OpenSSL 参考手册

这个 OpenSSL 快速参考备忘单展示了它的常用命令使用清单

入门

检查版本

```
$ openssl version -a
```

它在使用四个 CPU 内核并测试 RSA 算法的系统上运行速度有多快

```
$ openssl speed -multi 4 rsa
```

获得基本帮助

```
$ openssl help
```

生成 20 个随机字节并将它们显示在屏幕上

```
$ openssl rand -hex 20
```

基础

使用 Base64 编码文件

```
$ openssl base64 -in file.data
```

使用 Base64 编码一些文本

```
$ echo -n "some text" | openssl
```

Base64 解码一个文件并输出到另一个文件

```
$ openssl base64 -d -in encoded
.data -out decoded.data
```

编码/解码

列出可用的摘要算法

```
$ openssl list -digest-algorith
```

使用 SHA256 散列文件

```
$ openssl dgst -sha256 file.dat
```

使用 SHA256 散列文件及其二进制形式的输出（无输出十六进制编码）没有 ASCII 或编码字符将打印到控制台，只有纯字节。 您可以附加 '| xxd'

```
$ openssl dgst -binary -sha256
```

使用 SHA3-512 的哈希文本

```
$ echo -n "some text" | openssl
dgst -sha3-512
```

创建 HMAC - 使用特定密钥（以字节为单位）的文件的 SHA384

```
$ openssl dgst -SHA384 -mac
HMAC -macopt hexkey:369bd7d655
file.data
```

创建 HMAC - 一些文本的 SHA512

```
$ echo -n "some text" | openssl
dgst -mac HMAC -macopt
hexkey:369bd7d655 -sha512
```

使用哈希

列出可用的椭圆曲线

```
$ openssl ecparam -list_curves
```

创建 4096 位 RSA 公私密钥对

```
$ openssl genrsa -out pub_priv.key 4096
```

显示详细的私钥信息

```
$ openssl rsa -text -in pub_priv.key -noout
```

使用 AES-256 算法加密公私钥对

```
$ openssl rsa -in pub_priv.key -out encrypted.key -aes256
```

删除密钥文件加密并将它们保存到另一个文件

```
$ openssl rsa -in encrypted.key -out cleartext.key
```

将公私钥对文件的公钥复制到另一个文件中

列出所有支持的对称加密密码

```
$ openssl enc -list
```

使用提供的 ASCII 编码密码和 AES-128-ECB 算法加密文件

```
$ openssl rsa -in pub_priv.key -pubout -out pubkey.key
```

使用 RSA 公钥加密文件

```
$ openssl rsautl -encrypt -inkey pubkey.key -pubin -in cleartext.fil
```

使用 RSA 私钥解密文件

```
$ openssl rsautl -decrypt -inkey pub_priv.key -in ciphertext.file -c
```

使用 P-224 椭圆曲线创建私钥

```
$ openssl ecparam -name secp224k1 -genkey -out ecpriv.key
```

使用 3DES 算法加密私钥

```
$ openssl ec -in ecP384priv.key -des3 -out ecP384priv_enc.key
```

非对称加密

```
$ openssl enc -aes-128-ecb -in
cleartext.file -out
ciphertext.file -pass
pass:thisisthepassword
```

使用 AES-256-CBC 和密钥文件解密文件

```
$ openssl enc -d -aes-256-cbc -
in ciphertext.file -out
cleartext.file -pass
file:./key.file
```

使用以十六进制数字形式提供的特定加密密钥 (K) 加密文件

```
$ openssl enc -aes-128-ecb -in
cleartext.file -out
ciphertext.file -K
1881807b2d1b3d22f14e9ec52563d98
1 -nosalt
```

使用指定的加密密钥（K：256 位）和初始化向量（iv：128 位）在 CBC 块密码模式下使用 ARIA 256 加密文件

```
$ openssl enc -aria-256-cbc -in
cleartext.file -out
ciphertext.file -K
f92d2e986b7a2a01683b4c40d0cbcf6
feaa669ef2bb5ec3a25ce85d9548291
c1 -iv
470bc29762496046882b61ecee68e07
c -nosalt
```

