Oracle Linux

Managing Certificates and Public Key Infrastructure

F24286-14
November 2023

Oracle Linux Managing Certificates and Public Key Infrastructure,

F24286-14

Contents

Preface

1 About Public Key Infrastructure

2 Setting Up TLS/SSL with OpenSSL

3 Setting Up TLS/SSL with Other Tools

4 Managing System Certificates

Preface

Oracle Linux: Managing Certificates and Public Key Infrastructure describes features in Oracle Linux to manage certificates and public key infrastructure.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at https://www.oracle.com/corporate/accessibility/.

For information about the accessibility of the Oracle Help Center, see the Oracle Accessibility Conformance Report at https://www.oracle.com/corporate/accessibility/templates/t2-11535.html.

Access to Oracle Support for Accessibility

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit https://www.oracle.com/corporate/accessibility/learning-support.html#support-tab.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

1

About Public Key Infrastructure

This chapter provides a brief overview of the public key cryptography and how it works, including information about the public key infrastructure, which is used for the general management of keys on Oracle Linux.

What is Public Key Cryptography?

Public key cryptography is an encryption technique that's used to enable secure communications on an insecure public network and also to verify the identity of the entity on the other end of a network connection. Public key cryptography works by establishing an asymmetric pair keys. Data encrypted by one key is decrypted by the other key. One key is kept *private* and the other key is made *public*. Someone decrypting the data using the public key can be sure that the data was encrypted by someone who has access to the private key. Similarly, someone encrypting data using the public key can be sure that the data can only be decrypted by someone who has access to the private key.

Neither key on its own can establish the identity of the sender of the data. To achieve this, the public key is typically signed as belonging to the owner of the private key. The signing process is done by a trusted third party, known as a Certification Authority (CA). The creator of the private and public key pair sends the public key to the CA in the form of a Certificate Signing Request (CSR). The CA uses its own private key to sign a certificate, which contains an encrypted version of the originator's public key, together with other information about the entity (*subject*), the CA (*issuer*), the period of validity of the certificate, and the cryptographic algorithms used. This certificate can be made public or provided to any client that need to decrypt data that has been encrypted using the private key.

Clients that trust the CA can also trust the public key stored in the certificate. Decrypting the certificate with the CA certificate yields the public key that can then be used to create a secure communication channel that keeps the data confidential and which can be used to establish the identity of the originator of data moving through the channel.

For the Internet, many public top-level or *root* CAs and intermediary CAs exist that are trusted by a root CA to issue certificates on behalf of entities. An intermediary CA returns a certificate chain, where each certificate in the chain authenticates the public key of the signer of the previous certificate in the chain up to and including a root CA.

CA certificates are only used to establish the identity of a public key and the period for which the public key is considered valid. When the certificate expires, data encrypted using the public key can still be decrypted by the private key. This means that the private key must be kept safe forever for communications to always be considered secure. A mechanism also exists within public key cryptography that can be used to help mitigate against private key compromises. This mechanism is known as Perfect Forward Secrecy (PFS) and uses a key exchange algorithm to securely agree on a random and disposable session key that can be used with a symmetric cipher to encrypt data. The advantage of this approach is that if the session key is compromised, only the communications in that particular communication session are exposed. Equally, if the private key is compromised, all the actual communication sessions aren't automatically exposed either.

Another added benefit of PFS is that it simplifies the computationally expensive and slow process of decrypting and validating each piece of information using the asymmetric key pair and the CA certificate. In reality, the process of decrypting the public key and validating it against the CA certificate and then using it to decrypt data within a communication session is only done at the beginning of the session, until PFS is established. The algorithm to create and share the random session key is typically the Diffie-Hellman key exchange. The session key then uses a symmetric cipher to perform more rapid encryption and decryption of data through the rest of the session. The cipher most commonly used for this purpose is AES, which can take advantage of hardware to make encryption and communication in ciphertext almost as fast as communicating with plaintext.

The handling of the communication channel and the negotiation where the client and server side switch from asymmetric to symmetric cryptography are all achieved using the Transport Layer Security (TLS) or Secure Sockets Layer (SSL) cryptographic protocols.

