TURING

OpenSSL Cookbook
OpenSSL攻略

[英] Ivan Ristić 著

李振宇 译

人民邮电出版社
北 京

前　言

OpenSSL虽然有很多缺点，但依旧是最成功、最重要的开源项目之一。说它成功是因为它得到了广泛的使用，说它重要是因为大部分互联网基础设施的安全都依赖于它。这一项目包括关键密码学算法的高性能实现，完整的SSL/TLS和PKI栈，以及命令行工具。我认为可以这么说，只要你的工作与安全、Web开发或者系统管理搭边，你就不可避免地需要与OpenSSL打交道。互联网由开源产品统治着，而这些产品几乎全都依赖OpenSSL。

这本小册子包括两种使用OpenSSL的方法。第1章介绍如何执行常规任务（比如密钥和证书的生成），以及如何配置为了SSL/TLS功能而依赖OpenSSL的程序。这一章还会讨论如何创建完整的CA，这对于开发和内网环境会非常有用。第2章主要关注如何使用OpenSSL测试服务器安全。虽然测试有时候会很费时间，但是如果你想知道到底发生了什么，就必须进行这类底层测试。

这两章都截取自《HTTPS权威指南：在服务器和Web应用上部署SSL/TLS和PKI》。我之所以决定将OpenSSL相关章节独立成一本免费的小册子，是因为我发现很多人都需要好的文档。对于OpenSSL来说尤其如此，因为它自身的文档非常差；在互联网上找到的东西要么是过时的，要么是错误的。

除此之外，为了让读者了解《HTTPS权威指南》的内容，出版商还会对外发布一到两章。为了更好地实践这种思路，我想我不仅需要提供免费的OpenSSL章节，而且要持续地维护和改进这本书。因此，就有了这本小册子。

反馈

对这本会一直持续更新的书来说，读者的反馈非常重要。传统的出版物如果再版（技术书籍因为市场太小，所以再版的可能性比较低），读者的反馈需要经过很多年才能反映到新版的书籍里面；而对于这本书，你会发现你的反馈会在几天之内出现在新的版本里面。总之，你发送给我的反馈会影响这本书的演进。

可以通过我的电子邮箱来联系我：ivanr@webkreator.com 。有时候我也会在Twitter上给你回复，我的账号是：@ivanristic。

关于《HTTPS 权威指南》

当我刚开始使用SSL的时候，我非常希望有一本类似《HTTPS权威指南》的书。我记不清那是什么时候了，不过肯定是很早之前，那时候我们还需要给Apache打补丁以便让它支持SSL。不过我还记得，当我在2005年写第一本书*Apache Security*的时候，我开始感激密码学的复杂性，甚至开始喜欢上它了。

2009年，我开始从事于SSL Labs的工作，此时密码学的世界才向我敞开。之后的几年（直到2015年）我依旧在学习。密码学是一个独特的领域，你学的东西越多，就会发现知道的东西越少。

在支持SSL Labs用户的那段时间里，我意识到虽然有很多关于SSL/TLS以及PKI的著作，但是这些材料都存在两个问题：(1) 你需要的内容分布在不同的地方，很难形成整体（例如，RFC），因而难以寻找；(2) 大多数著作都非常详细而且深入底层。很多文档已经废弃。为了让这些材料变得有价值，我花了好几年时间工作和学习，才仅仅开始理解整个生态体系。

《HTTPS权威指南》弥补了这些文档的空白。它是一本经过实践的书，刚开始会有一个简单的介绍以及固定的理论背景，然后会讨论你日常工作中需要的所有东西。它还覆盖了一些密钥相关的知识，例如协议攻击。如果你想了解更多，还有非常多的参考文档以及外部资源可供学习。

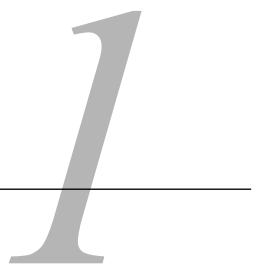
关于作者

Ivan Ristić是一位安全研究员、工程师、作者。他对于Web应用防火墙领域的发展，开源Web应用防火墙ModSecurity的开发，以及在SSL Labs网站上对SSL/TLS和PKI的研究、工具和指南的发表，都作出了很大的贡献，因此享誉世界。

他写过*Apache Security*、*ModSecurity Handbook*和《HTTPS权威指南》三本书，并将其发表到Feisty Duck上。Feisty Duck是他持续写作和发表的平台。Ivan是安全社区的活跃参与者，经常在Black Hat、RSA、OWASP AppSec等各种安全会议上进行演讲。他之前就职于Qualys公司，担任应用安全研究主任。

目　录

第 1 章 OpenSSL



OpenSSL是一个开源项目，包括密码库和SSL/TLS工具集。从项目的官方站点可以看到：

> OpenSSL项目是**安全套接字层**（secure sockets layer，SSL）和**传输层安全**（transport layer security，TLS）协议的一个实现，是大家共同努力开发出的代码可靠、功能齐全、商业级别的开源工具集。项目由遍布世界的志愿者所组成的社区进行管理，他们通过互联网进行沟通、计划和开发OpenSSL工具集以及相关的文档。

OpenSSL在这一领域已经成为事实上的标准，并且拥有比较长的历史。OpenSSL的代码前身是SSLeay[①]，由Eric A. Young和Tim J. Hudson在1995年开发出来。因为后来Eric和Tim停止了SSLeay的更新，转而为澳大利亚的RSA公司开发商业版本的SSL/TLS工具集BSAFE SSL-C，所以OpenSSL项目在1998年的最后几天诞生了。

现在几乎所有的服务器软件和很多客户端软件都在使用OpenSSL，其中基于命令行的工具是进行密钥、证书管理以及测试最常用到的软件了。有意思的是，之前很多使用其他库作为SSL/TLS解析的浏览器正发生一些变化，例如Google正在将Chrome迁移到他们自己的OpenSSL分支BoringSSL。[②]

OpenSSL是基于OpenSSL和SSLeay双许可证下的授权模式，两种许可都类似于BSD，同时有一个建议的条款。OpenSSL的许可在很长一段时间里都是争论的焦点，因为其中任何一个许可都与GPL家族不兼容。因此，你会发现那些基于GPL许可的程序比较偏向于使用GnuTLS。

1.1 入门

如果你使用的操作系统是基于Unix的，那么几乎可以肯定上面已经安装了OpenSSL，我们的入门步骤也会变得非常简单。唯一可能的问题是版本不是最新的。在本节中，我假设你已经在使用Unix平台，因为OpenSSL天生就基于Unix平台。

Windows的用户就稍微麻烦了，最简单的情况是你仅仅需要使用OpenSSL的命令行工具，可

---

① 名称SSLeay中的eay是Eric A. Young的首字母缩写。

② BoringSSL，https://www.chromium.org/Home/chromium-security/boringssl（Chromium，检索于2015年6月30日）。

以在OpenSSL的网站上找到Shining Light Production的链接[①]，它拥有已经编译好的Windows版本的OpenSSL。如果你想要OpenSSL的所有功能，需要确保那些编译好的OpenSSL都是基于同一个版本编译出来的，否则可能会遇到难以解决的崩溃问题。最好的办法是使用一个包含你需要的所有程序的软件包。例如，如果你想要在Windows上运行Apache，那么可以直接从Apache Lounge获取对应的二进制文件。[②]

1.1.1　确定 OpenSSL 版本和配置

在开始之前，你要首先确定当前使用的OpenSSL版本。下面是我在Ubuntu 12.04 LTS上面运行`openssl vesion`获得的版本信息，后面所有的演示都是基于这个环境：

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

在写这本书的时候，OpenSSL 0.9.x正在向OpenSSL 1.0.x发展。1.0.1最大的意义在于它是第一个同时支持TLS 1.1和1.2的版本。总体来说，支持新协议是大趋势，所以我们还要经历一段新老协议不互通的过程。

> **注意**
> 不同的操作系统经常会修改OpenSSL的代码，主要是修复一些已知漏洞。然而项目的名称和版本号经常保持原样，也没有任何迹象表明代码其实是原项目的一个分支，一些行为已经变化了。例如现在使用的Ubuntu 12.04 LTS的OpenSSL是基于1.0.1c版本，全名是`openssl 1.0.1-4ubuntu5.16`，包含了很多已知问题的补丁。[③]

可以使用-a开关获取完整的版本信息：

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

---

① Win32 OpenSSL，http://slproweb.com/products/Win32OpenSSL.html（Shining Light Productions，检索于2014年7月3日）。

② Apache 2.4 VC14 Binaries and Modules，http://www.apachelounge.com/download/（Apache Lounge，检索于2015年7月15日）。

③ Precise版本的OpenSSL源代码包，https://launchpad.net/ubuntu/precise/+source/openssl（Ubuntu，检索于2014年7月3日）。

上面最后一行的输出（/usr/lib/ssl）是OpenSSL默认情况下查找配置和证书的目录。在我的系统里，该位置是/etc/ssl的别名（也就是软链接），Ubuntu会在其中保存与TLS相关的文件：

```
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
```

misc/目录包含一些补充脚本，其中最有用的脚本允许你实现一个私有的证书颁发机构。

1.1.2 构建 OpenSSL

大多数情况下使用操作系统默认提供的OpenSSL就够了，但最好还是升级到最新版本。例如你当前的系统还在使用OpenSSL 0.9.x，而你想使用更新的TLS协议（在OpenSSL 1.0.1以上版本中才有）。当然，新版本的OpenSSL可能无法提供你想要的所有功能，例如在Ubuntu 12.04 LTS上的openssl s_client命令就不支持SSL 2。虽然默认不支持SSL 2是对的，但是如果需要测试别的服务器是否支持SSL 2，我们就需要这类功能。

你可以先下载最新版本的OpenSSL（我使用的是1.0.1h）：

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

在编译之前我们还需要进行配置。一般情况下，我们配置的安装目录不要与系统默认提供的OpenSSL目录一样，例如：

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

enable-ec_nistp_64_gcc_128参数可以让我们使用优化后的一些常用的椭圆曲线算法，这个优化基于编译器的一些特性，默认情况下会关闭这些特性，而且无法自动检测。

然后执行下面这些命令：

```
$ make depend
$ make
$ sudo make install
```

然后在/opt/openssl目录下面，我们可以看到：

```
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
```

private/目录默认是空的，因为我们还没有生成任何私钥，certs目录也是如此。OpenSSL不包括任何根证书，因为维护一个可信证书库不在OpenSSL项目的范围之内。幸运的是，很多操作系

统可能已经安装了可信证书库，当然你也可以自己创建一个，1.1.4节会具体介绍。

> **注意**
> 编译软件之前，熟悉编译器的默认配置是很重要的。系统提供的软件一般都会尽可能地使用到编译器的优化选项，但是如果你自己编译的话，就不一定能保证这些优化选项都能用到。[①]

1.1.3　查看可用命令

OpenSSL包含了很多密码相关的工具。我计算了一下，我的版本大概有46个，简直可以称为密码学领域的瑞士军刀。即便现在你只会用到其中的一部分工具，但是还是有必要熟悉所有的工具，这样在将来需要的时候才能知道你有哪些工具可以使用。

OpenSSL没有专门的help关键字，任何时候输入OpenSSL无法识别的命令，就会显示帮助文本：

```
$ 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
```

第一部分帮助列出了所有可以使用的工具。如果对于某个命令，想获取更加详细的信息，可以使用man加上工具的名称。例如man ciphers会告诉我们密码套件是如何配置的。

帮助信息不止上面这些，但是剩下的就没这么有意思了，在第二部分的帮助信息里我们可以看到可用的消息摘要命令：

```
Message Digest commands (see the `dgst' command for more details)
md4               md5               rmd160            sha
sha1
```

然后在第三部分的帮助信息中，我们可以看到所有的加密命令：

```
Cipher commands (see the `enc' command for more details)
aes-128-cbc       aes-128-ecb       aes-192-cbc       aes-192-ecb
```

① compiler hardening in Ubuntu and Debian，https://outflux.net/blog/archives/2014/02/03/compiler-hardening-in-ubuntu-and-debian/（Kees Cook，2014年2月3日）。

```
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
```

1.1.4 创建可信证书库

OpenSSL没有自带可信根证书（也叫作可信证书库），所以如果你是自己从头开始安装的话，那么就需要从别的地方找找了。一种选择是使用操作系统自带的可信证书库，一般来说没有问题，但是这个可信证书库可能不是最新的。更好的一种选择是从Mozilla那里获取，虽然麻烦一点，但是Mozilla花费了很多时间维护一个可靠的可信证书库，例如我为自己在SSL Labs写的评估工具使用的就是这个可信证书库。

因为是开源的，所以Mozilla将可信证书库保存在源代码存储库中：

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

但是Mozilla将这些证书以一种比较特殊的格式存放，其他人很少使用。所以如果不介意的话，我们可以从第三方（例如Curl项目）去获取最新的PEM（privacy-enhanced mail）格式的可信证书库，这种格式可以直接使用：

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

当然，如果你更愿意直接从Mozilla下载可信证书库，也有现成的用Perl或者Go语言编写的脚本，这样你就无需自己编写了。在后面几节我会介绍这两个工具。

> **注意**
>
> 如果你最终决定自己编写解析Mozilla可信证书库脚本的话，需要注意根证书文件中实际包含了两种类型的证书：一部分是可信的，另外一部分是明确不可信的。Mozilla使用这种方式去禁用那些被盗用的中间证书（例如DigiNotar的那些老证书），上面提到的Perl和Go脚本会识别出不可信证书。

1. 使用Perl脚本

Curl项目提供了由Guenter Knauf使用Perl编写的Mozilla可信证书库转换脚本：

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

下载这个脚本之后直接运行就会从Mozilla下载最新的证书数据并且转换为PEM格式：

```
$ ./mk-ca-bundle.pl
```

```
Downloading 'certdata.txt' ...
Processing 'certdata.txt' ...
Done (156 CA certs processed, 19 untrusted skipped).
```

如果还有之前下载过的证书数据，这个脚本会将两份证书数据进行对比，然后只处理更新的部分。

2. 使用Go脚本

如果你更喜欢使用Go语言，那么可以从GitHub下载Adam Langley编写的工具：

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

开始转换之前先下载这个小工具：

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

然后下载Mozilla的证书数据：

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

最后使用下面的命令转换文件：

```
$ 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
```

在我的例子里面有一个无效的证书是Go X.509无法处理的，其他的都正常处理了。

1.2　密钥和证书管理

大多数用户借助OpenSSL是因为希望配置并运行能够支持SSL的Web服务器。整个过程包括3个步骤：(1) 生成强加密的私钥；(2) 创建证书签名申请（certificate signing request，CSR）并且发送给CA；(3) 在你的Web服务器上安装CA提供的证书。这些步骤（还有其他一些）会在本节详细说明。

1.2.1　生成密钥

在使用公钥加密之前的第一步是生成一个私钥。在开始之前，你必须进行几个选择。

- 密钥算法

  OpenSSL支持RSA、DSA和ECDSA密钥，但是在实际使用场景中不是所有密钥类型都适用的。例如对于Web服务器的密钥，所有人都使用RSA，因为DSA一般因为效率问题会限制在1024位（Internet Explorer不支持更长的DSA密钥），而ECDSA还没有被大部分的CA支持。对于SSH来说，一般都是使用DSA和RSA，而不是所有的客户端都支持ECDSA算法。

- 密钥长度

  默认的密钥长度一般都不够安全，所以我们需要指定要配置的密钥长度。例如RSA密钥默认的长度是512位，非常不安全。如果今天你的服务器上还是用512位的密钥，入侵者

可以先获取你的证书，使用暴力方式来算出对应的私钥，之后就可以冒充你的站点了。现在，一般认为2048位的RSA密钥是安全的，所以你应该采用这个长度的密钥。DSA密钥也应该不少于2048位，ECDSA密钥则应该是256位以上。

- *密码*

  强烈建议使用密码去保存密钥，虽然这是只一个可选项。受密码保护的密钥可以被安全地存储、传输以及备份。但与此同时，这样的密钥也会带来不便，因为我们如果没有密码的话，就无法使用密钥了。例如每次你想要重启Web服务器的时候就会被要求输入密码。大多数情况下，这会导致极大的不便，在现实中几乎无法使用。如果真的发生入侵，在生产环境中使用密码保护过的密钥其实并没有提高安全性，因为一旦使用密码解密后，私钥会被明文保存在程序内存中，攻击者如果能登录服务器，那么只需要花费一点时间就可以很容地获取到密钥。所以使用密码方式保护私钥只有在私钥并没有被放在生产环境服务器上的时候才有用。也就是说在生产环境上存放私钥和密码都是可以的。如果你想要让生产环境的私钥更加安全，那么需要考虑一下硬件解决方案。[①]

可以使用genrsa命令来生成RSA密钥：

```
$ 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: ****************
```