使用提供的密钥和 iv 在 COUNTER 块密码模式下使用 Camellia 192 算法加密文件

```
$ openssl enc -camellia-192-ctr
-in cleartext.file -out
ciphertext.file -K
6c7a1b3487d28d3bf444186d7c529b4
8d67dd6206c7a1b34 -iv
470bc29762496046882b61ecee68e07
c
```

对称加密

为私钥生成 DSA 参数。 2048 位长度

```
$ openssl dsaparam -out dsaparam.pem 2048
```

生成用于签署文档的 DSA 公私密钥并使用 AES128 算法对其进行保护

```
$ openssl gendsa -out dsaprivatekey.pem -aes-128-cbc dsaparam.pem
```

将DSA公私钥文件的公钥复制到另一个文件中

```
$ openssl dsa -in dsaprivatekey.pem -pubout -out dsapublickey.pem
```

打印出 DSA 密钥对文件的内容

```
$ openssl dsa -in dsaprivatekey.pem -text -noout
```

生成 CSR 文件和 4096 位 RSA 密钥对

```
$ openssl req -newkey rsa:4096
-keyout private.key -out
request.csr
```

显示证书签名请求 (CSR) 内容

```
$ openssl req -text -noout -in
```

显示 CSR 文件中包含的公钥

```
$ openssl req -pubkey -noout -i
n request.csr
```

使用 RSA 私钥对文件的 sha-256 哈希进行签名

```
$ openssl dgst -sha256 -sign rsakey.key -out signature.data documen
t.pdf
```

使用公钥验证 SHA-256 文件签名

```
$ openssl dgst -sha256 -verify publickey.pem -signature signature.da
```

使用 DSA 私钥对文件的 sha3-512 哈希进行签名

```
$ openssl pkeyutl -sign -pkeyopt digest:sha3-512 -in document.docx
-inkey dsaprivatekey.pem -out signature.data
```

验证 DSA 签名

```
$ openssl pkeyutl -verify -sigfile dsasignature.data -inkey dsakey.p
```

使用 P-384 椭圆曲线创建私钥

```
$ openssl ecparam -name secp384r1 -genkey -out ecP384priv.key
```

使用3DES算法加密私钥

```
$ openssl ec -in ecP384priv.key -des3 -out ecP384priv_enc.key
```

使用带有生成密钥的椭圆曲线对 PDF 文件进行签名

```
$ openssl pkeyutl -sign -inkey ecP384priv_enc.key -pkeyopt
digest:sha3-512 -in document.pdf -out signature.data
```

验证文件的签名。 如果没问题，您必须收到“签名验证成功”

```
$ openssl pkeyutl -verify -in document.pdf -sigfile signature.data -
```

数字签名

列出所有支持的密码套件

```
$ openssl ciphers -V 'ALL'
```

列出 AES 支持的所有密码套件

```
$ openssl ciphers -V 'AES'
```

列出所有支持 CAMELLIA 和 SHA256 算法的密码套件。

```
$ openssl ciphers -V 'CAMELLIA+SHA256'
```

使用端口 443 (HTTPS) 与服务器的 TLS 连接

```
$ openssl s_client -connect domain.com:443
```

使用 v1.2 与服务器的 TLS 连接

```
$ openssl s_client -tls1_2 -connect domain.com:443
```

TLS 连接和禁用 v1.0

使用现有私钥创建证书签名请求（CSR）。 当您需要在不更改私钥的情况下更新公共数字证书时，这会很有用

```
$ openssl req -new -key
private.key -out request.csr
```

创建 EC P384 曲线参数文件以在下一步中使用椭圆曲线生成 CSR

```
$ openssl genpkey -genparam -al
gorithm EC -out EC_params.pem -
pkeyopt
ec_paramgen_curve:secp384r1 -pk
eyopt ec_param_enc:named_curve
```

使用在上一步中创建的椭圆曲线 P384 参数文件创建 CSR 文件。 而不是使用 RSA 密钥。

```
$ openssl req -newkey
ec:EC_params.pem -keyout
EC_P384_priv.key -out
EC_request.csr
```

创建自签名证书，新的 2048 位 RSA 密钥对，有效期为一年

```
$ openssl req -newkey rsa:2048
-nodes -keyout priv.key -x509 -
days 365 -out cert.crt
```