OpenSSL, GnuTLS, and Network Security Services (NSS) provide open source implementations of the TLS and SSL protocols. You can also use the keytool command provided with OpenJDK package to manage Java Keystores, often used by Java-based applications. If a hierarchy of trust is confined to the organization's intranet, you can use these implementations to generate a root certificate and set up a CA for that domain. However, unless you install this self-signed root certificate on each system in the organization, browsers, LDAP or IPA authentication, and other software that use certificates would prompt the user about the potential untrusted relationship.

Note:

If you do use certificates for a domain that are validated by a root or intermediary-level CA, you don't need to distribute a root certificate, as the appropriate certificate is already present on each system.

Typically, TLS/SSL certificates expire after one year. Other certificates, including root certificates that are distributed with web browsers and which are issued by root and intermediary CAs, expire after a period of five to 10 years. To avoid having applications display warnings about out-of-date certificates, plan to replace TLS/SSL certificates before they expire. For root certificates, you would typically update the software before the certificate expires.

If you request a signed certificate from a CA for which a root certificate or certificate chain that authenticates the CA's public key doesn't already exist on the system, obtain a trusted root certificate from the CA. To avoid a potential man-in-the-middle attack, verify the authenticity of the root certificate before importing it. Check that the certificate's fingerprint matches the fingerprint that's published by the CA.

About SSL and TLS

Both Secure Sockets Layer (SSL) and Transport Layer Security (TLS) are communications protocols that ensure secure connections and exchanges between server and client systems. Both protocols provide encryption and authentication to secure network communications. However, SSL is an older technology and has been replaced by TLS. The cryptography used by TLS is more complex, advanced, robust, and secure. Authentication with TLS is faster and alert messaging is improved.

Despite this change to the underlying protocol, the OpenSSL project retains its name and the *SSL* terminology is often used interchangeably to describe TLS functionality. In the context of secure communications, SSL is now understood as referring to the TLS protocol and TLS certificates. Any references to SSL in this documentation are intended to be understood in the context of TLS.

Automatic Certificate Management Environment (ACME)

Automatic Certificate Management Environment (ACME) is a protocol and framework that's published by the IETF in RFC 8555 and which can be used for the signing and creation of certificates where domain validation is required.

The protocol uses JSON formatted messages over HTTPS with a CA to handle validation of domain ownership automatically by having the ACME client perform an action that can only be done with control of the domain name. For example, the CA could either request the provision of a DNS record, or could request a specific HTTP resource to be made available on a web server at the domain name.

After the CA validates that the entity requesting a certificate has ownership of the domain, the CA can sign the certificate that's sent to it by the ACME client. Typically, the client can automatically install the certificate at a location that's usable by services running on the system.

ACME lowers the cost and complexity associated with managing public key infrastructure. Sometimes, obtaining signed certificates for systems within domains can be free, depending on the selection of CA. For example, Let's Encrypt, the originator of the ACME protocol, provides a free and open CA service. Other commercial CAs are starting to also offer free ACME based certificates as well.

While the first version of the ACME protocol could be used to create only single domain certificates, ACME v2 can be used for the creation and signing of certificates with wildcard domains, such as *.example.com. Therefore, you can use a single certificate across all subdomains. Note that ACME only validates domains. If you need certificates that require more validations, you might need signed certificates from an established CA that offers services beyond ACME.

If you need to create and issue certificates across an infrastructure to use TLS/SSL protected services, consider using a CA that supports ACME and using an ACME client. ACME can automatically generate the key pairs and CSR, submit the CSR to a CA for validation, perform any validation steps for the CA, and obtain the signed certificate and store it somewhere that's accessible to services and applications. Many clients automatically set periodic `cron` tasks to check for certificate expiry and to automatically request a new certificate before the current certificate expires.

2

Setting Up TLS/SSL with OpenSSL

This chapter describes the OpenSSL tools that are available in Oracle Linux and how to use them to create Certificate Signing Requests (CSRs), self-signed certificates, and privately owned CA certificates. Also covered in this chapter are instructions on how to use the OpenSSL tools to validate and test certificates that are configured for a protocol to confirm that services are configured correctly.