这里，我指定私钥会使用AES-128算法来加密保存。当然也可以使用AES-192或者AES-256（分别使用开关-aes192和-aes256），但是最好不要使用其他算法（DES、3DES和SEED）。

> **警告**
>
> 上面输出结果中的e值表示公用指数，默认情况下会被设置为65 537。这是所谓的**短公用指数**（short public exponent），它可以显著提高RSA的验证性能。如果希望验证过程更快，请使用-3开关，选择3作为公用指数。但是历史上使用3作为公用指数有很多弱点，这就是所有人都建议你继续使用65 537的原因，后者被证明是安全和效率的一个平衡点。

私钥以所谓的PEM格式存储，该格式仅包含文本：

```
$ cat fd.key
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
```

① 一小部分组织会有非常严格的安全需求，要求不惜任何代价保证私钥安全。对他们来说解决方案是使用硬件安全模块（hardware security module，HSM），这类产品的设计使得即便能够物理访问它，也无法导出密钥。为实现此事，HSM不仅仅生成和存储密钥，还会执行所有必须的操作（例如，生成签名）。HSM非常昂贵。

```
DEK-Info: AES-128-CBC,01EC21976A463CE36E9DB59FF6AF689A

vERmFJzsLeAEDqWdXX4rNwogJp+y95uTnw+bOjWRw1+O1qgGqxQXPtH3lWDUz1Ym
mkpxmIwlSidVSUuUrrUzIL+V21EJ1W9iQ71SJoPOyzX7dYX5GCAwQm9Tsb4OFhV/
[21 lines removed...]
4phGTprEnEwrffRnYrt7khQwrJhNsw6TTtthMhx/UCJdpQdaLW/TuylaJMWL1JRW
i321s5me5ej6Pr4fGccNOe7lZK+563d7v5znAx+Wo1C+F7YgF+g8LOQ8emC+6AVV
-----END RSA PRIVATE KEY-----
```

乍一看私钥是一堆随机数据，其实不是。你可以使用下面的rsa命令解析出私钥的结构：

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

如果你需要单独查看密钥的公开部分，可以使用下面的rsa命令：

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

如果你查看这个刚生成的文件，就会发现有明显的标识，表示这部分确实是公开的信息：

```
$ cat fd-public.key
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAnlccwQ9FRyJYHM8sFNsY
PUHJHJzhJdwcS7kBptutf/L6OvoEAzCVHi/mOqAA4QM5BziZgnvv+FNnE3sgE5pz
iovEHJ3C959mNQmpvnedXwfcOIlbrNqdISJiPOjs6mDCzYjSO1NCQoy3UpYwvwj7
OryR1F+abARehlts/Xs/PtX3VamrljiJN6JNgFICy3ZvEhLZEKxR7oob7TnyZDrj
IHxBbqPNzeiqLCFLFPGgJPaOcH8DdovBTesvu7wr/ecsf8CYyUCdEwGkZh9DKtdU
```

```
HFa9H8tWW2mX6uwYeHCnf2HTwOE8vjtOb8oYQxlQxtL7dpFyMgrpPOoOVkZZW/PO
NQIDAQAB
-----END PUBLIC KEY-----
```

一个好习惯是验证一下输出内容是否像你所期望的那样。例如，如果你忘记在命令行中包含-pubout开关，那么输出结果就会包括私钥信息而不是只有公钥。

DSA的密钥生成分成两个部分：先生成DSA的参数，然后再生成密钥。当然我倾向于使用一个命令来包括这两个步骤：

```
$ 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: ****************
```

这种方式可以让我生成一个受密码保护的密钥，而不会在磁盘上留下临时文件（DSA参数）或者临时的密钥。

创建ECDSA密钥的过程是类似的，但是不能创建任意长度的密钥。对于每个密钥，你需要选择一个*命名曲线*（named curve），它可以控制密钥长度，同时也限定了椭圆曲线的参数。下面的例子使用secp256r1这个命名曲线创建一个256位长度的ECDSA密钥：

```
$ 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支持非常多的命名曲线（可以使用-list_curves开关获取完整的曲线列表），但是对于Web服务器使用的密钥来说，你只能使用两种：secp256r1（OpenSSL使用prime256v1作为名称）和secp384r1，因为只有这两个是大多数浏览器都支持的。

> **注意**
> 如果你使用OpenSSL 1.0.2，在生成密钥的时候可以使用genpkey命令节省大量的时间，因为它支持各种密钥类型以及配置参数。现在它是统一的密钥生成接口。

1.2.2 创建证书签名申请

一旦有了私钥，就可以创建*证书签名申请*（certificate signing request，CSR）。这是要求CA给证书签名的一种正式申请，该申请包含申请证书的实体的公钥以及该实体的某些信息。该数据将成为证书的一部分。CSR始终使用它携带的公钥所对应的私钥进行签名。

CSR创建的过程一般都是交互式的，你需要提供区分证书所需的不同元素。认真阅读openssl工具的帮助。如果你想让某一个字段为空，不要直接回车，必须输入一个点（.）；如果直接回车，

OpenSSL会直接使用这个字段默认的值（虽然几乎所有人都这么做，但是如果使用的是默认的OpenSSL配置，直接回车没有任何意义。只有在你意识到可以通过直接修改OpenSSL配置或者提供自己的配置文件来修改默认配置的时候，直接回车才是没问题的）。

```
$ 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 []:
```

> **注意**
>
> 根据RFC 2985的5.4.1节[①]，**质询密码**（challenge password）是一个可选字段，用于在证书吊销过程中确认申请过该证书的最初实体的身份。如果输入这个字段，则会将密码包括在CSR文件中并发送给CA。几乎没有CA会依赖这个字段，我所看到的帮助信息都建议将这一字段留空，因为设置一个质询密码并没有增加CSR的任何安全性。另外不要将这个字段和密钥密码混淆了，它们的作用是不一样的。

CSR生成之后，可以使用它去直接进行证书签名或者将它发送给公共CA让他们对证书进行签名。下面会具体讲这两种方式，但是在操作之前，最好再检查一遍CSR是正确的。可以这么做：

```
$ 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:
```

① RFC 2985: PKCS #9: Selected Object Classes and Attribute Types Version 2.0，http://tools.ietf.org/html/rfc2985（M. Nystrom和B. Kaliski，2000年11月）。

```
                [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
```

1.2.3 用当前证书生成 CSR 文件

如果你想更新一张证书并且不想对里面的信息作任何更改，那么实现可以变得简单一些。使用下面的命令可以用当前的证书创建一个全新的CSR文件：

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

> **注意**
> 除非是你在使用某种公钥钉扎的形式并且希望继续使用之前的密钥，否则建议每次申请新证书的时候都生成一个全新的密钥。密钥生成起来很快而且成本低廉，但是可以减少泄露的风险。

1.2.4 非交互方式生成 CSR

生成CSR并不一定要使用交互方式。使用自定义的OpenSSL配置文件，可以将这个过程自动化（本节中介绍的），并且还可做一些交互方式无法完成的事情（后续几节会介绍）。

例如我们想自动生成www.feistyduck.com的CSR文件，可以先创建一个fd.cnf文件：

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

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

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

然后使用下面的命令直接创建CSR文件：

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

1.2.5　自签名证书

如果你只是想安装一台自己使用的TLS服务器,那么可以不必找CA去获取一个公开信任的证书，自己就可以直接签发一个。最简单的方式就是生成自签名证书。如果你使用Firefox，那么可以在第一次访问网站的时候创建一个证书例外，然后就可以像使用公开可信证书一样正常访问了。

如果已经有了CSR，可以使用下面的文件创建证书：

```
$ 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: ****************
```

也可以无需单独创建一个CSR，使用下面的命令直接使用私钥创建自签名证书：

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

如果你不想有交互提示，直接使用-subj并带上标题信息就可以了：

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

1.2.6　创建对多个主机名有效的证书

默认情况下，OpenSSL创建的证书只包含一个公用名而且只能设置一个主机名。因为这个限制，即便你有其他相关联的站点，也不得不为每个站点生成一张单独的证书。在这种情况下，使用一张多域名（multidomain）的证书就有意义了。即便你是维护一个站点，也得确保用户在访问站点的所有子域名的时候证书是有效的。在实际使用中意味着使用至少两个名称，一个是使用www开头的，一个是没有任何前缀的（例如www.feistyduck.com和feistyduck.com）。

有两种方式在一张证书里面支持多主机名。一种方式是在X.509的使用者可选名称（subject alternative name，SAN）扩展字段里面列出所有要使用的主机名；另外一种就是使用泛域名。可以将两种方式合在一起，这样更加方便。在实际使用的时候，可以设置顶级域名和一个泛域名来囊括所有二级域名（例如feistyduck.com和*.feistyduck.com）。

> **警告**
> 当证书包括可选名称的时候，所有公用名就会被忽略。CA新创建的证书甚至可能不再包括任何公用名，所以，请在可选名称列表中包含所有想要的主机名。

首先，将扩展信息放在一个单独的文本文件中，我将该文件命名为fd.ext。在这个文件中，指定扩展的名称（subjectAltName），并且像下面这样列出需要的主机名：

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

然后当使用x509命令签发证书的时候，使用-extfile开关引用该文件：

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

剩下的步骤与之前的一样，当然在检查证书的时候你会发现它包括了SAN扩展信息：

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

1.2.7 检查证书

第一眼看到证书内容的时候会觉得它们就是一堆随机数据，而你只需要知道如何解析它们，就会发现其实里面包含了很多信息。x509命令可以帮助你查看刚生成的自签名证书。

下面的例子中我使用-text来打印证书内容，使用-noout则不打印编码后的证书内容，这样可以减少信息干扰（默认情况下会打印）：

```
$ 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
```

就像上面的例子一样，自签名证书一般只包括最基本的证书数据。相比而言，公共CA签发的证书则含有更多有意义的信息（通过X.509扩展机制），让我们快速了解一下。

基本约束（basic constraint）扩展用于标记证书是否是一个CA，这样的证书可以给其他证书进行签名。非CA证书则没有这个扩展项或者其中CA的值会被设置为FALSE。这是一个关键扩展，意味着所有软件必须识别这个字段。

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

密钥用法（key usage，KU）和扩展密钥用法（extended key usage，EKU）扩展限制了证书的使用范围。如果这两个扩展存在，只有列表里面的使用方式是允许的。如果这个扩展不存在，则没有任何限制。你看到的这个例子是非常典型的Web服务器证书，也就是说它不能进行代码签名：

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

CRL分发点（CRL distribution point）扩展列出了CA证书吊销列表（certificate revocation list，CRL）的地址，当证书需要被吊销的时候这个信息非常重要。CA会对CRL进行签名，并且每隔一段时间发布一次（例如，7天）。

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

> **注意**
> 你可能注意到了CRL的地址没有使用安全服务器，你可能会担心这样的话这个链接是否就不安全了。事实上不会的，因为每个CRL都由所签发的CA进行签名，浏览器可以验证CRL的完整性。事实上如果CRL由TLS进行分发，在验证CRL服务器所提供的证书是否吊销的时候，浏览器就会面临先有鸡还是先有蛋的问题。

证书策略（certificate policy）扩展用来指出证书使用哪种策略签发，例如扩展验证（extended validation，EV）标识就在这里（证书下面这个例子）。每个标识都拥有一个唯一的对象标识符（object identifier，OID），对签发的CA来说它们是唯一的。另外这个扩展一般还包括一个或者多个证书策略声明（certificate policy statement，CPS），一般都是网页或者PDF文档。

```
X509v3 Certificate Policies:
    Policy: 2.16.840.1.114414.1.7.23.3
    CPS: http://certificates.starfieldtech.com/repository/
```

颁发机构信息访问（authority information access，AIA）扩展包括了两个重要信息。首先它列出了CA的在线证书状态协议（online certificate status protocol，OCSP）响应程序的地址，可以用来实时监测证书的吊销情况。这个扩展可能还带上这个证书颁发者的证书地址（也就是证书链的上一层证书）。现在服务器证书已经很少直接使用根证书进行签名了，所以说用户一般需要在他们的配置里面加上一个或者多个中间证书，这时候就很容易出现因为漏了导致证书验证失败。有一些客户端（例如IE）会使用这个信息来获取中间CA的证书，从而弥补服务器忘记配置中间证书的问题，但是大部分的客户端不会。

```
Authority Information Access:
    OCSP - URI:http://ocsp.starfieldtech.com/
    CA Issuers - URI:http://certificates.starfieldtech.com/repository/sf_intermediate.crt
```

使用者密钥标识符（subject key identifier）和颁发机构密钥标识符（authority key identifier）

扩展分别建立了唯一的使用者和颁发机构标识符。证书的颁发机构密钥标识符扩展的信息必须与颁发者的使用者密钥标识符扩展里面的信息一致。这些信息在证书链路径建立过程中相当有用，客户端会试图从分支（服务器）证书开始，寻找到根证书所有可能的路径。证书机构经常一个私钥对应多个证书，而这个字段允许软件可以非常可靠地让证书和密钥对应起来。现实中很多服务器提供的证书链其实都是错误的，但是因为浏览器可以自动寻找到其他可信的路径，所以这种情况常常被忽略了。

```
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
```

最后使用者可选名称（subject alternative name）扩展用来列出所有合法的主机名。这个扩展是可选的，如果不存在，客户端就会使用使用者（subject）字段里面公用名（common name，CN）提供的信息；如果扩展存在，那么在验证过程中CN字段的内容会被忽略。

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

1.2.8 密钥和证书格式转换

私钥和证书可以以各种格式进行存储，所以你可能经常需要进行各种格式之间的转换，最常见的格式如下所示。

- Binary (DER) certificate
  包含原始格式的X.509证书，使用DER ASN.1编码。
- ASCII (PEM) certificate(s)
  包含base64编码过的DER证书，它们以-----BEGIN CERTIFICATE-----开头，以-----END CERTIFICATE-----结尾。虽然有些程序可以允许多个证书存在一个文件中，但是一般来说一个文件只有一张证书。例如Apache Web服务器要求服务器的证书全部在一个文件里面，而中间证书一起放在另外一个文件中。
- Binary (DER) key
  包含DER ASN.1编码后的私钥的原始格式。OpenSSL使用他自己传统的方式创建密钥（SSLeay）格式。还有另外一种不常使用的格式叫作PKCS#8（RFC 5208定义的）。OpenSSL可以使用pkcs8命令进行PKCS#8格式的转换。
- ASCII (PEM) key
  包括base64编码后的DER密钥和一些元数据信息（例如密码的保存算法）。
- PKCS#7 certificate(s)
  RFC 2315定义的一种比较复杂的格式，设计的目的是用于签名和加密数据的传输。一般常见的是.p7b和.p7c扩展名的文件，并且文件里面可以包括所需的整个证书链。Java的密钥管理工具支持这种格式。

- PKCS#12 (PFX) key and certificate(s)
  一种可以用来保存服务器私钥和整个证书链的复杂格式，一般以.p12和.pfx扩展名结尾。这类格式常见于Microsoft的产品，但是也用于客户端证书。虽然很久以前PFX表示PKCS#12之前的版本，现在PFX常被用作PKCS#12的代名词，不过你已经很难遇到老版本了。

1. PEM和DER转换

使用x509工具进行PEM和DER格式之间的证书转换，从PEM转换到DER：

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

从DER转换到PEM：

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

在私钥的DER和PEM格式之间进行转换方式是一样的，但是需要使用rsa或者dsa命令分别用作RSA和DSA密钥。

2. PKCS#12（PFX）转换

只需要一个命令就可以将PEM转换成PKCS#12。下面的例子将密钥（fd.key）、证书（fd.crt）以及中间证书（fd-chain.crt）转换成一个PKCS#12文件：

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

如果想反过来转换就没那么直接了，虽然也可以使用一个命令，但是这样结果也会存在一个文件里面：

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

现在可以用你最喜欢的编辑器打开fd.pem然后手动将其分为独立的密钥、证书和中间证书文件，同时你会发现每一部分的前面都有额外的内容。例如：