使用 CSR 文件和用于签名的私钥创建并签署新证书（您必须准备好 openssl.cnf 文件）

```
$ openssl ca -in request.csr -o
ut certificate.crt -config
./CA/config/openssl.cnf
```

显示PEM格式证书信息

```
$ openssl x509 -text -noout -in
```

以 Abstract Sintax Notation One (ASN.1) 显示证书信息

```
$ openssl asn1parse -in cert.cr
```

提取证书的公钥

```
$ openssl x509 -pubkey -noout -
```

在证书中提取公钥的模数

```
$ openssl x509 -modulus -noout
```

从 HTTPS/TLS 连接中提取域证书

```
$ openssl s_client -connect
domain.com:443 | openssl x509 -
```

```
$ openssl s_client -no_tls1 domain.com:443
```

使用特定密码套件的 TLS 连接

```
$ openssl s_client -cipher DHE-RSA-AES256-GCM-SHA384 domain.com:443
```

显示服务器提供的所有证书的 TLS 连接

```
$ openssl s_client -showcerts domain.com:443
```

使用证书、私钥和仅支持 TLS 1.2 设置监听端口以接收 TLS 连接

```
$ openssl s_server -port 443 -cert cert.crt -key priv.key -tls1_2
```

从 HTTPS/TLS 连接中提取域证书

```
$ openssl s_client -connect domain.com:443 | openssl x509 -out certi
```

nmap 命令：通过 HTTPS/TLS 连接显示启用的密码套件

```
$ nmap --script ssl-enum-ciphers -p 443 domain.com
```

nmap 命令：使用 SNI 通过 TLS (HTTPS) 连接显示启用的密码套件。（将其更改为所需的 IP 和域名）

```
$ nmap --script ssl-enum-ciphers --script-args=tls.servername=domair
```



```
out certificate.crt
```

将证书从 PEM 格式转换为 DER 格式

```
$ openssl x509 -inform PEM -out
form DER -in cert.crt -out
cert.der
```

检查证书公钥是否与私钥和请求文件匹配。每个文件一步。必须在输出哈希中匹配

```
$ openssl x509 -modulus -in
certificate.crt -noout |
openssl dgst -sha256
$ openssl rsa -modulus -in
private.key -noout | openssl dg
st -sha256
$ openssl req -modulus -in
request.csr -noout | openssl dg
st -sha256
```

将证书从 PEM (base64) 格式转换为 DER（二进制）格式

```
$ openssl x509 -in certificate.
pem -outform DER -out certif-
icate.der
```

将证书和私钥插入 PKCS #12 格式文件。这些文件可以导入到 Windows 证书管理器或 Java Key Store (jks) 文件中

```
$ openssl pkcs12 -export -out
cert_key.p12 -inkey private.key
-in certificate.crt
```

显示 PKCS #12 文件的内容

```
$ openssl pkcs12 -in cert_key.p
```

将 .p12 文件转换为 Java Key Store。此命令使用 java keytool 而不是 openssl。

```
keytool -importkeystore -destke
ystore javakeystore.jks -srckey
store cert_key.p12 -srcstoretyp
e pkcs12
```

将 PEM 证书转换为 PKCS #7 格式

```
$ openssl crl2pkcs7 -nocrl -cer
tfile certificate.crt -out
cert.p7b
```

将 PKCS #7 文件从 PEM 转换为 DER

```
$ openssl pkcs7 -in cert.p7b -o
utform DER -out p7.der
```

查看

使用具有证书扩展名的命令将 cert.xxx 替换为证书名称

```
$ openssl x509 -in cert.pem -te
$ openssl x509 -in cert.cer -te
$ openssl x509 -in cert.crt -te
```

如果您收到以下错误，则表示您正在尝试查看 DER 编码的证书，并且需要使用下面“查看 DER 编码的证书”部分中的命令：

```
unable to load certificate
12626:error:0906D06C:PEM routin
es:PEM_read_bio:no start line:p
em_lib.c:647:Expecting: TRUSTED
CERTIFICATE View DER encoded Ce
rtificate
```