Features of the openssl Command

With the `openssl` command, which is included in the `openssl` package, you can perform a wide range of cryptography functions from the OpenSSL library, including the following:

- Create and managing pairs of private and public keys.
- Perform public key cryptographic operations.
- Create self-signed certificates.
- Create certificate signing requests (CSRs).
- Create certificate revocation lists (CRLs).
- Convert certificate files between various formats.
- Calculate message digests.
- Encrypt and decrypt files.
- Test client-side and server-side TLS/SSL with HTTP and SMTP servers.
- Verify, encrypt, and sign S/MIME email.
- Generate and test prime numbers and generate pseudo random data.

Creating Key Pairs

As a first step to use any form of public key cryptography, create a public/private key pair. You can then use the private key to create a Certificate Signing Request (CSR) that contains the associated a public key. The CSR can be used to obtain a signed certificate from a CA. Typically, the steps to create a key pair and a CSR or a self-signed certificate, are performed as a single-step operation when using OpenSSL to generate these files.

In the following instructions and example, the creation of a key pair is treated as an atomic operation so that the process can be described and elements can be called out for better understanding. Usually, this step is incorporated into other commands for efficiency.

The following are the main elements that you need to consider when creating a key pair:

- **Algorithm**

  OpenSSL provides the use of RSA and ECDSA key algorithms, with RSA keys being the most widely used. While DSA keys can be created, these must not be used unless required. ECDSA is a modern variant that provides much smaller and efficient key sizes than both RSA or DSA, along with corresponding security. ECDSA might be a good choice for performance. However, be aware that some environments might not recognize ECDSA keys.

- **Key Size**

  The key size checks the complexity of the key for the algorithm, which is specified in bits. Bigger-sized keys are more secure because they're more complex and harder to decipher. Bigger-sized keys also come with a performance hit, because each decryption bit requires more memory and processing to complete. Therefore, selecting a key size is a balance between security and performance. Key sizes are complex, in that they relate to the algorithms and ciphers that are being used. In general, when creating RSA keys, a key size is 2048 bits, while ECDSA keys provide similar security using a key size of 256 bits.

- **Passphrase**

  When creating a key that's encrypted and protected with a cipher, you're prompted for a passphrase that can be used to validate that you can use the key. Encrypting a key with a passphrase is optional but recommended. Using a passphrase with a key can be problematic when TLS is enabled for a system service, as the service can't be automatically restarted without user intervention. Often, where certificates are issued for services; for convenience, they're created without passphrases. If a private key is created without a passphrase, be aware that anyone who gains access to the private key file can emulate services to perform man-in-the-middle type snooping. When a key is protected with a passphrase, you can select a cipher algorithm to use to encrypt the contents of the private key. Many ciphers are available for this purpose. To obtain a complete list of ciphers, use the `openssl list-cipher-commands` command. The AES cipher is commonly used for this purpose and is typically specified with a key size of 128 or 256 (`aes128` or `aes256`).

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

```
sudo openssl genrsa -out private.key 2048

Generating RSA private key, 2048 bit long modulus
...................................................
....................................................
...................................................+++
................................+++
e is 65537 (0x10001)
```

This command generates an unencrypted key in the local directory, named *private.key*. The contents of the key look similar to the following example:

```
cat private.key

-----BEGIN RSA PRIVATE KEY-----
...[certificate text]
-----END RSA PRIVATE KEY-----
```

Note that even though the file is called *private.key* and the file contains some text that suggests that this is only the private key, the public key is also embedded within this file. So the single file represents the complete key pair. Thus, obtaining a copy of the public key is easier because the key is stored on the same file as the private key.

To create an encrypted key with a passphrase, run the same command but specify a cipher to use to encrypt the key with, for example:

```
sudo openssl genrsa -aes256 -out private.key 2048
```

```
Generating RSA private key, 2048 bit long modulus
............+++
.............................................................+++
e is 65537 (0x10001)
Enter pass phrase for private.key:
Verifying - Enter pass phrase for private.key:
```

In the previous example, the AES cipher is used with a 256 bit key. The command prompts you to enter a passphrase and verify it. The contents of the key file indicate that the key is encrypted, as shown in the following example:

```
cat private.key

-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: AES-256-CBC,2417E359B45960CD107A390748945752

key-content
-----END RSA PRIVATE KEY-----
```

If you create an encrypted key file and then decide that you would prefer a file that's not encrypted or doesn't require a passphrase, you can decrypt it by running the following command:

```
sudo openssl rsa -in private.key -out unencrypted.key

Enter pass phrase for private.key:
writing RSA key
```

You're prompted for the passphrase on the encrypted key, which is stored in *private.key*, and the unencrypted version of the same key is written to the file *unencrypted.key*.