```
Bag Attributes
    localKeyID: E3 11 E4 F1 2C ED 11 66 41 1B B8 83 35 D2 DD 07 FC DE 28 76
subject=/1.3.6.1.4.1.311.60.2.1.3=GB/2.5.4.15=Private Organization /↩
serialNumber=06694169/C=GB/ST=London/L=London/O=Feisty Duck Ltd /↩
CN=www.feistyduck.com
issuer=/C=US/ST=Arizona/L=Scottsdale/O=Starfield Technologies, Inc./OU=http:// ↩
certificates.starfieldtech.com/repository/CN=Starfield Secure Certification ↩
Authority
-----BEGIN CERTIFICATE-----
MIIF5zCCBM+gAwIBAgIHBG9JXlv9vTANBgkqhkiG9w0BAQUFADCB3DELMAkGA1UE
BhMCVVMxEDAOBgNVBAgTB0FyaXpvbmExEzARBgNVBAcTClNjb3R0c2RhbGUxJTAj
[...]
```

多出来的这些元数据可以非常方便地用来识别证书。显然你需要确保主证书文件囊括了分支服务器证书而不是别的证书，并且你还得确保中间证书放置的顺序是正确的，签发证书应该在被签名证书的后面。如果你看到了自签名的根证书，可以直接删除或者将它存到别的地方，不应该将其放到证书链里面。

> **警告**
> 除了编码后的密钥和证书，最终转换后的结果不应该包括任何别的内容。虽然有一些工具可以自动忽略那些不需要的部分，但不是所有工具都可以。如果在PEM保留了这些无用的内容，可能会导致后续排查问题的时候遇到困难。

可以直接让OpenSSL来划分不同的部分，不过需要执行多次的pkcs12命令（每次还得输入对应的密码）：

```
$ 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
```

这种方式并没有更简单，你还是得检查每个文件确保它们的内容是正确的，并且已经移除了那些元数据。

3. PKCS#7转换

使用crl2pkcs7命令将PEM转换成PKCS#7格式：

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

将pkcs7命令与-print_certs开关一起使用可以将PKCS#7转换成PEM格式：

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

与PKCS#12一样，你还得手动编辑fd.pem文件并且将其分成不同部分。

1.3 配置

本节我会讨论两个与TLS部署相关的话题。首先是密码套件的配置，你需要从可用的TLS密码套件里面找出你希望用来加密通信的那些套件。这个话题很重要，因为几乎每个使用OpenSSL的程序都会重用它的套件配置，这意味着你只需要学会如何给某一个程序配置密码套件后，别的地方同时也可以适用。第二个话题主要讨论纯加解密操作的性能衡量方式。

1.3.1 选择密码套件

TLS服务器配置中一个共同的任务是选择支持哪些密码套件。依赖OpenSSL的那些程序一般采用与OpenSSL同样的套件配置方式，只需要简单地将配置传递给OpenSSL就可以了。例如在Apache httpd中，它的密码套件配置可能类似于这样：

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

第一行控制密码套件的优先级（同时让httpd主动选择套件）。

第二行控制支持的套件列表。

因为有很多细节要考虑，所以要想配置合理的套件需要花费不少的时间。最好的方式是使用OpenSSL的ciphers命令来确定每个套件对应的配置。

1. 获取所支持的套件列表

在开始配置之前，需要先确定你安装的OpenSSL版本支持哪些套件。可以调用带有-v开关和ALL:COMPLEMENTOFALL参数的ciphers命令来列出所有套件（显然这里ALL并不是真正意义上的“所有”）：

```
$ 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...]
```

> **提示**
>
> 如果使用的是OpenSSL 1.0.0之后的版本，可以使用大写的-V开关来获取更详细的输出内容。这种模式下的输出内容会包括套件的ID，非常方便。例如OpenSSL不一定使用RFC标准的套件名称，在这种情况下就需要通过ID进行二次确认。

我这里一共输出了111个套件，每一行都包含一个套件的信息和下面这些信息：

(1) 套件名称

(2) 最低的TLS版本

(3) 密钥交换算法

(4) 密钥验证算法

(5) 对称加密算法和长度

(6) 消息摘要（完整性检查）算法

(7) 出口套件指示符

如果不使用ALL:COMPLEMENTOFALL而是使用别的密码套件参数，OpenSSL就只会列出对应的套件。例如你可以只显示基于RC4的那些密码套件，如下所示。

```
$ 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
```

虽然有一些套件是不安全的，OpenSSL还是会把你要求的套件全部输出来，当然你还得从这些配置里面筛选出安全的套件。而且套件的输出顺序也是有含义的，当你的TLS服务器设置为手动选择密码套件时（这是最好的方式，也应该这么设置），越靠前的套件优先级越高。

2. 关键字

密码套件关键字（keyword）是密码套件配置的最基本的组成部分，每个套件名称（例如，RC4-SHA）对应一个套件。其他关键字则根据某些标准对应一组套件。关键字名称是区分大小写的。我本来应该直接向你呈现记录有一堆复杂关键字的OpenSSL文档，但是发现加密文档不是最新的，缺少了最近新增的内容。所以我会在本节尝试记录下所有关键字。

组关键字是选择常用密码套件的一种快捷方式，例如HIGH只会列出安全性非常强的密码套件，如表1-1所示。

表1-1 组关键字

关 键 字	含 义
DEFAULT	默认的密码列表。这是在编译的时候确定的，OpenSSL 1.0.0的默认值一般是ALL:!aNULL:!eNULL。这一定是第一个特定的密码字符串
COMPLEMENTOFDEFAULT	这些密码包含在ALL里面，但是默认是未启用的。当前是ADH。需要注意的是这条规则不包含eNULL，也同时没有被ALL所囊括在内（如果有必要，请使用COMPLEMENTOFALL）
ALL	除了eNULL密码之外的所有密码套件，需要显式启用才可以使用
COMPLEMENTOFALL	不包含在ALL的密码套件中，当前是eNULL
HIGH	“High”加密密码套件。当前是指密钥长度超过128位的密码套件
MEDIUM	“Medium”加密密码套件，当前是指那些使用128位的加密算法
LOW	“Low”加密密码套件，当前是指使用64或者56位的加密算法，但是不包含那些出口的密码套件。不安全
EXP、EXPORT	出口加密算法。包含40和56位的算法。不安全
EXPORT40	40位出口加密算法。不安全
EXPORT56	56位出口加密算法。不安全
TLSv1、SSLv3、SSLv2	分别是TLS 1.0、SSL 3或者SSL 2支持的密码套件

摘要关键字会按照特定的摘要算法筛选出密码套件。例如MD5表示筛选出所有基于MD5作为完整性验证的密码套件，如表1-2所示。

表1-2 摘要算法关键字

关 键 字	含 义
MD5	使用MD5的密码套件。已废弃而且不安全
SHA、SHA1	使用SHA1的密码套件
SHA256(v1.0.0+)	使用SHA256的密码套件
SHA384(v1.0.0+)	使用SHA384的密码套件

> **注意**
> 摘要算法的关键字只筛选出在协议层会进行数据完整性验证的套件。TLS 1.2引入了对可验证加密的支持，它是一种将加密和完整性检查打包在一起的机制。当使用了AEAD（authenticated encryption with associated data）之后，TLS协议就不需要提供额外的完整性检查。因此你无法通过摘要算法筛选出AEAD套件（这些套件现在的名称里面会带上GCM）。这些套件名称虽然以SHA256和SHA384结尾，但是这里它们仅仅用来生成套件所需要的**伪随机函数**（pseudorandom function）。

验证关键字会筛选出那些基于它们所用验证方式的套件（参见表1-3）。现在几乎所有的公用证书都使用RSA作为验证。未来我们也许会看到越来越多的人使用椭圆曲线（elliptic curve，ECDSA）证书。

表1-3　验证关键字

关　键　字	含　义
aDH	使用DH验证的密码套件，即证书携带了DH密钥（v1.0.2+）
aDSS、DSS	使用DSS验证的密码套件，即证书携带了DSS密钥
aECDH（v1.0.0+）	使用ECDH验证的密码套件
aECDSA（v1.0.0+）	使用ECDSA验证的密码套件
aNULL	不支持验证的密码套件。现在是指匿名DH算法。不安全
aRSA	使用RSA验证的密码套件，即证书携带了RSA密钥
PSK	使用PSK（Pre-Shared密钥）进行验证的密码套件
SRP	使用SRP（安全远程密码）进行验证的密码套件

密钥交换关键字会基于所选择的密钥交换算法筛选套件（参见表1-4）。当使用临时Diffie-Hellman套件的时候，OpenSSL对套件和关键字的命名总是不一致。在套件名称里面，临时套件会将E放在密钥交换算法的后面（例如，ECDHE-RSA-RC4-SHA和DHE-RSA-AES256-SHA），但是关键字则是放在开头（例如，EEDHE和EDH）。更糟糕的是，有些老的套件会将E放在密钥交换算法的开头（例如，EDH-RSA-DES-CBC-SHA）。

表1-4　密钥交换关键字

关　键　字	含　义
ADH	匿名DH密码套件。不安全
AECDH(v1.0.0+)	匿名ECDH密码套件。不安全
DH	使用DH的密码套件（包括临时和匿名DH）
ECDH(v1.0.0+)	使用ECDH的密码套件（包括临时和匿名ECDH）
EDH(v1.0.0+)	使用临时DH进行密码协商的密码套件
EECDH（v1.0.0+）	使用临时ECDH的密码套件
kECDH（v1.0.0+）	使用ECDH密码协商的密码套件
kEDH	使用临时DH密钥协商的密码套件（包括匿名DH）

（续）

关 键 字	含 义
kEECDH（v1.0.0+）	使用临时ECDH密钥协商的密码套件（包括匿名ECDH）
kRSA、RSA	使用RSA进行密钥交换的密码套件

密码关键字基于它们使用的密码选择套件（参见表1-5）。

表1-5 密码关键字

关 键 字	含 义
3DES	使用三倍DES的密码套件
AES	使用AES的密码套件
AESGCM（v1.0.0+）	使用AES GCM的密码套件
CAMELLIA	使用Camellia的密码套件
DES	使用单独DES的密码套件。已废弃而且不安全
eNULL、NULL	不加密的密码套件。不安全
IDEA	使用IDEA的密码套件
RC2	使用RC2的密码套件。已废弃而且不安全
RC4	使用RC4的密码套件。不安全
SEED	使用SEED的密码套件

还有一些套件无法归到任何类别里面（参见表1-6）。它们中的大部分与GOST标准相关，是在前苏联解体之后，独联体中的一些国家开发出来的。

表1-6 其他关键字

关 键 字	含 义
@STRENGTH	根据加密算法密钥长度进行排序后的密码套件列表
aGOST	使用GOST R 34.10（要么2001，要么94）进行验证的密码套件，需要有GOST功能的引擎
aGOST01	使用GOST R34.10-2001验证的密码套件
aGOST94	使用GOST R 34.10-94验证的密码套件。已废弃。请使用GOST R 34.10-2001
kGOST	使用VKO 34.10进行密钥交换的密码套件，已在RFC 4357中指定
GOST94	基于GOST R 34.11-94使用HMAC的密码套件
GOST89MAC	使用GOST 28147-89而且不是HMAC的密码套件

3. 组合关键字

大部分情况下使用直接关键字本身即可，但是有时候也需要使用+号将两个或者多个关键字组合在一起来筛选出符合多个要求的套件。下面的例子中我们筛选出RC4和SHA的套件：

```
$ 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
```

4. 创建密码套件列表

构建一份密码套件配置的关键在于*当前套件列表*（current suite list）。这份列表最开始都是空的，没有任何套件，不过当你给配置字符串添加任何关键字之后这份列表就会发生变化。默认情况下，新套件会追加到结尾处。例如要选择RC4和AES算法的套件可以是：

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

冒号常用来分隔关键字，空格和逗号也有同样的作用。下面的命令与之前命令的效果一样：

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

5. 关键字修饰符

关键字修饰符是那些可以放在每个关键字的前面的字符，它可以改变默认行为（默认行为是加入到列表里面），支持下面这些操作。

- *结尾附加*

  在列表末尾加入套件。列表上已经有的套件会保持原来的位置。当关键字之前没有别的修饰符时，这是默认行为。

- *删除（-）*

  从列表中删除所有匹配的套件，但是后续其他关键字依旧可以让这些套件重现。

- *永久删除（!）*

  从列表中删除所有匹配的套件并且后续任何关键字也无法重现这些套件。当你确定永远不再使用某些套件的情况下，使用这个修饰符可以让你后续的决策变得简单，同时可以减少犯错。

- *移到行尾（+）*

  将所有符合的套件移到列表末尾。仅对已经存在的套件有效，不会新增套件到列表里面。在你希望保留某些弱套件同时希望优先使用强密码套件的时候，就可以使用这种方式。例如RC4:+MD5会保留所有的RC4套件，但是将其中基于MD5的套件移到末尾。

- *排序*

@STRENGTH和别的关键字都不一样（我猜测这也是为什么它的名称里面有一个@）：它不会增加或者减少任何套件，但是会按照套件的加密强度进行降序排列。自动排序是一个很好的想法，但是在现实中我们很难通过加密长度就判断出密码套件的强度。

例如下面的配置：

```
$ 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
```

理论上，输出是按照强度排序的，现实中你会希望能够更好地控制套件顺序：

- 例如当使用TLS 1.0以及更低版本的时候，AES256-SHA（CBC套件）会遭受到BEAST攻击，这时候你会希望提高RC4-SHA套件的优先级，它不受BEAST攻击的影响。
- 3DES表面上是168位强度，实际上一种叫作中途相遇（meet-in-the-middle）的攻击可以将它的强度降到112位，[①]加上其他的问题甚至可以将强度降为108位。[②]实际上DES-CBC3-SHA应该被归为128位密码套件，严格来说OpenSSL将3DES归为168位密码套件其实是一个bug，未来有可能会修复。

6. 处理错误

你在配置的时候可能会遇到两种错误，第一种是排版错误或者使用了不存在的关键字：

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

这个输出结果非常含糊，但是的确包含了错误信息。

另外一种错误是出现类似下面的结果，一般是因为不存在匹配的密码套件。

```
$ 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:
```

7. 合并多个套件

为了说明如何将多个套件的功能组合在一起，我会举一个现实的例子；但是请记住下面仅仅是一个例子。因为在进行配置之后我们有很多东西需要考虑，所以不存在一个配置可以匹配所有情况。

因此在开始配置之前，我们需要清晰地确定想要达成什么效果。对我来说，我希望有下面这样一个相对安全同时高性能的配置。

(1) 只使用加密效果128位及以上的密码套件（这就排除掉了3DES）。

(2) 只使用提供强验证的套件（这就排除掉了匿名和出口套件）。

(3) 不要使用任何依赖不安全算法的套件。

(4) 实现无论是用什么密钥和协议，都可以支持前向保密（forward secrecy）。这个要求会让我们无法使用RSA进行高性能的密钥交换，因此会导致一部分的性能损失。通过将更快的ECDHE的优先级提到DHE之前，可以尽可能地降低损耗。

(5) 优先使用ECDSA而不是RSA。这项要求需要我们支持双密钥部署模式，因为这样我们才能尽可能地使用ECDSA算法，同时又可以支持那些只能使用RSA的客户端。

(6) 支持TLS 1.2的客户端会优先使用AES GCM套件，这个套件是在当前TLS版本中安全性最

---

① Cryptography/Meet In The Middle Attack，https://en.wikibooks.org/wiki/Cryptography/Meet_In_The_Middle_Attack（Wikibooks，检索于2014年3月31日）。

② Attacking Triple Encryption，http://th.informatik.uni-mannheim.de/people/lucks/publ.shtml（Stefan Lucks，1998）。

好的。

(7) 最近RC4有更多弱点被发现[①]，所以我将它放到了列表的末尾，当然最好是去掉它。虽然BEAST在某些情况下依旧是一个问题，我假设这个问题已经由客户端解决掉了。

从一开始就应该禁用掉所有不希望使用的套件，这样就可以减少很多配置，而且避免了不小心错误地引入不该使用的套件。

使用下面这些加密字符串可以筛选出弱密码套件。

- ❑ aNULL　无验证
- ❑ eNULL　无加密
- ❑ LOW　低强度套件
- ❑ 3DES　有效加密强度为108位
- ❑ MD5　使用MD5的套件
- ❑ EXP　已废弃的出口套件

因为DSA、PSK、SRP和ECDH很少使用，所以我直接将它们去除，这样也可以减少出现的套件的数量。OpenSSL虽然还支持IDEA和SEED，但它们都已经是废弃的算法了。在我的配置里也去掉了CAMELLIA，因为它比较慢而且不如AES支持的客户端多（例如在实际中不支持GCM或者ECDHE变体）。

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

现在我们将注意力聚焦在我们想要的效果上。因为前向保密是我们的最高优先级，所以我们以kEECHE和kEDH关键字开头：

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

你会发现如果直接使用上面的配置，RSA套件会排在前面，但其实我希望将ECDSA放在最前面：

```
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
[...]
```

因此我可以将kEECDH+ECDSA放在配置的最前面，这样就可以将ECDSA优先筛选出来了：

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

下面一个问题是老的套件（SSL 3）和新的套件混在一起了（TLS 1.2）。为了尽可能安全，我希望所有的TLS 1.2的客户端都能协商并使用TLS 1.2的密码套件。因此可以通过使用+SHA关键字来将老的套件放到列表的末尾（TLS 1.2套件只会使用SHA256和SHA384，所以它们不会匹

① On the Security of RC4 in TLS and WPA，http://www.isg.rhul.ac.uk/tls/（AlFardan等，2013年3月13日）。

配到）。

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

到这里基本完成，只需要在使用HIGH关键字将剩余的安全套件也加入到列表末尾即可。另外，还需要确保RC4套件在列表的末尾，可以使用+RC4（将RC4套件放到列表末尾）和RC4（将剩余的未在列表里面的RC4套件加入进来）：

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

我们再看一下最后的输出结果，一共由28个套件组成。第一组都是TLS 1.2的套件：