查看 PEM 编码证书

```
openssl x509 -in certificate.der -inform der -text -noout
```

如果您收到以下错误，则表示您正在尝试使用用于 DER 编码证书的命令查看 PEM 编码证书。 使用上面“查看 PEM 编码证书”部分中的命令：

```
unable to load certificate
13978:error:0D0680A8:asn1 encoding routines:ASN1_CHECK_TLEN:wrong t
ag:tasn_dec.c:1306:
13978:error:0D07803A:asn1 encoding routines:ASN1_ITEM_EX_D2I:nested
asn1 error:tasn_dec.c:380:Type=X509
```

查看 DER 编码证书

```
# subject + issuer
openssl crl2pkcs7 -nocrl -certfile host.domain.tld-ca-chain.pem | openssl pkcs7 -print_certs -noout
# full public keys
openssl crl2pkcs7 -nocrl -certfile host.domain.tld-ca-chain.pem | openssl pkcs7 -print_certs -text -noou
```

查看证书链中的所有证书

转换

将 DER 文件 (.crt .cer .der) 转换为 PEM

```
openssl x509 -inform der -in
certificate.cer -out
certificate.pem
```

将 PEM 文件转换为 DER

```
openssl x509 -outform der -in
certificate.pem -out
certificate.der
```

将包含私钥和证书的 PKCS#12 文件 (.pfx .p12) 转换为 PEM

```
openssl pkcs12 -in keyStore.pfx
-out keyStore.pem -nodes
# 您可以添加 -nocerts 以仅输出私钥
或添加 -nokeys 以仅输出证书
```

将 PEM 证书文件和私钥转换为 PKCS#12 (.pfx .p12)

```
openssl pkcs12 -export -out
certificate.pfx -inkey
privateKey.key -in
certificate.crt -certfile
```

将 PEM 转换为 DER

```
$ openssl x509 -outform der -in
certificate.pem -out
certificate.der
```

将 PEM 转换为 P7B

```
$ openssl crl2pkcs7 -nocrl -cer
tfile certificate.cer -out
certificate.p7b -certfile
CACert.cer
```

将 PEM 转换为 PFX

```
$ openssl pkcs12 -export -out
certificate.pfx -inkey
privateKey.key -in
certificate.crt -certfile
CACert.crt
```

OpenSSL 转换 PEM

将 P7B 转换为 PEM

```
$ openssl pkcs7 -print_certs -i
```

将 DER 转换为 PEM

```
$ openssl x509 -inform der -in
certificate.cer -out
certificate.pem
```

OpenSSL 转换 DER

将 PFX 转换为 PEM

```
$ openssl pkcs12 -in
certificate.pfx -out
certificate.cer -nodes
```

OpenSSL 转换 PFX

在命令行上使用 OpenSSL 您首先需要生成公钥和私钥。 您应该使用 -passout 参数对这个文件进行密码保护，这个参数可以采用许多不同的形式，因此请查阅

```
certificate.crt -certfile
CACert.crt
```

将 PEM 转换为 CRT（.CRT 文件）

```
openssl x509 -outform der -in
certificate.pem -out
certificate.crt
```

转换示例

```
n certificate.p7b -out
certificate.cer
```

将 P7B 转换成 PFX

```
$ openssl pkcs7 -print_certs -i
n certificate.p7b -out
certificate.cer
$ openssl pkcs12 -export -in
certificate.cer -inkey
privateKey.key -out
certificate.pfx -certfile
CACert.cer
```

OpenSSL 转换 P7B

NAME

```
$ rsa - RSA key processing tool
```

SYNOPSIS 概要

```
$ openssl rsa [-help] [-inform PEM|NET|DER] [-outform PEM|NET|DER]
[-in filename] [-passin arg] [-out filename] [-passout arg] [-aes12
8] [-aes192] [-aes256] [-camellia128] [-camellia192] [-camellia256]
[-des] [-des3] [-idea] [-text] [-noout] [-modulus] [-check] [-pubin
] [-pubout] [-RSAPublicKey_in] [-RSAPublicKey_out] [-engine id]
```

DESCRIPTION 描述

rsa 命令处理 RSA 密钥。 它们可以在各种形式之间转换，并且可以打印出它们的组成部分
请注意，此命令使用传统的 SSLeay 兼容格式进行私钥加密：较新的应用程序
应该使用 pkcs8 实用程序使用更安全的 PKCS#8 格式。

