I'm working on at the moment, is the book I wish I had back when I was starting to get involved with SSL. I don't remember when I started using SSL, but I do remember that when I was writing my first book, *Apache Security*, I began to appreciate the complexities of cryptography. I even began to like it. Before that point I thought that SSL was simple, but then I realized how vast the world of crypto actually is.

In 2009 I began to work on SSL Labs, and for me, the world of cryptography began to unravel. Fast-forward a couple of years, and in 2013 I still feel like I'm only starting. Cryptography is a unique field in which the more you learn, the less you know.

In supporting the SSL Labs users over the years, I've realized that there's a lot of documentation on SSL/TLS and PKI but also that it suffers from two problems: (1) it's not documented in one place, so the little bits and pieces (e.g., RFCs) are difficult to find, and (2) it tends to be very detailed and low level. It took me years of work and study to begin to understand the entire ecosystem.

*Bulletproof SSL/TLS and PKI* aims to address the documentation gap, as a very practical book that first paints the whole picture and then proceeds to discuss the bits and pieces that you need in daily work, going as deep as needed to explain what you need to know.

## 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 two books, *Apache Security* and *ModSecurity Handbook*, 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 Cookbook

OpenSSL is an open source project that consists of a cryptographic library and an SSL 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 toolkit called BSAFE SSL-C at RSA Australia.

Today, OpenSSL is ubiquitous on the server side and in many client tools. Interestingly, browsers tend to use other libraries. The command-line tools provided by OpenSSL are most commonly used to manage keys and certificates.

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.

## 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 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.

### 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 TLS 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[2] is based on OpenSSL 1.0.1c. At the time of this writing, the full name of the package is `openssl 1.0.1-4ubuntu5.5`, and it adds 44 patches to OpenSSL 1.0.1c.

[2] Ubuntu: "openssl" source package in Precise

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

```
$ openssl version -a
OpenSSL 1.0.1 14 Mar 2012
built on: Wed May 23 00:01:41 UTC 2012
platform: debian-amd64
options:  bn(64,64) rc4(16x,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 SSL-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 custom *Certificate Authority* (CA).

## Building OpenSSL

In most cases, you will be using the operating system–supplied version of OpenSSL, but sometimes there are good reasons to upgrade. For example, your current server platform may still be using OpenSSL 0.9.x, and you might want to support newer protocol versions (available only in OpenSSL 1.0.1). Further, the newer versions may not have all the features you need. For example, on Ubuntu 12.04 LTS, there's no support for SSL 2.0 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 2.0 support.

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

```
$ wget http://www.openssl.org/source/openssl-1.0.1c.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 an another location. For example:

```
$ ./config --prefix=/opt/openssl --openssldir=/opt/openssl
```

You can then follow with:

```
$ 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.

## 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://mxr.mozilla.org/mozilla/source/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 is also smart enough to perform new conversations only when data is updated.

## Conversion Using Go

If you prefer the Go programming language, Adam Langley has a conversion tool in it that you can find on 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://mxr.mozilla.org/mozilla/source/security/nss/lib/ckfw/builtins↩
/certdata.txt?raw=1 --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, but otherwise the conversion worked.

## 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 SSL keys everyone uses RSA, because DSA keys are effectively limited to 1024 bits (Windows 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, 2048-bit RSA keys are considered secure, and that's what you should use. Aim also to use 2048 bits for DSA keys and at least 224 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. This, although not ideal, is much better than using unprotected keys. If you need better security, you should invest in a hardware solution.[3]

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 65537 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,

[3] 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.

> which is why generally everyone recommends that you stick with 65537. 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 ASCII:

```
$ cat fd.key
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: AES-256-CBC,717D24945A0CA95E2800B026D9D431CC

vERmFJzsLeAEDqWdXX4rNwogJp+y95uTnw+bOjWRw1+O1qgGqxQXPtH3LWDUz1Ym
mkpxmIwlSidVSUuUrrUzIL+V21EJ1W9iQ71SJoPOyzX7dYX5GCAwQm9Tsb40FhV/
[21 lines removed...]
4phGTprEnEwrffRnYrt7khQwrJhNsw6TTtthMhx/UCJdpQdaLW/TuylaJMWL1JRW
i321s5me5ej6Pr4fGccNOe7lZK+563d7v5znAx+Wo1C+F7YgF+g8LOQ8emC+6AVV
-----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 -tex -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 generate the corresponding public key, 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: ****************
```

The public key is much shorter than the private key:

```
$ cat fd-public.key
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAnlccwQ9FRyJYHM8sFNsY
PUHJHJzhJdwcS7kBptutf/L6OvoEAzCVHi/m0qAA4QM5BziZgnvv+FNnE3sgE5pz
iovEHJ3C959mNQmpvnedXwfcOIlbrNqdISJiP0js6mDCzYjSO1NCQoy3UpYwvwj7
0ryR1F+abARehlts/Xs/PtX3VamrljiJN6JNgFICy3ZvEhLZEKxR7oob7TnyZDrj
IHxBbqPNzeiqLCFLFPGgJPa0cH8DdovBTesvu7wr/ecsf8CYyUCdEwGkZh9DKtdU
HFa9H8tWW2mX6uwYeHCnf2HTw0E8vjtOb8oYQxlQxtL7dpFyMgrpPOoOVkZZW/P0
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 while avoiding 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 see the full list if you specify 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`.

## 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.

CSR creation is usually an interactive process that takes the private server key as input. 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, *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 actually quite rare to find a CA that relies on this field, however. 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 him or her 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
```

## 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 the next section).

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 = distinguished_name

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

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

```
$ openssl req -new -config fd.cnf -key fd.key -out fd.csr
Enter pass phrase for fd.key: ****************
```

You'll be asked for the passphrase only if you used one during key generation.

## Signing Your Own Certificates

If you're installing an SSL server for your own use, you probably don't want to go to a CA to get a publicly trusted certificate. It's much easier to sign your own. The fastest way to do this is to generate 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
```

## Creating Certificates Valid for Multiple Hostnames

With OpenSSL, by default, generated certificates have only one common name and are thus valid for only one hostname. Therefore, if you have several web sites, you must generate a separate certificate for each site. When the same person or group of people maintain several web sites, it's fine to use only one certificate for all of the sites; this is what multidomain certificates are for. Actually, even if you're running a single web site, you should 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, whatever common names are used in the distinguished name are ignored. Newer certificates may not even include a common name. 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

Most of the time, certificates appear to use what are essentially random arrays of bytes. But they contain a great deal of information; you just need to know how to unpack it. The `x509` command does just that, so let's look at our self-signed certificates.

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 SSL, 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 SSL, 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) can use the information provided in this extension to fix an incomplete certificate chain, but many don'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. Certificate 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 is 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.

```
    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, the Apache web server requires 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 certificate 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 a protected server key with the corresponding certificate as well as the intermediate certificates. It's commonly seen with `.p12` and

`.pfx` extensions. This format is commonly used in Microsoft products. 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:

```
$ openssl pkcs12 -export -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 Autho
-----BEGIN CERTIFICATE-----
```

```
MIIF5zCCBM+gAwIBAgIHBG9JXlv9vTANBgkqhkiG9w0BAQUFADCB3DELMAkGA1UE
BhMCVVMxEDAOBgNVBAgTB0FyaXpvbmExEzARBgNVBAcTClNjb3R0c2RhbGUxJTAj
[...]
```

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 SSL deployment. The first is cipher suite configuration, in which you specify which of the many suites available in SSL 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 SSL 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 "RC4-SHA:HIGH:!aNULL"
```

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 bits:

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 SSL 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. 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 v1.0, SSL v3.0, or SSL v2.0 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 and SHA2 (v1.0+). |
| `SHA256` (v1.0+) | Cipher suites using SHA-256. |
| `SHA384` (v1.0+) | Cipher suites using SHA-384. |

**Note**

TLS 1.2 introduced support for authenticated encryption, which bundles encryption with integrity checks. 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, even though their names include `SHA256` and `SHA384` suffixes.

Authentication keywords select suites based on the authentication method they use. Today, virtually all public certificates use RSA for authentication. In the future, we should 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. **Not implemented.** |
| `aDSS`, `DSS` | Cipher suites using DSS authentication, i.e., the certificates carry DSS keys. |
| `aECDH` (v1.0+) | Cipher suites that use ECDH authentication. |
| `aECDSA` (v1.0+) | 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+) | Anonymous ECDH cipher suites. **Insecure.** |
| `DH` | Cipher suites using DH (includes ephemeral and anonymous DH). |
| `ECDH` (v1.0+) | Cipher suites using ECDH (includes ephemeral and anonymous ECDH). |
| `EDH` (v1.0+) | Cipher suites using ephemeral DH key agreement. |
| `EECDH` (v1.0+) | Cipher suites using ephemeral ECDH. |
| `kECDH` (v1.0+) | Cipher suites using ECDH key agreement. |
| `kEDH` | Cipher suites using ephemeral DH key agreements (includes anonymous DH). |
| `kEECDH` (v1.0+) | 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+) | 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. |
| `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 the 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,[4] and further issues make the strength as low as 108 bits.[5] This fact makes DES-CBC3-SHA inferior to RC4-SHA and any other 128-bit cipher suite.