```
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
```

优先使用ECDHE，然后是DHE，最后才是剩余的TLS 1.2套件。在任何一组里面，ECDSA和GCM都拥有更高的优先级。

第二组是TLS 1.0客户端会使用的套件，使用类似第一组的优先级顺序：

```
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
```

最后把RC4套件放到末尾：

```
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
```

8. 推荐配置

前面一节展示了一个使用OpenSSL套件关键字生成密码套件配置的例子，但这不是最好的方

式。事实上没有任何一个配置可以满足所有人。在本节中我会基于你的偏好和风险评定，提供几个不同的配置供你选择。

本节设计的配置原理与前面的方式一样，但是我会修改两个地方来获得更好的性能。首先我会将128位的套件放在列表的前面，因为虽然256位的套件安全性更高，但是128位在当前已经足够安全，所以使用256位的意义不大，而且还会造成性能损失。另外我倾向于选择HMAC-SHA而不是HMAC-SHA256或者HMAC-SHA384，后面两个更慢而且所提高的安全性当前看来意义不大，HMAC-SHA就已经足够了。

另外我将由原来用关键字进行配置的方式改成直接使用套件名称。使用关键字不是个坏主意，你只需要确定安全的需求，剩下的由OpenSSL库去帮你选择，你甚至不需要知道会使用哪些套件。在实际中这种方式并不理想，因为我们对所需要的套件要求越来越严格，特别是我们有时候就想使用其中某个套件，以及按照我们所希望的顺序排列。

在配置里面直接使用套件名称还有一个好处：你只需要列出你想要的套件即可；而且如果你看了别人的配置，就可以直接通过套件名称确定它们使用的套件，不再需要通过OpenSSL执行关键字命令后才能获取。

下面是我默认开始的配置，同时满足了强加密和高性能。

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

上面这份配置只提供了支持前向保密和强加密的套件。当前流行的浏览器和客户端都能支持，但是一些老的客户端，例如运行在Windows XP上的那些老版本IE就会失败。如果真的需要支持那些非常老的客户端，那只能考虑在末尾加上下面的套件：

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

这些老旧套件大部分使用RSA作为密钥交换，因为无法提供前向保密的功能。我们将AES放在前面，但是为了最大限度地支持更多的客户端，我们也将3DES和不安全的RC4加了进去。如果无法避免使用RC4，最好用ECDHE套件来提供前向保密。

1.3.2 性能

你可能知道，计算速度是任何加密操作最大的影响因子。OpenSSL自带了性能评测工具，你可以使用它对系统的能力和上限进行一个大致的评定。你可以使用speed命令来执行评测。

如果调用speed命令的时候没有带上任何参数，OpenSSL会输出很多无用信息，最好的方式是只测试你关心的那些套件。从Web服务器安全考虑，你可能会关心RC4、AES、RSA、ECDH和SHA算法。

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

输出一共包括三个部分，第一部分包括OpenSSL的版本信息和编译时间。如果你在测试不同版本的OpenSSL，并且编译时的选项也不一样，那么这些信息将会很有帮助：

```
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.
```

第二部分包含对称密码性能数据（例如，散列函数和私有密码算法）：

```
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
```

最后，第三部分包含非对称加密算法的性能数据：

```
                  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
```

这些输出可以干什么？你可以对比相同平台下不同编译选项或者不同OpenSSL版本对性能的影响。例如前面的结果就是真实环境下的OpenSSL 0.9.8k运行之后的结果(发行方打过了补丁)。我迁移到OpenSSL 1.0.1h的出发点是因为希望支持TLS 1.1和TLS 1.2；那是否会有性能的影响呢？我下载编译了OpenSSL 1.0.1h进行测试。让我们看一下：

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

很显然，OpenSSL 1.0.1h的性能在这台服务器上几乎快一倍 (使用2048位的RSA密钥)；性能从每秒277次签名操作提高到每秒450次。这个结果意味着我升级之后还能获得更好的性能，这真的是一个好消息！

直接使用这个测试结果来衡量部署之后的性能并不合适，因为在实际场景中有很多影响性能的因素，而且很多影响因素与TLS没有直接关系 (例如HTTP的长连接设置、缓存设置等)。因此这些数字最多可用来作一个粗略的估计。

在粗略估计之前，你还需要考虑一些东西。默认情况下speed命令只会使用一个进程。而现在大多数服务器都有多核，所以你必须让speed命令并发使用多个实例才能算出整台服务器能支持的TLS操作次数。使用-multi参数就可以实现这个测试，我的服务器有4个核，所以我使用下面这个命令：

```
$ 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
```

正如预期的那样，性能几乎是原来的4倍。我又看了一下每秒钟可以执行多少次RSA签名操作，因为这是一台服务器上最消耗CPU的密码操作，所以也是最先产生瓶颈的地方。1090次/秒的签名结果告诉我们，这台服务器每秒可以处理大约1000次全新的TLS连接。这对我来说已经足够了，而且有非常健康、安全的余量。因为我在服务器上启用了会话恢复，所以我知道能够每秒处理超过1000次的TLS连接。我希望这台服务器上的流量能够多到让我可以担心TLS的性能为止。

另外一个不能完全相信speed命令输出结果的原因在于，它不会默认使用可用密码中最快的实现方式。在某种情况下，默认的输出具有欺骗性。例如，支持AES-NI指令的服务器可以加速AES计算，在测试的时候这个特性不会默认被用到。

```
$ openssl speed aes-128-cbc
[...]
The 'numbers' are in 1000s of bytes per second processed.
type             16 bytes     64 bytes    256 bytes   1024 bytes   8192 bytes
aes-128 cbc      67546.70k    74183.00k    69278.82k   155942.87k   156486.38k
```

为了激活硬件加速卡，需要在命令行上使用-evp开关：

```
$ openssl speed -evp aes-128-cbc
[...]
The 'numbers' are in 1000s of bytes per second processed.
type             16 bytes     64 bytes    256 bytes   1024 bytes   8192 bytes
aes-128-cbc     188523.36k   223595.37k   229763.58k   203658.58k   206452.14k
```

1.4 创建私有证书颁发机构

如果想要建立自己的CA，OpenSSL已经包含了所有你需要的东西。所有的操作都通过纯命令行执行，虽然不那么友好，整个过程也比较长，但是这可以让你去思考每个细节。

我建议自己创建一套私有的CA主要是出于教学的目的，不过还有一些别的原因。OpenSSL的CA天然满足个人或者小团体的需求，例如在开发环境使用一套CA比到处使用自签名的证书好得多。同时还可以通过客户端证书来提供双向验证，这可以极大地提高敏感Web应用的安全性。

运行私有CA最大的挑战不是设置问题，而是如何保证基础结构的安全。例如根密钥必须离线保存，因为所有的安全都依赖它。另一方面，CRL和OCSP响应程序证书必须定期进行更新，而这会要求根密钥保持联机。

1.4.1　功能和限制

在本节的剩余部分，我们会创建一个与公共CA类似的私有CA架构。会先有一个根CA，然后创建其他的二级CA。接着我们会通过CRL和OCSP服务提供证书吊销信息。为了让根CA的私钥可以离线保存，OCSP响应程序需要使用它们自己的身份。这并非是最简单的CA，但是相对来说比较安全。另外二级CA会在技术上进行限制，只能给允许的主机名签发证书。

完成设置之后，必须将根证书安全地分发给所有客户端。一旦根证书分发完毕，就可以开始签发客户端和服务器证书了。有一个限制是以这种方式设置的OCSP响应程序主要用来测试，因为只能承受比较小的负载。

1.4.2　创建根 CA

创建全新的CA有几个步骤：配置、创建目录结构和初始化密钥文件，最后生成根密钥和证书。本节描述了整个过程和常见的CA操作。

1. 根CA配置

创建CA之前，我们需要先准备一个配置文件告诉OpenSSL我们希望的配置。一般情况下并不需要配置文件，但是根CA的创建操作复杂，使用配置文件可以简便很多。OpenSSL的配置文件很强大，在开始之前我建议你熟悉一下这些配置的功能（命令行上使用man config命令）。

配置文件第一部分包括了CA的名称、基础URL和CA可分辨名称等基本信息。因为这些配置都很灵活，只需配置一次即可。

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

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

第二部分直接控制了CA的操作。有关每个设置的完整信息，可以通过ca命令来获取它的文档（命令行上输入man ca）。大部分命令从字面意思就可以很容易理解，我们需要告诉OpenSSL存放文件的路径。因为根CA只用作二级CA的签发，所以我把有效期设置为10年。另外默认情况下使用SHA256作为签名算法。

默认策略（policy_c_o_match）限制了这张CA签发的证书的国家名和组织名会与CA本身一样。对于公共CA来说很少会有这样的设置，但对于私有CA来说这种方式比较合适：

```
[ca_default]
```

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

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

第三部包含了req命令的配置，req命令只会在创建自签发根证书的时候用到一次。最重要的部分是扩展：基本限制（basicContraints）扩展表明这个证书是一张CA，密钥用法（keyUsage）扩展用来说明这个CA的用处：

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

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

配置的第四部分包括了根CA创建证书所需要的信息。因为基本限制（basicContraints）扩展的设置，所有的证书都将成为CA，但是我们需要把pathlen设置为0，表示这些CA无法再签发新的CA了。

所有二级CA都会受到限制，也就是说他们签发的证书只能对一些域名的子集有效，并且会被限制使用场景。第一，扩展密钥用法（extendedKeyUsage）扩展限制了只能进行客户端验证（clientAuth）和服务器验证（serverAuth），也就是TLS的客户端和服务器验证。第二，名称限制（nameContraints）扩展限制了允许签发的域名只有example.com和example.org。理论上这样的设置让你可以签发二级CA给第三方，同时可以通过限制他们无法签发任意域名的主机名来保证

安全。排除这两个IP段的要求来自CAB论坛的*Baseline Requirements*，该规范从技术上定义了对二级CA的限制。[①]

实际上，名称限制（nameContraints）并不完美，因为当前还有很多主流的平台无法识别名称限制扩展。如果你将这个扩展标记为关键扩展，就会导致很多平台拒绝识别你的证书。如果不将其标记为关键扩展，那么很多平台就不会识别这个扩展，导致名称限制实际没有任何效果。

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

[crl_info]
URI.0                    = $crl_url

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

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

最后两部分的配置表示有了这个扩展的证书可以对OCSP响应进行签名。为了能够运行OCSP响应程序，我们生成一个特别的证书，并且将OCSP的签名能力赋予这张证书。从扩展可以看出这张证书不是一个CA：

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

2. 根CA的目录结构

下面我们会创建“根CA配置”中说到的目录结构，并且会初始化一些CA操作中会用到的文件。

```
$ mkdir root-ca
$ cd root-ca
$ mkdir certs db private
$ chmod 700 private
```

① *Baseline Requirements*，https://cabforum.org/baseline-requirements/（CAB论坛，检索于2014年7月9日）。

```
$ touch db/index
$ openssl rand -hex 16 > db/serial
$ echo 1001 > db/crlnumber
```

我们会用到以下这几个目录。

- certs/
  存放证书的地方；证书在签名之后会放置到这个目录下。
- db/
  这个目录用于证书数据库（index），一些包括下一张证书以及CRL数字的文件。OpenSSL会创建额外需要的一些文件。
- private/
  这个目录会存放私钥，一个给CA使用，一个给OCSP响应程序使用。务必确保其他用户都不能访问这个目录（事实上，如果你真的很在意这个CA，那么这台存放根证书和密钥的服务器的用户账户必须尽可能少）。

> **注意**
> 创建一个新的CA证书的时候，就像本节我所做的这样，很重要的一点是使用随机数生成器来初始化证书序列号。当你使用同样的名称创建和发布多个CA证书的时候会非常有用（另外，常见的情况是你在生成证书的过程中出错了而不得不重新来过）；通过这样的方式可以让每个证书拥有不一样的序列号，这样就不会互相冲突了。

3. 根CA生成

我们需要分两步来创建根CA。首先，我们生成密钥和CSR文件。当我们使用-config开关之后，所有需要的信息都会从配置文件中加载进来：

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

第二步我们会创建自签名证书。-extension开关指向了配置文件的ca_ext部分，这样可以激活根CA所需的扩展。

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

4. 数据库文件结构

db/index中的数据库是一个包含证书信息的明文文件，每行一个证书。我们刚刚创建根CA，现在这个文件只有一行信息：

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

每一行包括以下6个以制表符分隔的值。

(1) 状态标记［V表示有效（valid），R表示已吊销（revoked），E表示已过期（expired）］
(2) 过期时间（以YYMMDDHHMMSSZ格式表示）
(3) 吊销日期，如果没有被吊销则为空
(4) 序列号（十六进制）
(5) 文件路径（如果不知道就显示unknown）
(6) 可分辨名称

5. 根CA操作

使用ca命令的-gencrl开关给新CA生成CRL：

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

使用ca的的命令来签发证书。需要注意的是-extensions开关需要指向配置文件里面正确的部分（例如，你肯定不希望再生成另一个根CA）。

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

如果要吊销证书，可以使用ca命令的-revoke开关，不过需要有一份你想吊销的证书的副本。不过因为所有的证书都存在certs/目录下，所以只需要知道序列号即可。如果知道证书的可分辨名称，就可以在数据库里面查到它的序列号了。

为-crl_reason开关中的值选择一个正确的理由。该值可以是以下这些值之一：unspecified、keyCompromise、CACompromise、affiliationChanged、superseded、cessationOfOperation、certificateHold和removeFromCRL。

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

6. 创建用于OCSP签名的证书

首先我们需要给OCSP响应程序创建一个私钥和CSR。这两个操作对所有的非CA证书都适用，所以不需要指定配置文件：

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

第二步需要使用根CA签发一张证书。-extensions开关的值选择ocsp_ext，以确保设置了OCSP签名所需要的扩展。我将这个证书的生命周期减少为365天（原来默认是3650天）。这些OCSP

证书是没有吊销信息的，所以无法吊销它们。因此你会希望尽可能减少它们的生命周期。30天是一个比较好的选择，当然前提是你已经准备好频繁地创建新的OCSP证书。

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

现在你已经有了OCSP响应程序所需要的一切东西，可以直接在根CA所在的服务器上进行测试。当然，如果在生产环境中使用，就必须将OCSP响应程序的密钥和证书放到别的地方：

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

可以使用下面的命令来测试OCSP响应程序：

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

输出结果中的verify OK表示已经成功验证签名，而good表示这张证书还没有被吊销。

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

1.4.3 创建二级 CA

创建二级CA的过程和根CA几乎完全一样。本节我会突出那些不同之处，其他部分可以参考1.4.2节。

1. 二级CA配置

我们可以在前面根CA配置文件的基础上，进行一些适当的修改生成二级CA的配置。我们会把名称改为sub-ca并且使用另一个可分辨名称。我们将二级CA的OCSP响应程序放在另外一个端口，主要是因为ocsp命令不识别虚拟主机。如果为OCSP响应程序使用了适合的Web服务器，就可以完全避免使用特别的端口。该证书默认的生命周期是365天，我们会每隔30天生成全新的CRL。

将copy_extensions更改为copy意味着在生成新证书的时候，如果我们的配置文件里面没有设置某些扩展，那么就会使用CSR文件里面的扩展字段。进行此更改之后，在准备CSR文件的时候就可以加入别的一些需要的字段，这些信息会在生成证书的时候加入到证书里面。这个特性有几

分危险（因为允许其他人可以在一定程度上直接控制证书里面的内容），不过我认为在较小的环境中这么做是可以的。

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

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

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

在配置文件的最后面，我们会增加两个新的配置，分别用于服务器和客户端证书的生成。唯一的区别就是keyUsage和extendedKeyUsage扩展。注意到我们把basicContraints扩展的值设置为false。之所以这么做，原因在于我们会复制CSR文件里面的扩展。如果在配置文件中没有显示设置这个扩展，那么就可能会用到CSR文件中的basicContraints了。

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

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

改好配置文件之后，按照根CA的过程创建一个同样的目录结构，不过可以使用另外一个名称，比如sub-ca。

2. 二级CA生成

与前面一样，创建二级CA需要两步。第一步生成密钥和CSR。当我们使用-config开关的时候，所有需要的信息都会从配置文件中加载进来。