COMMAND OPTIONS 命令选项

```
-help
#> 打印出使用信息。

-inform DER|NET|PEM
#> 这指定了输入格式。 DER 选项使用与 PKCS #1 RSAPrivateKey 或 SubjectPub
licKeyInfo 格式兼容的 ASN1 DER 编码形式。 PEM 形式是默认格式：它由 DER 格式
base64 编码，并带有额外的页眉和页脚行。 输入 PKCS#8 格式的私钥也 接受。 NET
形式是一种在注释部分中描述的格式。

-outform DER|NET|PEM
#> 这指定了输出格式，选项与 -inform 选项具有相同的含义。

-in filename
#> 如果未指定此选项，这将指定要从中读取密钥的输入文件名或标准输入。 如果密钥被加
密，将提示输入密码。

-passin arg
#> 输入文件密码源。有关 arg 格式的更多信息，请参阅 openssl 中的 PASS PHRASE
ARGUMENTS 部分。

-out filename
#> 如果未指定此选项，这将指定要写入密钥的输出文件名或标准输出。如果设置了任何加密
选项，则会提示输入密码。输出文件名不应与输入文件名相同。

-passout password
#> 输出文件密码源。有关 arg 格式的更多信息，请参阅 openssl 中的 PASS PHRASE
```

OpenSSL 文档

```
$ openssl genrsa -out private.p
```

这将创建一个名为 private.pem 的密钥文件，它使用 4096 位。 这个文件实际上有私钥和公钥，所以你应该从这个文件中提取公钥：

```
$ openssl rsa -in private.pem -
out public.pem -outform PEM -pu
bout
# or
$ openssl rsa -in private.pem -
pubout > public.pem
# or
$ openssl rsa -in private.pem -
pubout -out public.pem
```

您现在将拥有仅包含您的公钥的 public.pem，您可以与第 3 方自由共享。 您可以通过使用您的公钥自己加密一些东西然后使用您的私钥解密来测试这一切，首先我们需要一些数据来加密：

示例文件：

```
$ echo 'too many secrets' > fil
```

您现在在 file.txt 中有一些数据，让我们使用 OpenSSL 和公钥对其进行加密：

```
$ openssl rsautl -encrypt -inke
y public.pem -pubin -in
file.txt -out file.ssl
```

这会创建一个 file.txt 的加密版本，称为 file.ssl，如果你看这个文件，它只是二进制垃圾，对任何人都没有什么用处。 现在您可以使用私钥对其进行解密：

```
$ openssl rsautl -decrypt -inke
y private.pem -in file.ssl -out
decrypted.txt
```

您现在将在 decrypted.txt 中有一个未加密的文件：

```
cat decrypted.txt
 |output -> too many secrets
```

通过 OpenSSL 生成 rsa 密钥

-aes128	-aes192	-aes256
-des3	-des	

支持以下加密算法

ARGUMENTS 部分。

-aes128|-aes192|-aes256|-camellia128|-camellia192|-camellia256|-des|-des3|-idea
#> 这些选项在输出之前使用指定的密码加密私钥。提示输入密码。如果未指定这些选项，则密钥将以纯文本形式写入。这意味着使用 rsa 实用程序读取没有加密选项的加密密钥可用于从密钥中删除密码短语，或者通过设置可用于添加或更改密码短语的加密选项。这些选项只能用于 PEM 格式的输出文件。

-text
#> 除了编码版本之外，还以纯文本形式打印出各种公钥或私钥组件。

-noout
#> 此选项可防止输出密钥的编码版本。

-modulus
#> 此选项打印出密钥模数的值。

-check
#> 此选项检查 RSA 私钥的一致性。

-pubin
#> 默认情况下，从输入文件中读取私钥：使用此选项，改为读取公钥。

-pubout
#> 默认情况下输出私钥：使用此选项将输出公钥。 如果输入是公钥，则会自动设置此选项。

-RSAPublicKey_in, -RSAPublicKey_out
#> 类似于 -pubin 和 -pubout，除了使用 RSAPublicKey 格式。

-engine id
#> 指定引擎（通过其唯一 ID 字符串）将导致 rsa 尝试获取对指定引擎的功能引用，从而在需要时对其进行初始化。 然后引擎将被设置为所有可用算法的默认值