## Handling Errors

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

```
$ openssl ciphers -v 'HIGH:@STRENGTH'
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:
```

## 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.

[4] Meet-in-the-middle attack (Citizendium)

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

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,[6] 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
- `eNULL`; no encryption
- `LOW`; low-strength suites
- `3DES`; effective strength of 108 bits
- `MD5`; suites that use MD5
- `EXP`; obsolete export suites

[6] On the Security of RC4 in TLS (Information Security Group at Royal Holloway, University of London, 13 March 2013)

To reduce the number of suites displayed, I'm also going to eliminate all DSA, PSK, SRP, and ECDH suites, because they're used only very rarely:

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

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 !kEDH !PSK !SRP !kECDH
```

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
```

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
```

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
```

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
```

## 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 SSL 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 1.0.1c because I wish to support TLS 1.1 and TLS 1.2; will there be any performance impact? I've downloaded and compiled OpenSSL 1.0.1c for a test. Let's see:

```
$ openssl-1.0.1c speed rsa
[...]
OpenSSL 1.0.1c 10 May 2012
built on: Mon Oct 15 15:13:24 UTC 2012
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   9793.3 131672.5
rsa 1024 bits 0.000327s 0.000020s   3056.0  49734.5
rsa 2048 bits 0.002218s 0.000068s    450.8  14748.9
rsa 4096 bits 0.015814s 0.000255s     63.2   3921.3
```

Apparently, OpenSSL 1.0.1c is almost twice as fast on this server for my use case (2048-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 SSL (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 SSL 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 operation performed on a server and is thus always the first bottleneck. The example number of 1,090 signatures/second tells us that this server can handle about 1,000 brand-new SSL 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 SSL connections per second. I wish I had enough traffic on that server to worry about the performance of SSL.

# A SSL/TLS Deployment Best Practices

*This appendix contains the complete contents of the SSL/TLS Deployment Best Practices document (version 1.3, dated 17 September 2013), which is an SSL Labs publication.*[1] *This document, which I maintain, is a concise, high-level overview of everything you need to know about SSL/TLS deployment. Included here with permission from Qualys.*

## Introduction

SSL/TLS is a deceptively simple technology. It is easy to deploy, and it just works… except that it does not, really. The first part is true—SSL is easy to deploy—but it turns out that it is not easy to deploy *correctly*. To ensure that SSL provides the necessary security, users must put extra effort into properly configuring their servers.

In 2009, we began our work on SSL Labs because we wanted to understand how SSL was used and to remedy the lack of easy-to-use SSL tools and documentation. We have achieved some of our goals through our global surveys of SSL usage, as well as the online assessment tool, but the lack of documentation is still evident. This document is a first step toward addressing that problem.

Our aim here is to provide clear and concise instructions to help overworked administrators and programmers spend the minimum time possible to deploy a secure site or web application. In pursuit of clarity, we sacrifice completeness, foregoing certain advanced topics. The focus is on advice that is practical and easy to understand. For those who want more information, Section 6 gives useful pointers.

[1] SSL/TLS Deployment Best Practices (Qualys SSL Labs)

## 1. Private Key and Certificate

The quality of the protection provided by SSL depends entirely on the private key, which lays down the foundation for the security, and the certificate, which communicates the identity of the server to its visitors.

### 1.1. Use 2048-bit Private Keys

Use 2048-bit RSA or equivalent-strength ECDSA private keys for all your servers. Keys of this strength are secure and should stay secure for a considerable amount of time. If you already have 1024-bit RSA keys in production, replace them with stronger keys as soon as possible. If you believe that you need more than 2048 bits of security, give more consideration to ECDSA keys, which have better performance characteristics.

### 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 the arrangements practical. Recommended policies include the following:

- Generate private keys and Certificate Signing Requests (CSRs) on a trusted computer. Some CAs offer to generate keys and a CSRs for you, but that's inappropriate.
- Password-protect keys to prevent compromise when they are stored in backup systems.
- After compromise, revoke old certificates and generate new keys to use with new certificates.
- Renew certificates every year and always with new private keys.

### 1.3. Ensure Sufficient Hostname Coverage

Ensure that your certificates cover all the names you wish to use with a site. For example, your main name is *www.example.com*, but you may also have *www.example.net* configured. Your goal is to avoid invalid certificate warnings, which will confuse your users and weaken their trust.