All OpenSSL keys are generated in Privacy Enhanced Mail (PEM) format, which is a plain text format that encapsulates the content of the key as a base64 encoded string. Certificates can be encoded by using several different formatting conventions. For more information about changing the format of a certificate, see Changing Key or Certificate Format.

You can view the contents of a private key as follows:

```
sudo openssl rsa -text -in private.key
```

Notably, a private key also contains its public key counterpart. This public key component is used when submitting a CSR or when creating a self-signed certificate. The public key component can be viewed by using the following command:

```
sudo openssl rsa -pubout -in private.key
```

Creating Certificate Signing Requests With OpenSSL

A private key can be used to create a Certificate Signing Request (CSR). A public and private key can be used to encrypt communications. However, a client must still validate the public certificate presented for use with encrypted communication as coming from an expected and trusted source. Without some way to validate the public key, the client can easily succumb to man-in-the-middle style attacks that would render encryption futile.

To solve this problem, public key infrastructure typically involves third parties, called Certification Authorities (CAs), that can sign a certificate as authentic for a particular public key. If the client has a copy of the CA certificate, the client can validate a certificate for a domain, based on the signature in the certificate. Most systems are installed with some trusted CA certificates by default. To check the CA certificates that are trusted by the system, use the following command:

```
sudo openssl version -d
```

By default, this directory is `/etc/pki/tls` and the `/etc/pki/tls/certs` subdirectory contains all the trusted certificates.

To obtain a signed certificate from a CA, a CSR must be generated using the public key component within its associated private key. The CSR is then presented to the CA which can validate the information in the request and use this information to generate a valid and signed public certificate. The CSR is associated with a domain name for the host or hosts on which the certificate is be used. The CA uses this information to create a certificate with a specified expiry date.

The following example shows the command syntax for interactively creating a CSR from a private key:

```
sudo openssl req -new -key private.key -out domain.example.com.csr

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) [XX]:GB
State or Province Name (full name) []:.
Locality Name (eg, city) [Default City]:London
Organization Name (eg, company) [Default Company Ltd]:Example Ltd
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:domain.example.com
Email Address []:webmaster@example.com

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

Note that the default values can be configured in the `/etc/pki/tls/openssl.cnf` file. The `Common Name` is the most important value in the CSR. This value associates the certificate request with the hostname and domain name for the host on which the certificate is to be used. Note that if a client connects to a host that's issued a certificate for a different domain, the certificate is invalid.

You can generate a CSR and private key at the same time. With the following command, you can specify values for the different fields in the CSR on the command line:

```
sudo openssl req -new -nodes '/CN=domain.example.com/O=Example Ltd/C=GB/
L=London' \
  -newkey rsa:1024 -keyout private.key -out domain.example.com.csr
```

You can view the information contained in a CSR as follows:

```
sudo openssl req -in domain.example.com.csr -noout -text
```

After you have a CSR, you can submit it to a CA. The CA uses the CSR to generate a signed certificate and then returns the certificate with a certificate chain that can be used to validate the certificate.

Signing Certificates With OpenSSL

For environments where you don't have control over client systems, always use a recognized, independent CA to sign certificates. Operating system and software vendors negotiate with independent CAs to include CA validation certificates, along with the software that they distribute. Obtaining validation certificates from major CA providers means that most users don't have to manage their own trusted CA certificate list. Any browser visiting a website over HTTPS can validate the site's public certificate by matching the CA signature to the CA certificates that it has in its own store.

If you have control over client systems, you can either provide the clients with the self-signed certificate, directly; or, you can set up private CA certificate to sign all the certificates that are used within the organization and then distribute the CA certificate to clients. Using the second approach validates all certificates that are signed within the organization, which results in tighter control over the security of the certificates within the organization, which can result in lower infrastructure costs.

Creating Self-Signed Certificates for Testing and Development

Self-signed certificates are often created for development and testing purposes. Because these certificates are not validated by trusted CAs, trust for these certificates need to be configured manually. If the private key is compromised it can't be revoked but must be manually removed from trust allow list. Never use these certificates in production environments. A CA-signed certificate is always preferable to a self-signed certificate. However, using self-signed certificates can be less costly and useful for testing and development, without the hassle of managing private CA or obtaining CA-signed certificates for every test platform.

With the `openssl` command, you can generate self-signed certificates that can be used immediately. This command creates a CSR for the private key and then generates an X.509 certificate directly from the CSR, signing the certificate with itself.

For this reason, the command is similar to the command that you would run to create a private key and CSR, with the exception that you must also specify the period of validity. As a good practice, only generate a self-signed certificate for the duration needed for testing purposes. This way, if the private key is compromised, the validity period is limited, and a new certificate can be generated when the old certificate expires.

For example, you would use the following command to create a self-signed X.509 certificate that's valid for 30 days.