OpenSSL 中的 RSA 工具选项

要删除 RSA 私钥上的密码短语：

```
$ openssl rsa -in key.pem -out
```

要使用三重 DES 加密私钥：

```
$ openssl rsa -in key.pem -des3
-out keyout.pem
```

要将私钥从 PEM 格式转换为 DER 格式：

```
$ openssl rsa -in key.pem -outf
orm DER -out keyout.der
```

将私钥的组件打印到标准输出：

```
$ openssl rsa -in key.pem -text
-noout
```

仅输出私钥的公共部分：

```
$ openssl rsa -in key.pem -pubo
ut -out pubkey.pem
```

以 RSAPublicKey 格式输出私钥的公共部分：

```
$ openssl rsa -in key.pem -RSAP
ublicKey_out -out pubkey.pem
```

示例

格式

```
-----BEGIN RSA PUBLIC KEY-----
-----END RSA PUBLIC KEY-----
```

RSA 公钥

```
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
-----END RSA PRIVATE KEY-----
```

加密的 PEM 私钥

```
-----BEGIN X509 CRL-----
-----END X509 CRL-----
```

CRL

```
-----BEGIN CERTIFICATE-----
-----END CERTIFICATE-----
```

CRT

```
-----BEGIN CERTIFICATE REQUEST-
-----END CERTIFICATE REQUEST---
```

CSR

```
-----BEGIN NEW CERTIFICATE REQUI
-----END NEW CERTIFICATE REQUES
```

NEW CSR

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

PEM

```
-----BEGIN PKCS7-----
-----END PKCS7-----
```

PKCS7

```
-----BEGIN PRIVATE KEY-----
```

```
-----BEGIN DSA PRIVATE KEY-----
```

为了让 OpenSSL 将其识别为 PEM 格式，它必须使用 Base64 进行编码，并带有以下标头：

```
-----BEGIN CERTIFICATE-----
and footer :
-----END CERTIFICATE-----
```

此外，每行的长度不得超过 79 个字符。否则你会收到错误：

```
2675996:error:0906D064:PEM rout
ines:PEM_read_bio:bad base64 de
code:pem_lib.c:818:
```

注意：PEM 标准 (RFC1421) 要求行长度为 64 个字符。 可以使用 UNIX 命令行实用程序转换存储为单行的 PEM 证书：

```
$ fold -w 64
```

PKCS#1 RSAPublicKey (PEM header: BEGIN RSA PUBLIC KEY)

```
-----END PRIVATE KEY-----
```
私钥

```
-----END DSA PRIVATE KEY-----
```
DSA密钥

PKCS#8 EncryptedPrivateKeyInfo (PEM header: BEGIN ENCRYPTED PRIVATE KEY)

PKCS#8 PrivateKeyInfo (PEM header: BEGIN PRIVATE KEY)

X.509 SubjectPublicKeyInfo (PEM header: BEGIN PUBLIC KEY)

CSR PEM header : (PEM header:—-BEGIN NEW CERTIFICATE REQUEST—-)

DSA PrivateKeyInfo (PEM header: (—-BEGIN DSA PRIVATE KEY—-)

识别为 PEM 格式

```
-----BEGIN EC PRIVATE KEY-----
-----BEGIN EC PRIVATE KEY-----
```
椭圆曲线

```
-----BEGIN PGP PRIVATE KEY BLOCK
-----END PGP PRIVATE KEY BLOCK--
```
PGP 私钥

```
-----BEGIN PGP PUBLIC KEY BLOCK
-----END PGP PUBLIC KEY BLOCK--
```
PGP 公钥

校验

在建立 SSL/TLS 连接之前，客户端需要确保收到的证书有效。为了做到这一点，客户端不仅要验证其公钥的真实性，还要验证与之相关的其他元数据（了解这一点对于了解典型数字证书的内容很重要）：

签名验证 这确保了证书没有以任何方式被更改

证书尚未过期 当证书由 CA 颁发时，它会指定一个到期日期

证书主题与主机名匹配 证书是为特定服务器颁发的。因此，证书主题名称需要与客户端尝试连接的 URL 相匹配

它没有被撤销 有时证书可以在任何需要的情况下被其颁发者撤销（例如，关联的私钥已被公开，因此证书无效）

它由受信任的 CA 签名 为了证明证书的真实性，我们需要获取 CA 证书并验证其可信度。然而在 PKI 中有一个信任链的概念，因此 CA 证书可能是由另一个 CA 颁发的。因此我们需要获得另一个 CA 的证书并验证它。依此类推……因此，为了信任证书，我们需要一直导航到根 CA。最后，如果我们信任根 CA，可以肯定地说我们信任整个链