Even when there is only one name configured on your servers, remember that you cannot control how your users arrive at the site or how others link to the site. In most cases, you should ensure that the certificate works with and without the *www* prefix (e.g., for both *example.com* and *www.example.com*). The rule of thumb is this: 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 should be avoided if using them means exposing the underlying keys to a larger group of people, and especially if crossing organizational boundaries. In other words, the fewer people who have access to the private keys, the better.

## 1.4. Obtain Certificates from a Reliable CA

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

**Security posture**
: All CAs undergo regular audits (otherwise they would not be able to operate as CAs), 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 reacted to compromises and if they learned from their mistakes.

**Substantial market share**
: A CA that meets this criterion will most likely make it through security incidents, which wasn't the case with some smaller ones in the past.

**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 minimum, your selected CA should provide support for both Certificate Revocation List (CRL) and Online Certificate Status Protocol (OCSP) revocation and provide an OCSP service with good performance. They should offer both domain-validated and Extended Validation certificates, ideally with your 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 business that will give you good tools to manage them.

**Support**
: Choose a business that will give you good support if and when you need it.

# 2. Configuration

With correct SSL 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. Deploy with Complete and Valid Certificate Chains

In most deployments, the server certificate alone is insufficient; two or more certificates are needed to establish a complete chain of trust. A common problem is configuring the server certificate correctly but forgetting to include other required certificates. Further, although these other certificates are typically valid for longer periods of time, they too expire, and when they do, they invalidate the entire chain. Your CA should be able to provide you with all the additional certificates required.

An invalid certificate chain renders the actual server certificate invalid and results in browser warnings. In practice, this problem is sometimes difficult to diagnose because some browsers can deal with these problems and reconstruct a complete correct chain, and some cannot.

## 2.2. Use Only 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. Of these:

- SSL v2 is insecure and must not be used.
- SSL v3 is very old and obsolete. Because it lacks some key features and because virtually all clients support TLS 1.0 and better, you should not support SSL v3 unless you have a *very* good reason.
- TLS v1.0 is largely still secure; we do not know of major security flaws when they are used for protocols other than HTTP. When used with HTTP, it can *almost* be made secure with careful configuration.
- TLS v1.1 and v1.2 are without known security issues.

TLS v1.2 should be your main protocol. This version is superior because it offers important features that are unavailable in earlier protocol versions. If your server platform (or any intermediary device) does not support TLS v1.2, make plans to upgrade at an accelerated pace. If your service providers do not support TLS v1.2, require that they upgrade.

In order to support older clients, you need to continue to support TLS v1.0 and TLS v1.1 for the time being. With some workarounds (explained in subsequent sections), these protocols can still be considered secure enough for most web sites.

## 2.3. Use Only 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), as well as exchanging data securely. In SSL and TLS, cipher sites are used to 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.

Your goal should be thus to use only suites that provide authentication and encryption of 128 bits or stronger. Everything else must be avoided:

- Anonymous Diffie-Hellman (ADH) suites do not provide authentication.
- NULL cipher suites provide no encryption.
- Export key exchange suites use authentication that can easily be broken.
- Suites with weak ciphers (typically of 40 and 56 bits) use encryption that can easily be broken.
- RC4 is weaker than previously thought.[2] You should remove support for this cipher in the near future.
- 3DES provides only 108 bits of security (or 112, depending on the source), which is below the recommended minimum of 128 bits. You should remove support for this cipher in the near future.

## 2.4. Control Cipher Suite Selection

In SSL v3 and later versions, clients submit a list of cipher suites that they support, and servers choose one suite from the list to negotiate a secure communication channel. Not all servers do this well, however; some will select the first supported suite from the list. Having servers select the right cipher suite is critical for security (more about that in Section 2.7).

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

## 2.5. Support Forward Secrecy

*Forward Secrecy*[3] is a protocol feature that enables secure conversations that are not dependent on the server's private key. With cipher suites that do not support Forward Secrecy, someone who can recover a server's private key can decrypt all earlier encrypted conversations if they have them recorded. 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.[4]