```
sudo openssl req -new -x509 -days 30 -nodes -newkey rsa:2048 -keyout private.key \
  -out public.cert -subj '/C=US/ST=Ca/L=Sunnydale/CN=www.example.com'
```

The generated *private.key* file contains the private key and the *public.cert* file contains the self-signed certificate. Typically, you name these files with the same value as the Common Name so that you can track which certificates and keys apply to which host and domain name.

Note that you can set the `-newkey` value to suit custom algorithm and key size requirements. In this example, the algorithm is set to RSA and the key size is set at 2048 bits.

You can copy the self-signed certificate file to the trusted certificate store for any client system and the client system validates the certificate as a match whenever it makes a connection to the host that serves it.

You can also use the `keytool` command to generate self-signed certificates, but this command's primary purpose is to install and manage JSSE (Java Secure Socket Extension) digital certificates for use with Java applications. See Java for more information.

Creating a Private Certification Authority

By creating a private Certification Authority (CA), you can process CSRs for all the certificates within the organization. You're also capable of managing the Certificate Revocation List (CRL), which client systems can use to detect whether a certificate is still valid or if it has been revoked.

This approach is better than using self-signed certificates because you can control revocation. However, the CA certificate must still be distributed to all the client systems that need to validate public certificates within the organization.

Create the CA Root

The CA Root is the fundamental certificate for a CA and is not usually used to sign server or client certificates. The CA Root is usually used to sign one or more intermediary certificates to grant them power to sign other certificates. This model means that if a CA Intermediary private key is compromised, the CA Intermediary can be added to a certificate revocation list and all of the certificates that are signed by the Intermediary are automatically invalidated.

This model helps to protect the integrity of the entire public key infrastructure. Without a CA Root, there is no public key infrastructure, as the CA Root is used to create the chain of trust that is used to validate all certificates in the hierarchy. The CA Root should generally be created and maintained on a system that is completely isolated, ideally with minimal or no network access and no direct access to the Internet. The security measures that are implemented around the CA Root are critical to the security of the entire public key infrastructure. If the CA Root private key is compromised, every certificate that is ever signed by the entire chain may be compromised as well.

To create a CA Root for your organization, you must create a root key pair according to a defined configuration that OpenSSL can use to manage the CA configuration and the database of metadata for certificates that it issues.

There are several steps that you need to take to create the CA Root, which are described in the following procedures and examples.

Create a CA Directory Structure

All of the certificates and metadata that are managed by the CA Root are stored in a specific directory structure and within some preconfigured files. You should create the structure according to your own requirements, but follow these general steps:

1. Create a directory to store all of the CA-related data:

   ```
   sudo mkdir /root/ca
   ```

You can store this directory anywhere on the system. However, keep in mind that it contains very sensitive data, so ensure that it is located somewhere with very restricted access.

2. Change to the CA directory to perform all of the subsequent steps in this procedure:

   ```
   sudo cd /root/ca
   ```

3. Create directories to contain the following for your system: CA certificates, CA database content, Certificate Revocation List, all newly issued certificates, and your private keys:

   ```
   sudo mkdir certs db crl newcerts private
   ```

4. Protect your private keys to ensure that access to the directory where these are stored is limited to the current user:

   ```
   sudo chmod 700 private
   ```

5. Create the files that will be used for your CA database:

   ```
   sudo touch db/index.txt
   sudo openssl rand -hex 16  > db/serial
   sudo echo 1001 |sudo tee  db/crlnumber
   ```

Create a CA Root Configuration File

The CA Root configuration should be created and stored in the directory where all of the CA related content is stored. For example, you would create a file in `/root/ca/ca-root.conf` and populate it with the following content:

```
[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             = "AU"
organizationName        = "Example Org"
commonName              = "Root CA"

[ca_default]
home                    = .
database                = $home/db/index.txt
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        = 30
default_md              = sha256
policy                  = policy_strict

[policy_strict]
# The root CA should only sign intermediary certificates that match.
```

```
# See the POLICY FORMAT section of `man ca`.
countryName             = match
stateOrProvinceName     = optional
organizationName        = match
organizationalUnitName  = optional
commonName              = supplied
emailAddress            = optional

[policy_loose]
# Allow the intermediary CA to sign a more diverse range of certificates.
# See the POLICY FORMAT section of the `ca` manual page.
countryName             = optional
stateOrProvinceName     = optional
localityName            = optional
organizationName        = optional
organizationalUnitName  = optional
commonName              = supplied
emailAddress            = optional

[req]
# Standard Req options
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]
# Extensions for a the CA root (`man x509v3_config`).
basicConstraints        = critical,CA:true
keyUsage                = critical,keyCertSign,cRLSign
subjectKeyIdentifier    = hash

[intermediary_ext]
# Extensions for an intermediary CA.
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always,issuer
basicConstraints = critical, CA:true, pathlen:0
keyUsage = critical, digitalSignature, cRLSign, keyCertSign

[server_ext]
# Extensions for server certificates.
basicConstraints = CA:FALSE
nsCertType = server
nsComment = "OpenSSL Generated Server Certificate"
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid,issuer:always
keyUsage = critical, digitalSignature, keyEncipherment
extendedKeyUsage = serverAuth

[client_ext]
# Extensions for client certificates.
basicConstraints = CA:FALSE
nsCertType = client, email
nsComment = "OpenSSL Generated Client Certificate"
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid,issuer
keyUsage = critical, nonRepudiation, digitalSignature, keyEncipherment
```