介绍

a) 您的整个 CA 链在一个文件中，实际的网络服务器或客户端证书在另一个文件中

```
$ openssl verify -untrusted ca-chain.pem 客户端证书.pem
```

b) 单独文件中的根证书和中间证书以及另一个文件中的实际网络服务器或客户端证书

```
$ openssl verify -CAfile root.pem -untrusted intermediate-chain.pem
client-cert.pem
```

如果您有多个中间 CA（例如 root.pem -> intermediate1.pem -> intermediate2.pem -> client-cert.pem），将它们连接到一个文件中并通过：-untrusted intermediate-chain.pem 或执行它与 cat：

```
$ openssl verify -CAfile root.pem -untrusted <(cat
intermediate1.pem intermediate2.pem) client-cert.pem
```

实例

```
$ openssl verify -CAfile letsencrypt-root-cert/isrgrootx1.pem.txt -
untrusted letsencrypt-intermediate-cert/letsencryptauthorityx3.pem.
txt /etc/letsencrypt/live/sitename.tld/cert.pem
/etc/letsencrypt/live/sitename.tld/cert.pem: OK
```

验证信任链

```
$ openssl x509 -enddate -noout -in file.pem
```

验证本地证书文件

这是我的 bash 命令行，用于按过期顺序列出多个证书，最近过期的证书最先过期。

```
for pem in /etc/ssl/certs/*.pem; do
    printf '%s: %s\n' \
      "$(date --date="$(openssl x509 -enddate -noout -in "$pem"|cut
-d= -f 2)" --iso-8601)" \
      "$pem"
done | sort
```

示例输出：

```
2015-12-16: /etc/ssl/certs/Staat_der_Nederlanden_Root_CA.pem
2016-03-22: /etc/ssl/certs/CA_Disig.pem
2016-08-14: /etc/ssl/certs/EBG_Elektronik_Sertifika_Hizmet_S.pem
```

```
curl --insecure -v
https://www.google.com 2>&1 | a
wk 'BEGIN { cert=0 } /^\* Serve
r certificate:/ { cert=1 } /^\*
/ { if (cert) print }'

* Server certificate:
*  subject: C=US; ST=California
; L=Mountain View; O=Google LLC
; CN=www.google.com
*  start date: Mar  1 09:46:35
2019 GMT
*  expire date: May 24 09:25:00
2019 GMT
*  issuer: C=US; O=Google Trust
Services; CN=Google Internet Au
thority G3
*  SSL certificate verify ok.
```

验证远程服务器

这是一个 bash 函数，它会检查你所有的服务器，假设你正在使用 DNS 循环法。 请注意，这需要 GNU 日期并且不能在 Mac OS 上运行

```
function check_certs () {
  if [ -z "$1" ]
  then
    echo "domain name missing"
    exit 1
  fi
  name="$1"
  shift

  now_epoch=$( date +%s )

  dig +noall +answer $name | while read _ _ _ _ ip;
  do
    echo -n "$ip:"
    expiry_date=$( echo | openssl s_client -showcerts -servername
$name -connect $ip:443 2>/dev/null | openssl x509 -inform pem -noou
t -enddate | cut -d "=" -f 2 )
    echo -n " $expiry_date";
    expiry_epoch=$( date -d "$expiry_date" +%s )
    expiry_days="$(( ($expiry_epoch - $now_epoch) / (3600 * 24) ))"
    echo "    $expiry_days days"
  done
}
```

输出示例：

```
$ check_certs stackoverflow.com
151.101.1.69: Aug 14 12:00:00 2019 GMT    603 days
151.101.65.69: Aug 14 12:00:00 2019 GMT    603 days
151.101.129.69: Aug 14 12:00:00 2019 GMT    603 days
151.101.193.69: Aug 14 12:00:00 2019 GMT    603 days
```