## 2.6. Disable Client-Initiated Renegotiation

In SSL/TLS, renegotiation allows parties to stop exchanging data in order to renegotiate how the communication is secured. There are some cases in which renegotiation needs to be initiated by the server, but there is no known need for clients to do so. Further, client-initiated renegotiation may make your servers easier to attack using *Denial of Service* (DoS) attacks.[5]

## 2.7. Mitigate Known Problems

Nothing is perfectly secure, and at any given time there may be issues with the security stack. It is good practice to keep an eye on what happens in the security world and to adapt to situations as necessary. At the very least, you should apply vendor patches as soon as they become available.

The following issues require your attention:

**Disable insecure renegotiation**

In 2009, the renegotiation feature was found to be insecure and the protocols needed to be updated.[6] Most vendors have issued patches by now or, at the very least, provided workarounds for the problem. Insecure renegotiation is dangerous because it is easy to exploit and has effects similar to Cross-Site Request Forgery (CSRF) and, in some cases, Cross-Site Scripting (XSS).

**Disable TLS compression**

In 2012, the CRIME attack[7] showed how information leakage introduced by TLS compression can be used by attackers to uncover parts of sensitive data (e.g., session cook-

[3] Deploying Forward Secrecy (Qualys Security Labs; 25 June 2013)

[4] Increasing DHE strength on Apache 2.4.x (Ivan Ristić's blog; 15 August 2013)

[5] TLS Renegotiation and Denial of Service Attacks (Qualys Security Labs Blog, 31 October 2011)

[6] SSL and TLS Authentication Gap Vulnerability Discovered (Qualys Security Labs Blog; 5 November 2009)

[7] CRIME: Information Leakage Attack against SSL/TLS (Qualys Security Labs Blog; 14 September 2012)

ies). Very few clients supported TLS compression then (and even fewer support it now), which means that it is unlikely that you will experience any performance issues by disabling TLS compression on your servers. Attacks against TLS compression are of low risk.

**Mitigate information leakage stemming from HTTP compression**

Two variations of the CRIME attack were disclosed in 2013. Rather than focus on TLS compression (which is what CRIME did), TIME and BREACH attacks focus on secrets in HTTP response bodies compressed using HTTP compression. Given that HTTP compression is very important to a great many companies, these problems are more difficult to address. Mitigation might require changes to application code.[8]

TIME and BREACH attacks require significant resources to carry out. But if someone is motivated enough to use them, the impact is equivalent to CSRF.

**Disable RC4**

The RC4 cipher suite is considered insecure and should be disabled. At the moment, the best attacks we know require millions of requests, a lot of bandwidth, and time. Thus, the risk is still relatively low, but we expect that the attacks will improve in the future.

**Be aware of the BEAST attack**

The 2011 BEAST attack[9] targets a 2004 vulnerability in TLS 1.0 and earlier protocol versions, previously thought to be impractical to exploit. For a period of time, server-side mitigation of the BEAST attack was considered appropriate, even though the weakness is on the client side. Unfortunately, to mitigate server-side requires RC4, which we now recommend disabling. Because of that, and because the BEAST attack is by now largely mitigated client-side, we no longer recommend server-side mitigation.[10]

The impact of a successful BEAST attack is similar to that of session hijacking.

## 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. However, because SSL configuration does not usually have a significant overall performance impact, we are limiting the discussion in this section to the common configuration problems that result in serious performance degradation.

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

[9] Mitigating the BEAST attack on TLS (Qualys Security Labs Blog; 17 October 2011)

[10] Is BEAST Still a Threat? (Qualys Security Labs; 10 September 2013)

### 3.1. Do Not Use Too-Strong Private Keys

The cryptographic handshake, which is used to establish secure connections, is an operation whose 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 keys stronger than 2048 bits is a waste of CPU power and is likely to impair user experience.

### 3.2. Ensure That Session Resumption Works Correctly

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 Persistent Connections (HTTP)

These days, most of the overhead of SSL comes not from the CPU-hungry cryptographic operations but from network latency. An SSL handshake is performed after the TCP handshake completes; it requires a further exchange of packets. To minimize the cost of latency, you enable HTTP connection persistence (Keep-Alive), allowing your users to submit many HTTP requests over a single TCP connection.

### 3.4. Enable Caching of Public Resources (HTTP)

When communicating over SSL, browsers assume that all traffic is sensitive. They will typically use memory to cache certain resources, but once you close the browser, all the content may be lost. To get a performance boost and enable long-term caching of some resources, mark public resources (e.g., images) as public by attaching the `Cache-Control: public` response header to them.

## 4. Application Design (HTTP)

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, as well as ways to use them securely.

## 4.1. Encrypt 100% of Your Web Site

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

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

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. Avoid Mixed Content