```
$ openssl req -new \
    -config sub-ca.conf \
    -out sub-ca.csr \
```

```
    -keyout private/sub-ca.key
```

第二步我们使用根CA来签发证书。-extensions开关指向配置文件中的sub_ca_ext，从而使用二级CA所需要的扩展。

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

3. 二级CA操作

要签发服务器证书，可以在处理CSR文件的时候，在-extensions开关中指定server_ext：

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

要签发客户端证书，可以在处理CSR文件的时候，在-extensions开关中指定client_ext：

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

> **注意**
>
> 当收到新证书申请请求的时候，你需要在对所有信息进行验证之后才能进行操作。你需要确保所有资料符合规定，特别是当你处理的CSR文件是别人生成的时。特别要注意证书的可分辨名称以及basicContraints和subjectAlternativeName扩展。

二级CA的CRL生成和证书的吊销过程与根CA是一样的。唯一不同的是OCSP响应程序所使用的端口；二级CA使用的是9081端口。推荐OCSP响应程序使用独立的证书，这样可以避免将二级CA部署到公开的服务器上。

第 2 章

使用OpenSSL进行测试

2

因为有大量的协议特性和实现上的一些怪癖，所以有时候很难确定安全服务器精确的配置和特性是什么。虽然现在有很多工具都适用于这个问题，但是很难知道它们是如何实现的，有时候很难完全信任它们的结果。尽管我花了很多年的时间测试安全服务器，也看到了很多好的工具，但是当我想要理解细节的时候，又会回归到使用OpenSSL和Wireshark。我并不是说每次测试都应该使用OpenSSL；相反，应该找到一种可信赖的自动化工具。在真的想要确定某些事情的时候，唯一的方式就是亲自使用OpenSSL去测试。

2.1 连接 SSL 服务

OpenSSL自带的客户端工具可以用来连接安全服务器。这个工具很像telnet或者nc，严格来说是处理SSL/TLS协议层，但是允许你完全控制其之上的协议层。

你需要提供主机名和端口来连接服务器，例如：

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

一旦输入这个命令，就会看到很多诊断输出（稍后详细解释），然后你就有机会输入任何你需要的命令。因为我们访问的是一台HTTP服务器，所以最明智的做法是提交一个HTTP请求。下面的例子中，我提交一个HEAD请求，因为它告诉服务器不需要发送响应体。

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

HTTP/1.1 200 OK
Date: Tue, 10 Mar 2015 17:13:23 GMT
Server: Apache
Strict-Transport-Security: max-age=31536000
Cache-control: no-cache, must-revalidate
Content-Type: text/html;charset=UTF-8
Transfer-Encoding: chunked
Set-Cookie: JSESSIONID=7F3D840B9C2FDB1FF7E5731590BD9C99; Path=/; Secure; HttpOnly
Connection: close

read:errno=0
```

现在我们知道TLS通信层在工作了：我们连上了HTTP服务器，提交了请求，然后收到了响

应。让我们回到诊断输出，最开始的几行显示的是服务器的证书信息。

```
CONNECTED(00000003)
depth=3 L = ValiCert Validation Network, O = "ValiCert, Inc.", OU = ValiCert Class ↩
2 Policy Validation Authority, CN = http://www.valicert.com/, emailAddress = ↩
info@valicert.com
verify error:num=19:self signed certificate in certificate chain
verify return:0
```

在我的系统上（你的系统也许也一样），s_client不会寻找系统默认的可信证书，它会提示在证书链中存在自签名证书。在大多数情况下不用关注证书是否有效，但是如果有必要的话，可以让s_client指定可信根证书，例如：

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

与之前提示的不一样，现在你可以看到它验证了证书链上每一级的证书。为了验证通过，你必须选择好CA证书路径。我在例子中使用的路径（/etc/ssl/certs/ca-certificates.crt）在Ubuntu 12.04 LTS上是有效的，但并不代表在你的系统上也是如此。如果不想使用系统提供的CA证书来进行验证，可以依赖由Mozilla提供的证书，我们曾经在1.1.4节中讨论过这些证书。

> **警告**
>
> Apple的操作系统OS X随机附带的OpenSSL经过修改，有时候会覆盖证书校验过程。换句话说，-CAfile开关可能无法如我们期望的那样工作。可以通过在调用s_client命令之前设置OPENSSL_X509_TEA_DISABLE环境变量来解决这个问题。[①]考虑到OS X上默认使用的OpenSSL还是基于非常陈旧的0.9.x分支，所以最好还是升级到最新的版本，可以使用Homebrew或者MacPorts进行安装。

输出中的下一部分按照服务器交付的顺序罗列了所有证书。

```
Certificate chain
```

① Apple OpenSSL Verification Surprises，https://hynek.me/articles/apple-openssl-verification-surprises/（Hynek Schlawack，2014年3月3日）。

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

每一张证书都会在第一行显示使用者信息，第二行显示颁发者信息。

这部分很重要，特别是当你需要看看到底发送了什么证书的时候；浏览器证书查看器一般都会显示重建的证书链，可能与上面呈现的会完全不一样。为了确定证书链是否正确，你可能会希望验证使用者和颁发者信息是否匹配。从最上面的分支（Web服务器）证书开始，逐级遍历列表，看看当前证书的颁发者与从属证书的使用者是否匹配。最后你看到的颁发者指向了某个不在证书链里面的根证书；或者是一个自己指向自己的根证书。

下一个输出的是服务器证书；它非常长，为了更好地呈现，我进行了简化：

```
Server certificate
-----BEGIN CERTIFICATE-----
MIIF5zCCBM+gAwIBAgIHBG9JXlv9vTANBgkqhkiG9w0BAQUFADCB3DELMAkGA1UE
[30 lines removed...]
os5LW3PhHz8y9YFep2SV4c7+NrlZISHOZVzN
-----END CERTIFICATE-----
subject=/1.3.6.1.4.1.311.60.2.1.3=GB/businessCategory=Private Organization ↩
/serialNumber=06694169/C=GB/ST=London/L=London/O=Feisty Duck Ltd
/CN=www.feistyduck.com
issuer=/C=US/ST=Arizona/L=Scottsdale/O=Starfield Technologies, Inc./OU=http:/ ↩
/certificates.starfieldtech.com/repository/CN=Starfield Secure Certification
Authority/serialNumber=10688435
```

> **注意**
>
> 任何时候当你在主题栏看到很长一串数字而不是一个名称的时候，意味着OpenSSL并不认识这个**对象标识符**。对象标识符是全球唯一而且是用来清楚指向某些东西的标识符。例如在前面的输出中，OID 1.3.6.1.4.1.311.60.2.1.3应该显示成jurisdictionOfIncorporationCountryName，这个OID用于**扩展验证**证书。

如果想深入了解证书，就需要先从输出内容中复制一份并保存在单独的文件中，我会在2.2节中具体讨论。

下面的信息是关于TLS连接的，意思从字面上就可以很容易理解了：

```
---
No client certificate CA names sent
---
SSL handshake has read 3043 bytes and written 375 bytes
---
New, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES256-SHA
Server public key is 2048 bit
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
SSL-Session:
    Protocol  : TLSv1.1
    Cipher    : ECDHE-RSA-AES256-SHA
    Session-ID: 032554E059DB27BF8CD87EBC53E9FF29376265F0BBFDBBFB7773D2277E5559F5
    Session-ID-ctx:
    Master-Key: 1A55823368DB6EFC397DEE2DC3382B5BB416A061C19CEE162362158E90F1FB0846E ↩
EFDB2CCF564A18764F1A98F79A768
    Key-Arg   : None
    PSK identity: None
    PSK identity hint: None
    SRP username: None
    TLS session ticket lifetime hint: 300 (seconds)
    TLS session ticket:
    0000 - 77 c3 47 09 c4 45 e4 65-90 25 8b fd 77 4c 12 da   w.G..E.e.%..wL..
    0010 - 38 f0 43 09 08 a1 ec f0-8d 86 f8 b1 f0 7e 4b a9   8.C..........~K.
    0020 - fe 9f 14 8e 66 d7 5a dc-0f d0 0c 25 fc 99 b8 aa   ....f.Z....%....
    0030 - 8f 93 56 5a ac cd f8 66-ac 94 00 8b d1 02 63 91   ..VZ...f......c.
    0040 - 05 47 af 98 11 81 65 d9-48 5b 44 bb 41 d8 24 e8   .G....e.H[D.A.$.
    0050 - 2e 08 2d bb 25 59 f0 8f-bf aa 5c b6 fa 9c 12 a6   ..-.%Y....\.....
    0060 - a1 66 3f 84 2c f6 0f 06-51 c0 64 24 7a 9a 48 96   .f?.,...Q.d$z.H.
    0070 - a7 f6 a9 6e 94 f2 71 10-ff 00 4d 7a 97 e3 f5 8b   ...n..q...Mz....
    0080 - 2d 1a 19 9c 1a 8d e0 9c-e5 55 cd be d7 24 2e 24   -........U...$.$
    0090 - fc 59 54 b0 f8 f1 0a 5f-03 08 52 0d 90 99 c4 78   .YT...._..R....x
    00a0 - d2 93 61 d8 eb 76 15 27-03 5e a4 db 0c 05 bb 51   ..a..v.'.^.....Q
    00b0 - 6c 65 76 9b 4e 6b 6c 19-69 33 2a bd 02 1f 71 14   lev.Nkl.i3*...q.

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

最重要的信息是协议版本（TLS 1.1）和使用的密码套件（ECDHE-RSA-AES256-SHA）。另外还能发现服务器返回了一个会话ID和TLS 会话票证（一种在非服务器维护状态下即可恢复会话的方式），而且也支持安全重新协商。一旦理解了这些输出内容所包含的信息，以后就无需再关注它了。

> **警告**
> 操作系统发行版常常自带工具并且与普通版本都有所差别。这里还有另外一个例子：前面一个命令协商了TLS 1.1，即便服务器支持TLS 1.2，为什么？原来是这样的，Ubuntu 12.04 TLS附带的某些OpenSSL版本禁用了TLS 1.2作为客户端连接，为的是避免某些互操作性问题。为了避免这类问题，我建议使用之前配置和编译过的OpenSSL版本。

2.2　测试升级到 SSL 的协议

当使用HTTP的时候，TLS将整个明文通讯通道包起来成为HTTPS。其他一些协议开始是明文，然后升级到加密模式。如果想要测试这类协议，那么就需要告诉OpenSSL是什么协议，这样OpenSSL就能够代表你进行升级。在-startssl开关后面带上协议信息。例如：

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

在撰写本书的时候，OpenSSL支持smtp、pop3、imap、ftp和xmpp。

2.3　使用不同的握手格式

有时用OpenSSL测试服务器的时候，即便你知道服务器支持TLS（例如，使用浏览器访问的时候TLS可以正常工作），你对服务器的通信也可能会失败。其中一个可能的因素是服务器不支持老版本的SSL 2握手。

因为OpenSSL会尝试使用它理解的所有协议去进行协商，而且SSL 2默认情况下只能用老的SSL 2握手去进行协商。虽然这是个非常古老而且不安全的协议版本，但是从技术上来说老的握手过程并不是不安全的。它支持版本升级，也就是说可以协商更好的协议。然而SSL 2的握手格式不支持其后设计的许多连接协商特性。

因此，如果遇到了无法正常工作的情况，而且无法准确判断是什么的话，可以强制OpenSSL使用更新的握手格式。可以通过禁用SSL 2来实现：

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

另外一种实现同样效果的方式是在命令行上指明要访问的主机名：

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

为了指明主机名，OpenSSL需要使用一种更新的握手协议里面的特性（该特性叫作服务器名称指示，server name indication或SNI），该特性会强制废弃老的格式。

2.4　提取远程证书

当使用s_client连接远程安全服务器的时候，它会将服务器的PEM编码格式证书转储到标准输出。如果需要证书的话，那么可以从回滚（scroll-back）缓冲区里面复制出来。如果提前知道

你只需要获得证书的话，那么可以使用下面这个命令：

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


最前面的echo命令的目的是将shell命令和s_client分开。如果不用echo的话，那么s_client会一直等待你的输入，直到服务器超时（可能需要很长一段时间）。

默认情况下s_client只打印分支证书；如果希望打印完整的证书链，请使用-showcerts开关。使用了这个开关，前面这条命令会将所有的证书输出到同一个文件里面。

2.5 测试支持的协议

默认情况下s_client会尝试使用最高级的协议和远程服务器进行通信，并且在输出内容里面报告协商的版本信息。

```
Protocol  : TLSv1.1
```

有两种方式可以测试服务器是否某些版本的协议。第一种是在-ssl2、-ssl3、-tls1、-tls1_1或者是-tls1_2开关中明确地选择一个协议进行测试。另外一种方法是在-no_ssl2、-no_ssl3、-no_tls1、-no_tls1_1或者-no_tls1_2中选择一个或者多个你不想要进行测试的协议。

> **注意**
>
> 不是所有的OpenSSL都支持所有版本的协议。例如，老版本的OpenSSL不支持TLS 1.1和TLS 1.2，而新版本则可能不支持老的SSL 2之类的协议。

例如，当测试服务器不支持某个协议版本的时候可能会得到下面这些输出信息：

```
$ openssl s_client -connect www.example.com:443 -tls1_2 CONNECTED(00000003)
140455015261856:error:1408F10B:SSL routines:SSL3_GET_RECORD:wrong version number:s3_pkt.c:340:
---
no peer certificate available
---
No client certificate CA names sent
---
SSL handshake has read 5 bytes and written 7 bytes
---
New, (NONE), Cipher is (NONE)
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
SSL-Session:
    Protocol  : TLSv1.2
    Cipher    : 0000
    Session-ID:
    Session-ID-ctx:
    Master-Key:
    Key-Arg   : None
    PSK identity: None
    PSK identity hint: None
```

```
    SRP username: None
    Start Time: 1339231204
    Timeout   : 7200 (sec)
    Verify return code: 0 (ok)
---
```

2.6　测试支持的密码套件

如果你想要用OpenSSL去确定远程服务器是否支持某个特殊密码套件，这里有一个小技巧。加密配置字符串的设计是为了选择你想要使用的套件，所以如果只指定一个套件并且与服务器握手成功，那么就可以确定服务器支持这个套件。如果握手失败了，你就知道它并不支持。

我们以一个例子来测试一下服务器是否支持RC4-SHA算法，键入以下命令：

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

如果想确定特定服务器支持的所有算法套件，可以通过调用openssl cipehrs ALL来获取你的OpenSSL版本支持的所有套件列表，然后一个个提交给服务器来测试服务器是否支持该密码套件。我并非让你手动去做这些测试，通过一点点的自动化程序就可以很好地解决这个问题。事实上，遇到这种情况的时候去网上找个好工具可能是一个更好的办法。

这种测试方法有一个缺点就是你只能测试你的OpenSSL支持的密码套件。如果用的是1.0之前的版本，就会是一个大问题，因为那些版本支持的套件数量非常少（例如，我的服务器使用的0.9.8k版本只支持32个套件）。从1.0.1分支开始的版本都支持超过100种套件，几乎是能用到的所有套件了。

没有一种SSL/TLS库支持所有的密码套件，所以想要进行综合测试就有点困难了。在SSL Labs，进行部分握手测试时会遇到这种情况，我是通过使用定制的客户端去假装支持任意密码套件的方式来支持的。该客户端实际上甚至无法完成任何一个套件的协商，但是只需要发出协商就已经可以让我们知道服务器是否支持某个套件。使用这种方式不仅可以测试所有的套件，而且效率非常高。

2.7　测试要求包含 SNI 的服务器

最开始SSL和TLS协议设计的时候，每个IP端点（IP地址加端口）只支持一个网站。TLS的SNI扩展是为了在同一个IP端点上支持多张证书。TLS客户端使用这个扩展发送希望访问的主机名，而TLS服务器使用该扩展来选择的要用以响应的正确证书。简单来说，SNI支持了虚拟安全托管。

因为很多服务器还不支持SNI，所以大部分情况下你在使用s_client的时候不需要指定主机名。但是当你遇到启用了SNI支持的系统，就会遇到下面三种情况中的任意一种。

- ❑ 大多数情况下，无论是否提供SNI信息都会获得同样的证书。
- ❑ 服务器返回的证书可能是别的站点的证书，不是你想要测试的证书。
- ❑ 极少情况下服务器可能会中止握手，并且拒绝连接。

你可以在s_client中使用-servername开关来启用SNI：

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

你可以通过使用和不使用SNI开关，并且检查证书是否相同来确定某个站点是否支持SNI。如果证书不相同，表示SNI是必需的。

有时候，如果请求的主机名不可用，服务器会返回TLS警告。即便就服务器而言，这个警告不是致命的，客户端还是可能决定关闭连接。例如，使用老版本的OpenSSL（例如1.0.0之前的版本），你会得到下面的错误信息：