截止日期

使用私钥验证 TLS 证书

希望您永远不会遇到不知道用于生成 TLS 证书的私钥的情况，但如果您知道……这里是您可以检查的方法。

注意：这比将证书上传到生产环境以检查它们更好😉

假设我们已经生成了一个名为 example.com.key 的私钥和一个名为 example.com.crt 的证书，我们可以使用 openssl 检查 MD5 哈希值是否相同：

```
$ openssl x509 -noout -modulus -in example.com.crt | openssl md5
$ openssl rsa -noout -modulus -in example.com.key | openssl md5
```

```
* Using HTTP2, server supports
multi-use
* Connection state changed (HTT
P/2 confirmed)
* Copying HTTP/2 data in stream
buffer to connection buffer aft
er upgrade: len=0
* Using Stream ID: 1 (easy hand
le 0x7ff5dc803600)
* Connection state changed (MAX
_CONCURRENT_STREAMS updated)!
* Connection #0 to host www.goo
gle.com left intact
```

验证 curl

您需要为 curl 提供整个证书链，因为 curl 不再附带任何 CA 证书。 由于 cacert 选项只能使用一个文件，因此您需要将完整的链信息连接到 1 个文件中。 从 https://curl.haxx.se/ca/cacert.pem 获取根 CA 证书包。

```
$ curl --cacert certRepo -u
user:passwd -X GET -H 'Content-
Type: application/json' "https/
/somesecureserver.com/rest/fiel
d"
```

验证 openssl s_client

使用 SNI

如果远程服务器使用 SNI（即在一个 IP 地址上共享多个 SSL 主机），您将需要发送正确的主机名以获得正确的证书（-servername 选项用于启用 SNI 支持）。

```
$ openssl s_client -showcerts -
servername www.example.com -con
nect www.example.com:443 </dev/
null
```

没有 SNI

如果远程服务器没有使用 SNI，那么你可以跳过 -servername 参数：

```
openssl s_client -showcerts -co
nnect www.example.com:443 </dev
/null
```

要查看站点证书的完整详细信息，您也可以使用以下命令链：

```
$ echo | \
    openssl s_client -servernam
e www.example.com -connect
www.example.com:443 2>/dev/null
| \
    openssl x509 -text
```

对于带有 starttls 的 SMTP，请使用：

```
$ openssl s_client -connect
server:port -starttls smtp
```

对于 Client Auth 保护的资源，请使用：

为了让事情变得更好，你可以写一个脚本：

```
#!/bin/bash
CERT_MD5=$(openssl x509 -noout -modulus -in example.com.crt | openss
 KEY_MD5=$(openssl rsa  -noout -modulus -in example.com.key | openss

if [ "$CERT_MD5" == "$KEY_MD5" ]; then
  echo "Private key matches certificate"
else
  echo "Private key does not match certificate"
fi
```

```
$ openssl s_client -connect
host:port -key
our_private_key.pem -showcerts
\
    -cert our_server-signed_cer
t.pem
```

-prexit 也会返回数据：

```
$ openssl s_client -connect
host:port -prexit
```

Java Key store

Java 密钥库

```
$ keytool -importcert -file certificate.cer -keystore keystore.jks -alias "Alias"
$ ..\..\bin\keytool -import -trustcacerts -keystore cacerts -storepass changeit -noprompt -alias yourAli
$ keytool -import -alias joe -file mycert.cer -keystore mycerts -storepass changeit
```

创建

使用 certstrap 创建开发证书

```
$ brew install certstrap
$ certstrap init --common-name "ExampleDevCA" --expires "10 years" -o "My Tech Inc." -c "DE" -l "Muench
en" --st "Bayern" --stdout
$ certstrap request-cert --common-name "example.localhost" -o "My Tech Inc." -c "DE" -l "Muenchen" --st
"Bayern" --stdout --domain "*.example.localhost","example.localhost","localhost"
$ certstrap sign "example.localhost" --CA ExampleDevCA
```

使用 mkcert 创建开发证书

```
$ brew install mkcert
$ mkcert "*.example.localhost"

# Clean up with:
$ rm -vrf "$HOME/Library/Application Support/mkcert"
_wildcard.example*
```

另见

- OpenSSL 官网 *(openssl.org)*
- OpenSSL Cheat Sheet *(cheatography.com)*
- OpenSSL Cheat Sheet *(megamorf.gitlab.io)*