```
extendedKeyUsage = clientAuth, emailProtection

[crl_ext]
# Extension for CRLs.
authorityKeyIdentifier=keyid:always

[ocsp]
# Extension for OCSP signing certificates.
basicConstraints = CA:FALSE
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid,issuer
keyUsage = critical, digitalSignature
extendedKeyUsage = critical, OCSPSigning
```

The previous example shows a configuration that contains many optional entries that can help when performing different operations with OpenSSL. Most importantly, the configuration defines the extensions that can be applied to different certificate types to validate the types of operations the are valid for the certificate. This configuration also defines different policies that can be applied when signing certificates. For instance, you can use a strict policy to ensure that a particular metadata is specified; and, that it matches the CA values within a CSR, if the certificate is to be signed. This policy is important for generating intermediary CA certificates. A less restrictive policy can be applied for other certificates that are signed, either by the CA Root or any intermediary.

The following are descriptions of the various sections within this configuration file:

- **[default]**

  The default section defines some basic configuration information such as URLs where information such as the root certificate and the published revocation list for this CA might be published. Note that the `name` and `domain_suffix` entries here are used as variables to help construct some of these URLs and are also used to name and reference key files and certificates. You may wish to use the system hostname and the system domain for these values. This configuration entry also references the location of the default CA configuration entry at `ca_default`.

- **[ca_dn]**

  This section defines some default values for certificates that are generated for this CA's distinguished name. These values are written into the CSR and the self-signed certificate that is generated from it for the CA Root certificate.

- **[ca_default]**

  This section provides the configuration that controls the entire CA. This information provided maps the directories that were created for this CA to the configuration so that OpenSSL can correctly update files and store certificates and keys in the correct places. This section also defines some default values such as how many days a certificate is valid for and how many days the certificate revocation list is valid. Because this configuration is for a root CA, the number of days that the certificate is valid for can be set to 10 years, since a change to the root CA would mean that all of subsequent certificates in the infrastructure would also need to be re-issued. You can view all of the configuration file options in the `CA(1)` manual pages.

- **[policy_strict]**

  This section describes a strict policy that should be followed when signing some certificates, such as the intermediary CA certificates. The policy defines rules around the metadata within the certificate. For instance, there are rules that the country name and organizational name match the CA certificate. Other fields are optional, but a common name must be supplied.

- **[policy_loose]**

  This section is used for other certificates that are signed by this CA and its intermediaries, where a less restrictive policy is allowed. This policy entry allows the majority of fields to be optional and only requires that the common name is supplied.

- **[req]**

  This section is used one time to create the CA certificate request and defines the default options to use when the certificate request is generated, for example, a key length of 4096 bits for the root CA. There is also an option that points to the CA distinguished name that references the `ca_dn` section of this configuration file for obtaining the default values to use within the certificate request.

- **[ca_ext]**

  This extensions section defines those operations for which a certificate is valid. For the root CA, this certificate must be valid in order to sign all of the intermediary CA certificates and essentially has full rights. For more information about extensions, see the `X509V3_CONFIG(5)` manual page.

- **[intermediary_ext]**

  This section is separate extension configuration for certificates that are signed as intermediary CAs. This certificate has the same rights as the root CA, but is unable to sign certificates for further intermediary CAs, controlled with the `pathlen:0` within the certificate's `basicConstraints` option.

- **[server_ext]**

  This section includes typical extension options for server-side certificates, which are usually used for services like HTTPS and server-side mail services, and so on. These certificates are issued for validation and encryption purposes; they do not have signing rights. The configuration entry can be referenced when signing a certificate for this purpose.

- **[client_ext]**

  This section includes client-side certificates, which are often used for remote authentication, where a user may provide a certificate to validate and authenticate access to a system. These certificates also have specific extensions that control usage. This configuration entry can be used when signing a certificate for client side certificates to ensure that the correct extensions are applied to the certificate.

- **[crl_ext]**

  This extension is automatically applied when creating a CRL, but this extension is provided for completeness. See Manage a Certificate Revocation List

- **[ocsp]**

  The Online Certificate Status Protocol (OCSP) is an alternative approach to CRLs. An OCSP server can be set up to handle requests by client software to obtain the status of a certificate from a resource that is referenced in a signed certificate. Special extensions exist for this purpose. The `OCSP(1)` manual page can provide more information. See also Configure and Run an OCSP Server.

Create and Verify the CA Root Key Pair

Create a private key and a certificate signing request for the CA root using the configuration values that you have specified in the `ca-root.conf` file and save the private key to `private/root-ca.key`. Since this is the most valuable key in your entire infrastructure, ensure that you use a lengthy and suitable passphrase to protect it.

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

Now, create a self-signed certificate by using the CSR and the `ca-root.conf` file. Take care to specify that the certificate must use the extensions defined in the `ca_ext` portion of the configuration.

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

Using configuration from ca-root.conf
Enter pass phrase for ./private/root-ca.key:
Check that the request matches the signature
Signature ok
Certificate Details:
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            8f:75:11:1a:8e:33:b2:d1:09:a8:bf:07:9c:67:c8:3e
        Issuer:
            countryName              = AU
            organizationName         = Example Org
            commonName               = Root CA
        Validity
            Not Before: Oct 29 12:23:04 2019 GMT
            Not After : Oct 26 12:23:04 2029 GMT
        Subject:
            countryName              = AU
            organizationName         = Example Org
            commonName               = Root CA
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                RSA Public-Key: (4096 bit)
                Modulus:
                    00:b9:41:d6:10:36:d4:12:d3:5d:52:29:60:fc:e0:
                    90:34:f6:fb:3e:99:10:33:a1:1d:54:77:3c:11:37:
                    2d:78:c3:3c:3f:40:69:37:fc:de:59:20:c1:1c:07:
                    83:f7:ae:2b:19:03:a7:e8:c6:d6:88:03:b4:ec:60:
                    36:3d:f6:da:59:58:cc:18:18:3e:43:c9:79:11:5b:
                    cf:9e:15:a7:29:fe:dc:4f:7b:0b:93:f0:9a:2b:97:
                    0f:ab:3e:38:7c:e7:c7:d3:5e:34:e2:40:d0:fd:f2:
                    e4:5e:2c:8a:8e:11:83:de:6b:c4:5c:b8:ec:4b:9c:
                    d2:3f:06:3d:53:a6:4b:a6:e3:c6:f6:24:a2:8c:fb:
                    bf:9e:19:d7:60:4b:c5:b6:48:e4:5d:60:4f:2c:47:
                    ca:4a:31:79:bc:7b:5a:25:90:fc:d2:44:a1:79:73:
                    2e:e1:88:a0:73:1f:82:d3:63:3e:67:94:20:f8:be:
                    21:9b:c3:14:4d:3e:9b:19:33:be:9b:cb:e5:54:9f:
                    a7:3f:05:d1:64:56:5f:43:62:65:5b:89:f4:f1:e3:
                    24:e8:1c:d5:03:36:86:ce:9e:76:c7:52:dc:88:f5:
                    d9:87:62:00:82:4d:14:de:a3:60:21:54:12:83:da:
                    8e:8e:5f:63:c3:93:5a:e2:b9:60:16:74:06:c7:46:
                    49:6d:c2:7e:6c:3a:50:3b:bf:c5:d6:20:65:bd:21:
```

```
                    a9:ad:b2:1c:4c:13:bf:fd:b8:e1:04:b8:46:c9:6c:
                    29:db:f3:a6:50:3d:2b:9b:83:49:bb:61:c2:8e:94:
                    08:52:84:f2:6d:33:4b:1f:e0:90:ea:a8:ec:d6:ff:
                    97:b8:3d:74:9a:64:d0:f7:22:7d:22:fc:93:47:68:
                    54:63:7c:10:0a:82:2f:84:3f:56:28:cf:8a:03:76:
                    77:b9:db:af:02:6d:b9:36:7e:63:da:f5:d2:a5:6d:
                    54:86:e1:be:f0:e1:54:13:dd:63:0a:53:8e:55:24:
                    90:40:af:f6:38:47:d3:00:0c:ba:66:6a:cc:4b:df:
                    28:fe:02:74:eb:28:15:11:ca:da:a7:86:0f:1f:bd:
                    c4:ac:b9:b1:c7:cc:2a:2a:db:6e:fd:e6:8e:7b:02:
                    17:5e:a7:7d:08:53:e2:a4:69:ca:6b:1f:f1:74:5b:
                    ac:86:2a:f2:b0:80:ea:b7:30:c5:14:c8:12:1e:66:
                    5e:2f:f5:d5:a8:09:39:b4:23:25:fc:ca:35:d5:c0:
                    73:79:a0:8a:12:25:27:ee:f5:ce:9a:97:c0:27:31:
                    ac:21:98:8f:34:25:a5:7a:42:5c:a0:a1:5d:64:39:
                    aa:6a:5e:54:50:5e:ad:c4:fe:c7:93:b1:c0:f7:c9:
                    91:43:93
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Basic Constraints: critical
                CA:TRUE
            X509v3 Key Usage: critical
                Certificate Sign, CRL Sign
            X509v3 Subject Key Identifier:
                3C:D9:C3:56:BD:C0:45:83:C8:2B:C7:0F:96:30:CF:2A:55:23:B5:9D
Certificate is to be certified until Oct 26 12:23:04 2029 GMT (3650 days)
Sign the certificate? [y/n]:y


1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Data Base Updated
```