```
$ /opt/openssl-0.9.8k/bin/openssl s_client -connect www.feistyduck.com:443 -servername xyz.com
CONNECTED(00000003)
1255:error:14077458:SSL routines:SSL23_GET_SERVER_HELLO:reason(1112):s23_clnt.c:596:
```

2.8 测试会话复用

当加上-reconnect开关的时候，s_client命令可以用来测试会话复用。在此模式下，s_client与目标服务器连接6次，在第一次的时候会创建新的会话，然后在接下来的5次请求中复用同样的会话：

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

上面这个命令会产生大量输出，大部分都无需理会。最重要的部分是新会话和复用会话的信息。这些信息中应该只会在开头部分有一个新的会话，如下面这行：

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

之后是5次复用的会话，如下面这样：

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

大部分情况下你可能不会关心所有的输出，想要快点知道答案。那么可以使用下面这个命令：

```
$ echo | openssl s_client -connect www.feistyduck.com:443 -reconnect -no_ssl2 2> ↵
/dev/null | grep 'New\|Reuse'
New, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES256-GCM-SHA384
Reused, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES256-GCM-SHA384
Reused, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES256-GCM-SHA384
Reused, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES256-GCM-SHA384
Reused, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES256-GCM-SHA384
Reused, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES256-GCM-SHA384
```

下面是该命令的功能介绍。

- -reconnect开关激活了会话复用模式。
- -no_ssl2开关表明我们不希望使用SSL 2连接，这样第一次连接就会使用SSL 3以及更高的版本。老的SSL 2握手格式不支持TLS扩展，如果遇到支持会话票证作为会话复用的服务器就会出现问题。
- 2> /dev/null部分将你不需要关心的标准错误隐藏掉了。
- 最后，通过管道化的grep命令过滤出你关心的那几行。

> **注意**
> 如果在测试会话复用的时候不想使用会话票证，例如有时候不是所有客户端都支持该特性，可以使用-no_ticket开关禁用会话票证。

2.9　检查 OCSP 吊销状态

如果OCSP响应程序出现故障，有时候很难理解到底是为什么。用命令行检测证书吊销状态是可行的，但并不是非常直接。需要执行下面这些步骤。

(1) 获得想要检查吊销状态的证书。

(2) 获得颁发证书。

(3) 确定OCSP响应程序的URL。

(4) 提交OCSP请求并观察响应。

前面两个步骤可以在连接服务器的时候指明-showcerts开关：

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

输出信息的第一张证书属于服务器证书。如果证书链的配置没错的话，第二张证书就是颁发者的证书。为了确保没有问题，需要检查第一张证书的颁发者与第二张证书的使用者是否匹配：

```
---
Certificate chain
 0 s:/1.3.6.1.4.1.311.60.2.1.3=GB/businessCategory=Private Organization
/serialNumber=06694169/C=GB/ST=London/L=London/O=Feisty Duck Ltd ↩
/CN=www.feistyduck.com
   i:/C=US/ST=Arizona/L=Scottsdale/O=Starfield Technologies, Inc./OU=http:/ ↩
/certificates.starfieldtech.com/repository/CN=Starfield Secure Certification ↩
Authority/serialNumber=10688435
-----BEGIN CERTIFICATE-----
MIIF5zCCBM+gAwIBAgIHBG9JXlv9vTANBgkqhkiG9w0BAQUFADCB3DELMAkGA1UE
[30 lines of text removed]
os5LW3PhHz8y9YFep2SV4c7+NrlZISHOZVzN
-----END CERTIFICATE-----
 1 s:/C=US/ST=Arizona/L=Scottsdale/O=Starfield Technologies, Inc./OU=http:/ ↩
/certificates.starfieldtech.com/repository/CN=Starfield Secure Certification ↩
Authority/serialNumber=10688435
   i:/C=US/O=Starfield Technologies, Inc./OU=Starfield Class 2 Certification ↩
Authority
-----BEGIN CERTIFICATE-----
MIIFBzCCA++gAwIBAgICAgEwDQYJKoZIhvcNAQEFBQAwaDELMAkGA1UEBhMCVVMx
[...]
```

如果第二张证书不对的话，需要检测证书链里剩余的证书；有些服务器配置了错误的证书链顺序。如果无法在证书链中找到颁发者的证书，那就得去别的地方找找。有一种方式是通过查找分支证书的颁发机构信息访问（authority information access）扩展：

```
$ openssl x509 -in fd.crt -noout -text
[...]
```

```
    Authority Information Access:
        OCSP - URI:http://ocsp.starfieldtech.com/
        CA Issuers - URI:http://certificates.starfieldtech.com/repository/
_intermediate.crt
[...]
```


如果看到CA颁发者（CA issuer）的信息，就应该能找到颁发者证书的URL。如果颁发者的信息不存在，那么可以尝试在浏览器中打开这个网站，让浏览器构建整个证书链，然后从浏览器的证书查看器里面下载颁发者的证书。如果以上所有的尝试都失败的话，可以在可信库里面进行查找或者访问CA的网站。

如果你已经有了证书，想知道OCSP响应程序的地址，可以使用x509命令带上-ocsp_uri开关：

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

现在可以提交OCSP请求了：

```
$ openssl ocsp -issuer issuer.crt -cert fd.crt -url http://ocsp.starfieldtech.com/ -CAfile issuer.crt
WARNING: no nonce in response
Response verify OK
fd.crt: good
        This Update: Feb 18 17:59:10 2013 GMT
        Next Update: Feb 18 23:59:10 2013 GMT
```

你需要在响应里面查找两个结果。第一，检查响应自身是否有效（上面这个例子是Response Verify OK表示有效），第二，检查响应的结果是什么。如果你看到的状态是good那么表示该证书没有被吊销，如果是revoked则表示该证书已经被吊销了。

> **注意**
> 警告信息里面说的缺少nonce（加密通信中仅使用一次的密钥）的意思是OpenSSL想要使用nonce来防止重放攻击，但是服务器的响应中并没有nonce。这种情况多数是因为CA希望提高他们的OCSP响应程序的性能。当他们禁用了nonce保护（标准允许这么做），OCSP响应能够被制造（一般是批量制造）、缓存并且在一段时间内重用。

你可能会遇到某些OCSP响应程序无法成功响应前面的命令。这种情况下，下面的建议可能会有所帮助。

- 不要请求nonce
  有些服务器无法很好地处理nonce，响应就会出错。默认情况下OpenSSL会请求一个nonce，如果要禁用，可以通过在命令中指定-no_nonce开关。
- 在请求头中提供主机名
  有些HTTP请求没有在Host头中指定正确的主机名，虽然大部分的OCSP服务器都能正常响应，但也有一些不行。如果你遇到的错误里面包含HTTP错误码（例如404），那么可以尝试在OCSP请求的时候带上主机名。在OpenSSL 1.0.0以及更高版本中可以使用-header开关，不过该开关无法在文档中查到。

有了前面这两个要点，最终可以使用下面这行命令：

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

2.10　测试 OCSP stapling

OCSP stapling是一个可选特性，它允许在传输服务器证书的时候带上OCSP响应来验证证书的有效性。因为OCSP响应是通过已经存在的连接进行传输的，所以客户端无需另外单独获取。

只有客户端在握手请求的时候提交status_request扩展，服务器才会使用OCSP stapling。服务器如果支持OCSP stapling的话，就会在握手里面带上OCSP响应信息。

使用s_client工具带上-status 开关来请求OCSP stapling：

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

OCSP相关信息会在连接输出的最开始展现出来。例如，如果服务器不支持stapling的话，在输出的最开始部分就会看到：

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

如果服务器支持stapling，那么会在输出内容中看到完整的OCSP响应：

```
OCSP Response Data:
    OCSP Response Status: successful (0x0)
    Response Type: Basic OCSP Response
    Version: 1 (0x0)
    Responder Id: C = US, O = "GeoTrust, Inc.", CN = RapidSSL OCSP-TGV Responder
    Produced At: Jan 22 17:48:55 2014 GMT
    Responses:
    Certificate ID:
      Hash Algorithm: sha1
      Issuer Name Hash: 834F7C75EAC6542FED58B2BD2B15802865301E0E
      Issuer Key Hash: 6B693D6A18424ADD8F026539FD35248678911630
      Serial Number: 0FE760
    Cert Status: good
    This Update: Jan 22 17:48:55 2014 GMT
    Next Update: Jan 29 17:48:55 2014 GMT
[...]
```

证书状态为good表示该证书未被吊销。

2.11　检查 CRL 吊销状态

与OCSP相比，通过检查证书吊销列表（certificate revocation list，CRL）来检测证书有效性会更复杂，过程如下所示。

(1) 获得你想要检测吊销状态的证书。

(2) 获得颁发证书。

(3) 下载并验证CRL。

(4) 在CRL列表中查找证书序列号。

第一步与OCSP检测方式一样，可以按照2.9节中的步骤实施。

CRL的地址已经编码进服务器证书，可以使用下面的命令取出：

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

从CA获取CRL：

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

验证CRL是有效的（例如，是由颁发者的证书签名的）：

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

现在，确定希望进行检测的证书的序列号：

```
$ openssl x509 -in fd.crt -noout -serial
serial=0FE760
```

此时，可以将CRL转换成可读的的格式，然后手动检查：

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

            X509v3 CRL Number:
                92103
Revoked Certificates:
    Serial Number: 0F38D7
        Revocation Date: Nov 26 20:07:51 2013 GMT
    Serial Number: 6F29
        Revocation Date: Aug 15 20:48:57 2011 GMT
[...]
    Serial Number: 0C184E
        Revocation Date: Jun 13 23:00:12 2013 GMT
    Signature Algorithm: sha1WithRSAEncryption
         95:df:e5:59:bc:95:e8:2f:bb:0a:4f:20:ad:ca:8f:78:16:54:
         35:32:55:b0:c9:be:5b:89:da:ba:ae:67:19:6e:07:23:4d:5f:
         16:18:5c:f3:91:15:da:9e:68:b0:81:da:68:26:a0:33:9d:34:
         2d:5c:84:4b:70:fa:76:27:3a:fc:15:27:e8:4b:3a:6e:2e:1c:
         2c:71:58:15:8e:c2:7a:ac:9f:04:c0:f6:3c:f5:ee:e5:77:10:
         e7:88:83:00:44:c4:75:c4:2b:d3:09:55:b9:46:bf:fd:09:22:
         de:ab:07:64:3b:82:c0:4c:2e:10:9b:ab:dd:d2:cb:0c:a9:b0:
         51:7b:46:98:15:83:97:e5:ed:3d:ea:b9:65:d4:10:05:10:66:
         09:5c:c9:d3:88:c6:fb:28:0e:92:1e:35:b0:e0:25:35:65:b9:
         98:92:c7:fd:e2:c7:cc:e3:b5:48:08:27:1c:e5:fc:7f:31:8f:
```

```
0a:be:b2:62:dd:45:3b:fb:4f:25:62:66:45:34:eb:63:44:43:
cb:3b:40:77:b3:7f:6c:83:5c:99:4b:93:d9:39:62:48:5d:8c:
63:e2:a8:26:64:5d:08:e5:c3:08:e2:09:b0:d1:44:7b:92:96:
aa:45:9f:ed:36:f8:62:60:66:42:1c:ea:e9:9a:06:25:c4:85:
fc:77:f2:71
```

在CRL的开头是一些元数据，之后是吊销证书的列表，结尾部分是签名（我们在上一步中验证过）。如果服务器证书的序列号在这份列表里面，意味着该证书已经被吊销了。

如果不想逐行寻找序列号（有一些CRL可能会很长），那就用grep查找它，但是需要小心，确保格式正确（例如，如有必要，可以移除最开始的0x，删除所有在开头的0，并且将所有字母变成大写）。例如：

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

2.12　测试重新协商

s_client工具还有几个可以协助你手动进行重新协商测试的特性。首先，当你连接的时候，s_client会报告远程服务器是否支持重新协商。因为服务器如果支持安全重新协商的话，那么它会在握手阶段通过交换特殊的TLS扩展来表明支持该特性。如果服务器支持，输出可能是这样的（我已将重点加粗）：

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

如果服务器不支持安全重新协商，输出会有一些不同：

```
Secure Renegotiation IS NOT supported
```

即便服务器表明它支持安全重新协商，你也想测试一下他是否允许客户端开始这个重新协商。由客户端发起的重新协商（client-initiated renegotiation）是一个在实际中用不到的协议特性（因为如果有必要的话都可以由服务器发起重新协商），该特性会使得服务器更容易受到拒绝式服务攻击。

要开始重新协商，需要在单独一行中输入R字符。例如，假设我们正在与HTTP服务器通信，你可以在第一行发送一个请求，初始化重新协商，然后结束请求。在向一个支持客户端发起重新协商的Web服务器发起请求的时候，会出现以下内容：

```
HEAD / HTTP/1.0
R
RENEGOTIATING
depth=3 C = US, O = "VeriSign, Inc.", OU = Class 3 Public Primary Certification Authority
verify return:1
depth=2 C = US, O = "VeriSign, Inc.", OU = VeriSign Trust Network, OU = "(c) 2006 ↩
VeriSign, Inc. - For authorized use only", CN = VeriSign Class 3 Public Primary ↩
```

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

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

read:errno=0
```

当发生了重新协商，服务器会重新将它的证书发送给客户端。你可以在输出内容中看到对证书链的校验过程，在那之后是Host请求头。能够看到Web服务器的响应就证明服务器是支持重新协商的。由于在不同版本的SSL/TLS库中发现了不同形式的重新协商问题，不支持重新协商的服务器可能会直接断开连接，或者即使保持连接处于打开状态，也会拒绝在该连接上继续通信（结果通常是超时）。

不支持重新协商的服务器会直截了当地拒绝在该连接上进行第二次握手：

```
HEAD / HTTP/1.0
R
RENEGOTIATING
140003560109728:error:1409E0E5:SSL routines:SSL3_WRITE_BYTES:ssl handshake failure:s3_pkt.c:592:
```

撰写本书的时候，OpenSSL默认是按照服务器不支持安全重新协商的方式进行连接的。它也支持安全和不安全的重新协商，由服务器作出选择。如果与一个不支持安全重新协商的服务器进行重新协商并且成功，表明该服务器支持由客户端发起的不安全重新协商。

> **注意**
>
> 测试不安全重新协商最可靠的方式是采用本节使用的方法，但是要使用那些有重新协商功能的OpenSSL版本（例如，0.9.8k）。我提及这个的主要原因是有部分服务器同时支持安全和不安全的重新协商，而现代版本的OpenSSL只支持安全重新协商选项，导致很难检测该漏洞。

2.13　测试 BEAST 漏洞

BEAST攻击利用TLS 1.1之前所有SSL和TLS协议的漏洞。该漏洞影响所有CBC套件以及客户端和服务器的数据流；然而BEAST攻击只对客户端有效。现代浏览器使用被称为1/*n*–1分割的工作区来阻止被利用，不过一些服务器仍然在它们这一端部署减缓措施，特别是如果它们的一些用户还使用老的浏览器（未打补丁）。

理想的解决方式是使用TLS 1.1以及更高版本，但是支持这些新协议的客户端还不够多。事实上RC4本身现在也被认为不安全，这导致情况变得更加复杂。如果你认为BEAST比RC4更危险，那么应该为使用新浏览器的用户部署TLS 1.2，对其他所有人使用RC4。

- *严格缓解*

  如果使用TLS 1.0及之前的协议，那么不要使用任何CBC套件，只启用RC4套件。不支持RC4的客户端无法协商安全的连接，这种模式排除了网站潜在的一些用户，但是一些PCI的评审员对此有要求。

- RC4优先

  因为只有很小一部分客户端不支持RC4，第二种方式是启用CBC套件，但是对所有支持RC4的客户端强制使用RC4套件。这种方式为大部分用户提供了保护。

你对服务器行为的期望决定了如何进行测试。在两种方式下，通过使用`-no_ssl2`、`-no_tls_1_1`以及`-no_tls_1_2`开关来确保只使用了不安全的协议。

为了测试严格缓解的方式，可以尝试让客户端在连接的时候禁用所有的RC4套件：

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

如果连接成功了（只有在使用了有漏洞的CBC套件时才有可能），你就可以确定严格缓解的方案并没有生效。

为了测试RC4优先的方式，可以尝试在连接的时候将RC4放在密码套件列表的最后面：

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

支持RC4优先的服务器会为连接选择其中一种RC4套件，并且忽略所有其他CBC套件。如果再看看别的，可以发现服务器并没有使用任何缓解BEAST方式。

2.14　测试心脏出血

你可以手动测试心脏出血，也可以使用一些工具（因为心脏出血漏洞非常容易利用，所以有非常多的工具），不过这类工具的准确性都存在疑问。一些证据表明有些工具无法成功检测有漏洞的服务器。[①]考虑到心脏出血的严重性，最好手动测试或者使用那些可以让你看到整个过程的

① Bugs in Heartbleed detection scripts，https://www.hut3.net/blog/cns---networks-security/2014/04/14/bugs-in-heartbleed-detection-scripts-（Shannon Simpson和Adrian Hayter，2014年4月14日）。

工具。我会用一个修改过的OpenSSL版本来描述你可以使用的一种方式。

如果你的OpenSSL版本支持心跳协议的话（1.0.1及更高版本），测试的某些部分不需要对OpenSSL进行修改。例如，如果要测试远程服务器是否支持心跳协议，请在连接的时候使用-tlsextdebug开关显示服务器扩展：

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

不支持心跳扩展的服务器不会受到心脏出血漏洞的影响。要测试服务器是否会响应心跳请求，使用-msg开关请求将该协议消息展示出来，然后连接服务器，输入B并按回车：

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

输出内容显示了一对完整的心跳请求和响应。两条心跳消息的第二和第三字节是载荷（payload）的长度。我们提交了18字节（十六进制为12）的载荷，然后服务器返回了同样长度的载荷。两个例子中还有额外的16字节的填充。载荷的开始两字节组成了序列号，是OpenSSL用来匹配请求以及响应的。剩余的载荷以及填充部分都是随机数据。

要测试服务器是否有漏洞，我们需要准备一个特别的OpenSSL版本，它可以发送不正确的载荷长度。有漏洞的服务器会接受攻击者宣称的载荷长度，并且会以同样字节的内容作出响应，而不管攻击者是否真的提供了这么长的载荷。

此时你要决定是否想要创建具有攻击性的测试（通过获取进程的数据来攻击服务器）或者非攻击性的测试，这取决于你的环境。如果你有测试的许可，那么就采取攻击性测试。在这种情况下，你可以清楚地看到返回的内容，而且不会出现误判。例如，某些版本的GnuTLS支持心跳而且会响应带有不正确载荷长度的请求，但是却不会返回服务器数据。非攻击性测试无法可靠地检测出这种情况。

下面针对OpenSSL 1.0.1h的补丁形成了一个非攻击性测试的版本：

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

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

-               if (payload == 18 && seq == s->tlsext_hb_seq)
+               if ((payload == (18 + PAYLOAD_EXTRA)) && seq == s->tlsext_hb_seq)
                        {
                        s->tlsext_hb_seq++;
                        s->tlsext_hb_pending = 0;
@@ -2705,7 +2706,7 @@
        /*  消息类型 */
        *p++ = TLS1_HB_REQUEST;
        /* 载荷长度（18字节） */
-       s2n(payload, p);
+       s2n(payload + PAYLOAD_EXTRA, p);
        /* 序列号 */
        s2n(s->tlsext_hb_seq, p);
        /* 16随机字节 */
```