Mixed-content pages are those that are transmitted over SSL but include resources (e.g., JavaScript files, images, or CSS files) that are not transmitted over SSL. 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 your sites use, and either remove them, replace them with safer alternatives, or accept the risk of their continued use.

## 4.4. Secure Cookies

To be properly secure, a web site requires SSL but also requires that all its cookies are marked as secure. 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.

## 4.5. Deploy HTTP Strict Transport Security

*HTTP Strict Transport Security* (HSTS) is a safety net for SSL: it was designed to ensure that security remains intact even in the case of configuration problems and implementation errors. To activate HSTS protection, you set a single response header in your web sites. After that, browsers that support HSTS (at this time, Chrome, Firefox, and Opera) will 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 plain-text links to secure ones. As a bonus, it also disables click-through SSL certificate warnings. (SSL 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 SSL security of your web sites. New sites should always be designed with HSTS in mind and the old sites converted to support it whenever possible.

## 4.6. Disable Caching of Sensitive Content

The goal of this recommendation is to ensure that sensitive content is communicated to only the intended parties and that it is treated as sensitive. 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.

## 4.7. Ensure That There Are No Other Vulnerabilities

This item is a reminder that SSL does not equal security. SSL 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 (e.g., SQL injection, Cross-Site Scripting, and so on). 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 and ensure that you start out secure, and then periodically thereafter to ensure that you stay secure. For public web sites, our free online assessment tool on the SSL Labs web site is hard to beat. The *Handshake Simulation* feature in particular is very useful, because it shows exactly what security parameters would be used by a variety of commonly used SSL clients.

## 6. Advanced Topics

The following advanced topics are 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.

**Extended Validation certificates**

*Extended Validation* (EV) certificates are high-assurance certificates issued only after thorough offline checks.[11] Their purpose is to provide a strong connection between an organization and its online identity. EV certificates are more difficult to forge, provide slightly better security, and are better treated when browsers present them to end users.

**Public key pinning**

*Public key pinning* is designed to give web site operators the means to restrict which Certificate Authorities can issue certificates for their web sites. This feature has been deployed by Google for some time now (it's hard-coded into their browser, Chrome) and has proven to be very useful in preventing attacks and making the public aware of them. Two proposals are currently being developed: *Public Key Pinning Extension for HTTP*, by the Web Security Working Group, and *Trust Assertions for Certificate Keys*, by Marlinspike and Perrin.

**ECDSA private keys**

Virtually all web sites rely on RSA private keys. This algorithm is thus the key to the security of the Web, which is why attacks against it continue to improve. We are currently transitioning from 1024-bit to 2048-bit RSA keys for that very reason. There are some concerns, however, that further key length increases might lead to performance

[11] About EV SSL Certificates (CA/Browser Forum web site)

issues. Elliptic Curve cryptography uses different math and provides strong security assurances at smaller key lengths. RSA keys can be replaced with ECDSA. They are currently supported by only a small number of CAs, but we expect that most will offer them in the future.

**OCSP Stapling**
*OCSP Stapling* is a modification of the OCSP protocol that allows revocation information to be bundled with the certificate itself and thus served directly from the server to the browser. As a result, the browser does not need to contact OCSP servers for out-of-band validation, which results in better performance.

## Changes

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

### Version 1.3 (17 September 2013)

The following changes were made in this version:

- Recommend replacing 1024-bit certificates straightaway.
- 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.

## Acknowledgments

Special thanks to Marsh Ray (PhoneFactor), Nasko Oskov (Google), Adrian F. Dimcev, and Ryan Hurst (GlobalSign) for their valuable feedback and help in crafting this document. Also thanks to many others who generously share their knowledge of security and cryptography with the world. The words in this document might be mine, but the guidelines given draw on the work of the entire security community.

## About SSL Labs

SSL Labs 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 SSL configuration and SSL Pulse, a monthly scan of about 170,000 most popular SSL-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 6,000 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 Accuvant, BT, Dell SecureWorks, Fujitsu, NTT, Symantec, 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. For a quick overview, you will find here everything you need to know. If you want to go deeper, I suggest that you visit the online version of this book, where you will find our unique *diff view*, which shows all the changes between two versions of the text.

## v1.0 (May 2013)

First release.

## v1.1 (October 2013)

- 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 explain when the use of each type is appropriate. Add 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.