要建立非攻击性测试，将载荷长度增加到16字节或者将填充的长度增加到16字节。漏洞服务器对这样的请求作出响应时，将返回填充而不会返回任何其他内容。要建立攻击性测试，则将载荷长度增加到32字节。漏洞服务器会以50字节的载荷（OpenSSL默认发送的18字节加上你的32字节）作出响应并发送16字节的填充信息。通过使用这种方式增加发送载荷的长度，漏洞服务器最终会发送64 KB的数据。对心脏出血免疫的服务器则不会进行响应。

为了建立你自己的心脏出血测试工具，解压缩一个全新的OpenSSL源代码，请编辑ssl/t1_lib.c，按照补丁进行修改，然后像平时一样进行编译，但是不要进行安装。最终的openssl二进制会被放在apps/子目录中。因为采用的是静态编译，你可以将它重命名为类似openssl-heartbleed，然后移动到某个固定的地方。

下面是一个输出的例子，你可以获得漏洞服务器返回的16字节服务器数据（以粗体显示）：

```
B
HEARTBEATING
>>> TLS 1.2 [length 0025], HeartbeatRequest
    01 00 32 00 00 7c e8 f5 62 35 03 bb 00 34 19 4d
    57 7e f1 e5 90 6e 71 a9 26 85 96 1c c4 2b eb d5
    93 e2 d7 bb 5f
<<< TLS 1.2 [length 0045], HeartbeatResponse
    02 00 32 00 00 7c e8 f5 62 35 03 bb 00 34 19 4d
```

```
    57 7e f1 e5 90 6e 71 a9 26 85 96 1c c4 2b eb d5
    93 e2 d7 bb 5f 6f 81 0f aa dc e0 47 62 3f 7e dc
    60 95 c6 ba df c9 f6 9d 2b c8 66 f8 a5 45 64 0b
    d2 f5 3d a9 ad
read R BLOCK
```

如果你想要在一个响应中看到更多的数据，可以增加载荷的长度，重新编译，然后再测试一遍。也可以通过再次输入B命令，获取同样长度的另外一批数据。

2.15 确定 Diffie-Hellman 参数的强度

在OpenSSL 1.0.2以及之后的版本中，当你在连接服务器的时候，如果使用了临时Diffie-Hellman，s_client命令会输出密钥的强度。因此，要确定某些服务器DH参数的强度，你所要做的就是，在连接服务器的时候只提供那些将DH用作密钥交换的套件。例如：

```
$ openssl-1.0.2 s_client -connnect www.feistyduck.com:443 -cipher kEDH
[...]
---
No client certificate CA names sent
Peer signing digest: SHA512
Server Temp Key: DH, 2048 bits
---
[...]
```

支持出口套件的那些服务器可能提供了更加不安全的DH参数。为了检测这种情况，在连接的时候只提供出口DHE套件：

```
$ openssl-1.0.2 s_client -connnect www.feistyduck.com:443 -cipher kEDH+EXPORT
```

对那些正确配置的服务器，这条命令会失败。否则你会发现服务器会协商不安全的512位的DH参数。

附录 A

SSL/TLS部署最佳实践

附录A包括“SSL/TLS部署最佳实践”（版本1.4，日期为2014年12月8日）文档的全部内容[①]。我维护的这份文档是对所有SSL/TLS部署相关内容的简明高级概述。此处还包含来自Qualys（https://www.qualys.com/）的许可。

A.1 私钥和证书

TLS提供的保护程度完全依赖私钥和证书。私钥是安全的基础，证书将服务器的身份信息传输给用户。

A.1.1 使用 2048 位的私钥

所有服务器上都要使用2048位的RSA或者256位的ECDSA私钥。这种长度的密钥在较长一段时间内都是安全的。如果你还在生产环境中使用1024位的RSA密钥，要尽快替换成更强的密钥。如果你需要使用超过2048位的安全，那么可以考虑使用ECDSA密钥，它的性能更高一些。ECDSA的缺点是有一小部分客户端不支持ECDSA，所以你可能需要同时部署RSA和ECDSA密钥来保证兼容性。

A.1.2 保护私钥

要把私钥当作重要的资产，只让尽可能少的人有密钥的访问权限。以下是几个建议策略。

- 在可信的电脑上生成私钥和证书签名申请（certificate signing request，CSR）。有一些CA提供私钥和CSR文件的生成服务，但其实并不合适。
- 将私钥存储在备份系统中的时候，使用密码对其进行保护以防泄露。在生产环境中使用密码保护私钥没什么效果，有经验的攻击者可以直接从进程的内存中获得私钥。可以使用一些硬件设备来保证私钥的安全，即便发生服务器被入侵的情况也可以保证私钥不泄露；但是这些设备非常昂贵，只有少数对安全要求非常高的组织才会用到。
- 密钥泄露之后，需要吊销老的证书，并且生成新的密钥。

① SSL/TLS Deployment Best Practices，https://www.ssllabs.com/projects/best-practices/（Qualys SSL Labs）。

- 每年更新证书的时候都需要生成新的密钥。

A.1.3　确保覆盖足够多的主机名

确保你的证书能够覆盖站点需要用到的所有名称。例如你的主要名称是www.example.com，但是你还有www.example.net。你的目标是防止出现非法证书警告。这些警告会让用户感到迷惑，降低他们对网站的信任。

记住，即便在你的服务器上面只配置了一个名称，你也无法控制用户如何到来，以及其他站点如何与你链接。大多数情况下，你需要保证证书在有www前缀和没有www前缀的时候，网站都能正常访问（例如，证书需要覆盖example.com和www.example.com）。一般的经验是Web服务器的证书需要覆盖所有指向这台服务器的DNS名称。

泛域名证书有它们的使用场景，但是如果很多人都需要使用，特别是那些跨越了组织边界的人，那么尽量不要使用泛域名证书，因为这很容易让私钥的管理失控。换句话说，能访问私钥的人越少越好。需要认识到，共享证书会将不同站点绑定在一起，一个站点的漏洞会导致使用同样证书的其他站点也出现漏洞。

A.1.4　从可信赖的 CA 获得证书

选择一家可信赖的、对证书业务和安全非常重视的*证书颁发机构*（certification authority，CA）。选择CA的时候需要考虑以下几点。

- *安全性*

  所有CA都需要定期进行审计（否则不可以运营CA），不过一些CA公司更注重安全。想要知道哪家公司更好不是一件容易的事情；有一个办法是检测他们的历史情况，并且需要关注他们如何应对入侵以及是否从错误中吸取了教训。

- *可观的市场份额*

  满足这个条件的CA一般不会轻易召回他们颁发的证书，而过去有一些小公司出现过这种情况。

- *业务重点*

  CA如果出了严重的问题，那么将完全丧失这一块业务，因此他们会比较重视证书部门，而不会去追求其他有着潜在更高利润的机会。

- *提供的服务*

  你选择的CA至少需要支持*证书吊销列表*（certificate revocation list，CRL）和*在线证书状态协议*（online certificate status protocol，OCSP），而且OCSP服务的性能要好。他们还应该提供域名验证和*扩展验证*（extended validation，EV）的证书，并且可以让你选择公钥的算法。（现在大部分网站都使用RSA，不过ECDSA因为在性能上存在优势，未来会扮演更重要的角色）。

- *证书管理选项*

如果你需要管理数量非常多的证书，运维非常复杂的环境，那么选择CA的时候可以关注他们提供的管理工具。

- 技术支持

  选择一家有良好技术支持的CA，当你需要的时候，他们可以提供非常好的支持。

A.1.5　使用强证书签名算法

证书签名的安全性取决于用来进行签名的私钥和所使用的散列函数的长度。现在，大多数证书都采用的SHA1散列函数已经不再安全了。整个行业必须在2016年底完成SHA1证书的迁移，之后SHA1证书将完全无法使用。[①]

然而，Google的Chrome浏览器会对那些在最终期限之前还未过期的SHA1证书进行警告[②]，因此你应该立刻将所有过期时间在2015年之后的SHA1证书替换掉。另外一种方式是直接使用那些基于SHA2算法集的证书。但是，在这之前，检查一下你的用户是否都已经支持SHA2了。有一些老的客户端是不支持SHA2算法的，比如在Windows XP Service Pack 2上运行的IE6（在某些国家和组织中还广泛使用）。

A.2　配置

正确配置TLS服务器之后，你可以将站点的凭据展示给访问用户，让他们知道站点使用了安全的加密方式，所有的问题都已经解决了。

A.2.1　部署正确的证书链

大多数情况下只部署服务器证书是不够的，一般需要两个或者更多的证书才能构建一个完整的信任链条。有一个常见的错误是，正确配置了服务器证书，但是忘记将其他证书包含进来。虽然其他那些证书一般都会有很长的有效期，但是它们也会过期。当它们过期之后，就会使得整个证书链无效。你的CA应该能够提供你所需要的所有额外证书。

非法的证书链会致使正确的服务器证书变成非法证书，浏览器会出现警告信息。这类问题在实践中有时候很难排查，因为有一些浏览器可以应对这类问题，它们会重新构建一个完整、正确的证书链，不过有一些浏览器不会。

A.2.2　使用安全的协议

SSL/TLS家族里面一共有5种协议：SSL v2、SSL v3、TLS v1.0、TLS v1.1和TLS v1.2。它们

---

① SHA1 Deprecation Policy，http://blogs.technet.com/b/pki/archive/2013/11/12/sha1-deprecation-policy.aspx（Windows PKI blog，2013年11月12日）。

② Gradually Sunsetting SHA-1，http://blog.chromium.org/2014/09/gradually-sunsetting-sha-1.html（The Chromium Blog，2014年9月5日）。

的特点如下所示。

- SSL v2是不安全的，绝对不能使用它。
- SSL v3在使用HTTP的时候是不安全的，在与别的协议一起使用的时候是弱协议。
- TLS v1.0大体来说是安全的，我们还不清楚它与HTTP之外的协议一起使用是否有重要安全缺陷；不过当它与HTTP一起使用的时候，经过小心的配置，TLS v1.0还是可以变得安全的。
- TLS v1.1和v1.2还没有已知的安全问题。

TLS v1.2应该作为你最主要的协议。这个版本更高级，因为它提供了其他早期协议版本里面没有的重要特性。如果你的服务器平台（或者其他中间设备）不支持TLS v1.2的话，应该加快制定升级计划。如果你的服务提供方不支持TLS v1.2，请要求他们升级。

为了支持更老的客户端，你暂时需要继续支持TLS v1.0和TLS v1.1。通过某些变通方式（A.2.3节中会介绍），这些协议对大多数的站点来说还是足够安全的。

A.2.3　使用安全的密码套件

为了保证通信安全，首先需要确定你是在与你希望的团队进行通信（而不是与其他可能会进行窃听的第三方），同时需要安全地交换数据。在SSL和TLS中，密码套件用来定义如何进行安全的通信。密码套件以通过多元性实现安全为理念，是由不同的积木组合而成的。如果发现哪一块积木变弱或者不安全了，可以切换到另外一块。

你的最终目标是只使用那些提供大于等于128位的验证和加密强度的套件。其他所有套件都应该去除掉。

- 匿名Diffie-Hellman（anonymous Diffie-Hellman，ADH）套件不提供身份验证。
- NULL密码套件不提供加密。
- Export密码交换套件使用的身份验证方式很容易被破解。
- 对于使用弱密码的套件（通常是40或者56位），其加密很容易被破解。
- RC4比之前更加不安全了[①]。你应该尽快去掉RC4，不过在这之前需要检测去掉之后可能产生的互操作性影响。
- 3DES提供大约112位的安全性。这低于我们推荐的最少128位，不过还是足够安全的。3DES更大的问题是它比其他算法慢很多。因此，出于性能的原因我们不推荐使用它，不过可以将它放在密码列表的末尾，来兼容那些非常老的客户端。

A.2.4　控制密码套件选择

在SSL v3和之后的版本中，客户端会提交一组它们支持的密码套件，然后服务器会从这个列表中选择一个套件来协商一个安全的通道。然而不是所有服务器都会这样做，有一些服务器会选

① On the Security of RC4 in TLS and WPA，http://www.isg.rhul.ac.uk/tls/（Kenny Paterson等，2013年3月13日）。

择列表中第一个它能够支持的套件。让服务器能够选择正确的套件对于安全来说是非常关键的（有关此问题的更多信息，请参考A.2.7节）。

A.2.5　支持向前保密

向前保密[①]是既能够做到安全通话、又不依赖服务器私钥的一种协议特性。使用那些不支持向前保密的密码套件，如果哪个人可以恢复出服务器私钥，那么就可以解密所有之前被记录下来的已经加密的通话。你需要支持并且优先使用ECDHE套件，这样就能为现代Web浏览器启用向前保密。为了支持更多的客户端，你还应该使用DHE算法作为ECDHE的备份。[②]

A.2.6　禁用客户端发起的重新协商

在SSL/TLS中，重新协商允许各方为了重新协商通信安全，停止交换数据。有一些例子是重新协商需要由服务器发起，但是客户端却没有必要先发起。更重要的是，支持客户端发起的重新协商会导致你的服务器更容易遭受到拒绝服务（denial of service，DoS）攻击。[③]

A.2.7　解决已知问题

没有什么东西是绝对安全的，在任何时候都有可能出现安全问题。持续关注安全领域发生的事情是一个非常好的习惯，能够及时应对各种情况。最起码，当供应商发布补丁的时候，你应该尽快打上补丁。

你需要注意以下这些情况。

- 禁用不安全的重新协商

  在2009年的时候，重新协商特性不再安全，整个协议需要升级[④]。大多数供应商都打了相应的补丁，最起码也提供了应对方式。不安全的重新协商非常危险，因为可以非常容易地利用它们，产生类似跨站点请求伪造（cross-site request forgery，CSRF）的效果，某些情况下还会产生跨站点脚本（cross-site scripting，XSS）的效果。

- 禁用TLS压缩

  2012年的CRIME攻击[⑤]显示攻击者是如何利用TLS压缩来发现部分敏感数据的（例如，会

---

① Deploying Forward Secrecy，https://community.qualys.com/blogs/securitylabs/2013/06/25/ssl-labs-deploying-forward-secrecy（Qualys Security Labs，2013年6月25日）。

② Increasing DHE strength on Apache 2.4.x，http://blog.ivanristic.com/2013/08/increasing-dhe-strength-on-apache.html（Ivan Ristić的博客，2013年8月15日）。

③ TLS Renegotiation and Denial of Service Attacks，https://community.qualys.com/blogs/securitylabs/2011/10/31/tls-renegotiation-and-denial-of-service-attacks（Qualys Security Labs Blog，2011年10月）。

④ SSL and TLS Authentication Gap Vulnerability Discovered，https://community.qualys.com/blogs/securitylabs/2009/11/05/ssl-and-tls-authentication-gap-vulnerability-discovered（Qualys Security Labs Blog，2009年11月）。

⑤ CRIME: Information Leakage Attack against SSL/TLS，https://community.qualys.com/blogs/securitylabs/2012/09/14/crime-information-leakage-attack-against-ssltls（Qualys Security Labs Blog，2012年9月）。

话Cookie）。那时候只有非常少的客户端支持TLS压缩（现在就更少了），也就是说在你的服务器上禁用TLS压缩并不会产生任何性能问题。针对TLS压缩的攻击风险比较低。

- 解决由HTTP压缩产生的信息泄露

  2013年披露了CRIME攻击的两个变种，即TIME和BREACH攻击。这两种攻击不是针对TLS压缩（CRIME才是针对TLS压缩），而是关注使用HTTP压缩进行压缩的HTTP响应体中的密码。考虑到HTTP压缩对很多公司来说非常重要，这类问题非常难处理。完全解决需要修改应用代码。[①]TIME和BREACH攻击需要非常多的资源才能实施，不过如果有人有做此事的强烈动机的话，产生的影响等同于CSRF。

- 禁用RC4

  RC4加密已经不再安全了，应该禁用。[②]当前我们知道的最快攻击需要百万次的请求，也就是非常多的带宽和时间。因此风险相对来说比较低，不过未来的攻击手段可能会提高。在移除RC4之前，需要检查你当前的用户是否会受到影响；换句话说，检查你的用户是否只支持RC4。

- 认真对待BEAST攻击

  2011年公布的BEAST攻击[③]的目标是2004年的TLS 1.0以及更早版本的漏洞；之前，这个漏洞被认为无法在实际中利用。成功的BEAST攻击类似于会话劫持。虽然这个问题出现在客户端，但在开始的一段时间内，由服务器去解决BEAST攻击是合理的。不幸的是，要从服务器去解决这个问题，就必须引入RC4，而这个算法已经不再推荐使用了。同时，因为大部分客户端都解决了BEAST攻击，所以我们不再推荐从服务器解决这个问题[④]。在某些情况下，如果有大量的老旧客户端会受到BEAST攻击，也许使用RC4、TLS 1.0或者之前的协议版本是更安全的一种做法。只有在完全了解当前的环境及其威胁模型之后才能作出决定。

- 禁用SSL v3

  SSL v3受到2014年10月公布的POODLE攻击威胁[⑤]。此攻击很容易被利用来攻击HTTP客户端，让其运行JavaScript恶意软件。客户端也很容易被攻击者欺骗，从一个更安全的协议（例如，TLS v1.2）降级到不安全的SSL v3。最好的解决方案是完全禁用SSL v3，这对绝大多数站点来说都可以放心实施。

---

① Defending against the BREACH Attack，https://community.qualys.com/blogs/securitylabs/2013/08/07/defending-against-the-breach-attack（Qualys Security Labs，2013年8月7日）。

② Internet-Draft: Prohibiting RC4 Cipher Suites，https://datatracker.ietf.org/doc/draft-ietf-tls-prohibiting-rc4/（A.Popov，2014年10月1日）。

③ Mitigating the BEAST attack on TLS，https://community.qualys.com/blogs/securitylabs/2011/10/17/mitigating-the-beast-attack-on-tls（Qualys Security Labs Blog，2011年10月）。

④ Is BEAST Still a Threat，https://community.qualys.com/blogs/securitylabs/2013/09/10/is-beast-still-a-threat（Qualys Security Labs，2013年9月10日）。

⑤ This POODLE bites: exploiting the SSL 3.0 fallback，http://googleonlinesecurity.blogspot.com/2014/10/this-poodle-bites-exploiting-ssl-30.html（Google Online Security Blog，2014年10月14日）。

A.3　性能

在这份指导文档中，安全是我们的主要关注点，但我们也必须考虑性能问题。安全服务如果不能满足性能需求就会被遗弃。然而，因为TLS配置通常不会带来很大的性能开销，我们把讨论限定在会导致严重性能下降的常见配置问题上。

A.3.1　不要太安全了

建立安全连接过程中的密码握手的开销是由私钥长度决定的。使用的密钥过短会不安全，使用的密钥过长会导致“太安全”了，性能上会让人无法忍受。对于大多数网站，使用超过2048位的RSA密钥或者超过256位的ECDSA密钥会极大地浪费CPU，可能还会影响用户体验。同样，在临时密钥交换中使用超过2048位的DHE和256位的ECDHE的意义也不大。

A.3.2　确保正确使用会话恢复

会话恢复是一种性能优化技术，将耗时的密码计算操作的结果保存下来，以便在一段时间里重复使用。如果禁用会话恢复机制或者使用不当的话，会导致性能严重下降。

A.3.3　使用持久链接（HTTP）

今天，绝大多数TLS开销并非来自CPU密集型的密码计算操作，而是网络延迟。一个TLS握手是建立在TCP握手结束后，它需要交换更多的数据包。为了让网络延迟最小化，你需要启用HTTP持久化（连接保持），从而让你的用户能在一个TCP连接上发起多次HTTP请求。

A.3.4　为公共资源启用缓存（HTTP）

当使用TLS通信时，浏览器会假设所有的流量都是敏感信息。浏览器会把一些特定的资源缓存到内存中，一旦关闭浏览器，所有内容就丢失了。为了提升性能，对某些资源启用长期缓存，可通过将`Cache-Control: public`响应头附加到公共资源（例如，图片），将这些公共资源标记为公有。

A.3.5　使用 OCSP stapling

OCSP stapling是改版的OSCP协议，使得在TLS握手过程中就可以直接从服务器传递证书吊销信息给浏览器。因此，浏览器无需额外联系OCSP服务器来验证证书，从而大幅降低连接耗时。

A.4　应用设计（HTTP）

在SSL诞生后，HTTP协议和Web应用周边的平台仍然在不断地进化。进化的结果就是今天平

台上包含的一些特性可以对付加密。在本节中，我们会罗列出这些特性，并介绍如何安全地使用它们。

A.4.1　100%加密你的网站

事实上“加密只是一个可选项”的想法大概是今天最严重的安全问题之一。我们来看看以下这些问题。

- 网站应该支持TLS但没有
- 网站支持TLS但不是强制的
- 网站混合了TLS和非TLS的内容，甚至有时候在相同的网页上出现
- 网站编程错误导致TLS被攻陷

如果你知道自己在做什么的话，这些问题大部分是可以避免的，但是唯一可以有效保护网站通信安全的方式是强制对所有的内容进行加密，无一例外。

A.4.2　避免混合内容

混合内容的页面是指那些已经使用TLS，但有些资源（例如，JavaScript文件、图片和CSS）是通过非TLS的方式传输的。这类页面是不安全的，例如，主动中间人攻击者可以劫持其中一个不受保护的JavaScript的资源，从而劫持整个用户会话。就算你遵循了前面的建议，加密了自己网站上的所有内容，但也不排除来自第三方网站的资源是没有加密的。

A.4.3　理解和认知第三方信任

网站经常会通过第三方服务器提供的JavaScript来使用别人的服务，例如有非常多的站点在使用Google Analytics。内含的第三方代码创建了一个隐式的信任链，让第三方可以完全控制你的网站。第三方本身可能并没有恶意，但他们很容易成为攻击者的目标。原因很简单，如果一个大型第三方提供商被攻陷，那攻击者自然而然地就攻破了所有依赖这个服务的站点。

如果你采纳了A.4.2节的建议，至少你的第三方链接在加密后可以防止中间人攻击。此外，你应该进一步了解自己站点使用的那些服务，要么去除或替换它们，要么承担继续使用的风险。

A.4.4　安全 Cookie

为了安全，网站需要TLS，但是网站使用的Cookie也要标记为安全。如果不能保护Cookie，即使网站本身是100%加密的，中间人攻击者依旧有可能使用某些手段来获取信息。

A.4.5　部署 HSTS

HTTP严格传输安全（HTTP strict transport security，HSTS）是TLS协议的保护伞，它的设计能在存在配置和实现错误的情况下保证安全性。设置一个响应头就能让你的站点支持HSTS，这

样那些支持HSTS的浏览器（目前有Chrome、Firefox、Safari和Opera；IE很快就会支持）就会强制使用HSTS。

HSTS的目的很简单：在使用HSTS之后，所有与网站的不安全通信都是不允许的。这一目标通过自动把明文链接转换成安全链接来实现。一个额外的特性是不允许用户绕过证书警告（证书警告是中间人攻击的标志，而研究表明大多数用户都会无视警告，所以最好永远禁止用户这么做）。

支持HSTS是一项能大幅提高网站TLS安全性的措施。新的网站应该在设计的时候就考虑到HSTS，而旧的站点则应该尽快支持。

A.4.6　禁用敏感内容的缓存

这个建议的目的是保证敏感信息只会发送给应该发送的人。尽管代理服务器看不到加密流量，不能把它共享给别人，但是随着基于云的应用分发平台如雨后春笋般出现，你必须小心区分公开资源和敏感内容。

A.4.7　确保没有其他漏洞

这一点是提醒大家，TLS并不等于完全安全。TLS的设计只是解决安全的某一方面，即你和用户通信过程的保密和完整，但你还得面对其他威胁。大多数情况下，你需要确保你的站点没有别的问题。

A.5　验证

在配置的时候有很多可以进行调整的参数，而在改动之前，有时候很难完全确定修改之后会有什么影响。此外，有些时候改动可能是无意的；软件的升级也会悄悄引入一些变化。因此，我们建议使用一款详细的SSL/TLS评估工具来检查你的配置是否安全，并定期运行检查，确保站点一直保持安全。对于公开站点，我们推荐使用SSL Labs网站上的免费在线工具（https://www.ssllabs.com/ssltest/）。它的*握手模拟*（handshake simulation）功能在实践中非常有用，可以展示出各种常用的客户端在握手过程中到底使用了哪些安全参数。

A.6　高级议题

下面的这些议题超出了这份文档的范畴，需要对SSL/TLS和*公钥基础设施*（PKI）有更深的理解，而且这些议题依然存在争议。

- *扩展验证证书*

  EV证书是高保证的证书，只有经过充分的线下审核后才给予颁发[①]。证书的目的是提供组

① About EV SSL Certificates，https://www.cabforum.org/certificates.html（CAB论坛网站）。

织与其在线网站身份的强关联。EV证书更难伪造，提供了更好的安全性，并且在浏览器上呈现给用户时的待遇也更高。

- 公钥钉扎

  设计公钥钉扎的目的是让网站的运维人员可以限制只有某些CA才可以签发他们网站的证书。这个特性由Google发布了一段时间（硬编码到他们的Chrome浏览器里面），经证明对避免攻击非常有效，引发了大家的关注。2014年，Firefox也加入了对硬编码钉扎的支持。一个叫作HTTP的公钥钉扎扩展（public key pinning extension for HTTP，https://datatracker.ietf.org/doc/draft-ietf-websec-key-pinning/）的标准已经开发了很长时间，很快就会发布。我们期待这个特性今后至少会得到主流浏览器的支持。

- ECDSA私钥

  如今大多数网站都依赖RSA私钥。这个算法对Web通信安全来说非常关键，因此针对它的攻击一直在改进。在此之前，使用1024位RSA密钥的站点几乎都迁移到了2048位。不过有一些让人担忧的是，随着RSA密钥长度的增加，可能会出现性能问题。椭圆曲线加密（elliptic curve cryptography，ECC）使用了不同的数学方法，能用更短的密钥提供更强的安全性。因此RSA密钥可以被ECDSA替代。目前只有少数CA支持ECDSA，但我们期待未来会有更多。在迁移ECDSA的时候，一个主要的问题是并非所有客户端都支持它。如果你考虑使用ECDSA，需要确认它是否会影响用户连接你的服务器。有些平台支持双密钥部署，可以让你同时使用RSA和ECDSA以适配所有客户端。

A.7　改动

这份文档的最初版本是在2012年2月24日发布的。本节记录了从版本1.3开始，文档修改的时间。

A.7.1　版本 1.3（2013 年 9 月 17 日）

此版本中存在以下更改。

- 推荐替换1024位证书
- 推荐禁用SSL v3
- 删去了推荐服务器使用RC4来避免BEAST攻击
- 推荐禁用RC4
- 推荐在未来禁用3DES
- 警告关于CRIME攻击的变种（TIME和BREACH攻击）
- 推荐支持向前保密
- 加入对ECDSA证书的讨论

A.7.2　版本 1.4（2014 年 12 月 8 日）

此版本中存在以下更改。

- 讨论SHA1的消亡，推荐迁移到SHA2系列算法
- 推荐禁用SSL v3，提及POODLE攻击
- 扩展A.3.1节，涵盖DHE和ECDHE密钥交换的强度
- 在A.3.5节中加入了推荐OCSP stapling作为性能提高的手段

A.8　致谢

特别感谢Marsh Ray、Nasko Oskov、Adrian F. Dimcev和Ryan Hurst提供有价值的反馈并帮助起草这份文档。也感谢其他向整个世界慷慨分享信息安全和密码学知识的那些人。之所以能够在这里呈现这份指导手册，离不开整个安全社区的努力。

A.9　关于 SSL Labs

SSL Labs（https://www.ssllabs.com/）是Qualys为了理解SSL/TLS和PKI而进行的研究，同时提供工具和文档协助进行评估和配置。SSL Labs从2009年启动，通过使用线上免费评估工具已经有过成百上千次的评估。SSL Labs运行的项目还包括定时扫描整个互联网的TLS配置，以及每个月定期对世界上最受欢迎的、启用了TLS的150 000个网站进行扫描的SSL Pulse（https://www.trustworthyinternet.org/ssl-pulse/）。

A.10　关于 Qualys

Qualys公司（https://www.qualys.com/，纳斯达克股票代码：QLYS）是云安全和合规解决方案的先锋和领导者，在100个国家有超过6700位客户，包括福布斯全球100强和财富100强中的大多数。QualysGuard云平台和集成解决方案帮助企业简化安全操作，并通过按需提供关键安全智能，将全方位审计、合规、保护IT系统和Web应用进行自动化来降低合规成本。Qualys于1999年创立，与行业内领先的管理服务提供商以及咨询公司（如BT、Dell SecureWorks、Fujitsu、IBM、NTT、Symantec、Verizon和Wipro等）建立了良好的合作关系。该公司同时还是Council on CyberSecurity（http://www.counciloncybersecurity.org/）和Cloud Security Alliance（CSA，https://www.cloudsecurityalliance.org/）的创始成员之一。

附录 B 改　动

附录B会长时间跟踪《OpenSSL攻略》的演变。如果你想快速了解所有的变化，可以在这里找到你想知道的所有内容。

B.1　v1.0（2013 年 5 月）

第一个发行版。

B.2　v1.1（2013 年 10 月）

这个版本中包含以下更改。

- 将“SSL/TLS部署最佳实践”的版本更新为v1.3。此版本有以下几个显著更改：(1) 废弃RC4；(2) 认为BEAST攻击在服务器端得到缓解；(3) 将向前保密放到了它自己的类别中。整体上看还有许多其他较小的改进。
- 重新编写了密码套件配置的例子，将更多的关注点放在向前保密上，使之更具相关性。
- 讨论了全部三种密钥类型（RSA、DSA和ECDSA），解释了每种密钥的正确使用场景。加入了新的内容，解释如何生成DSA和ECDSA密钥。
- 标记出了OpenSSL 1.x分支引入的密码套件配置关键字。

感谢Michael Reschly、Brian Howson、Christian Folini、Karsten Weiss和Martin Carpenter给予的反馈意见。

B.3　v2.0（2015 年 3 月）

这个版本中包含以下更改。

- 加入了第2章，这是从《HTTPS权威指南》中摘出来的。这一章主要关注安全服务器评估。
- 增加了1.3.1节中的“推荐配置”部分，包括推荐的密码套件列表。现在配置OpenSSL的时候，我倾向于显式列出所有我想使用的密码套件。
- 增加了1.4节，包括了创建和部署私有CA所需要的每一步指导。

- 将“SSL/TLS部署最佳实践”的版本更新为v1.4。这次改版最重要的变化是废弃SHA1以及SSL 3的问题（POODLE）。

感谢Stephen N. Henson、Jeff Kayser和Olivier Levillain给予的反馈意见。