You are prompted for your private key passphrase to continue. After being shown the values of the certificate, you are prompted to sign the certificate. After signing the certificate, you can commit it to your CA database. The database files are updated to track this certificate within your public key infrastructure.

You can view the `db/index.txt` file to see the CA root certificate entry:

```
sudo cat db/index.txt

V   291026122304Z        8F75111A8E33B2D109A8BF079C67C83E  unknown /C=AU/
O=Example Org/CN=Root CA
```

The values that are displayed on each line within the database index include:

1. Status (`V` for valid, `R` for revoked, `E` for expired).
2. Expiry date in `YYMMDDHHMMSSZ` format.
3. Revocation date or empty if not revoked (in this example output, the field is empty).
4. Hexadecimal serial number.
5. File location or `unknown`, if not known.
6. Distinguished name.

Create an intermediary CA

The next step in creating your infrastructure is to create an intermediary CA that can process all of your server and client certificates. This is important because if the intermediary CA private key is compromised, the root CA can revoke its certificate and invalidate any other certificate that has been issued by that intermediary.

The intermediary CA should ideally be hosted on an alternate server with wider access as it will be used to handle the majority of your certificate requests. The intermediary CA is an exact model of the root CA, with the exception that its own certificate is signed by the root CA and is configured with the appropriate extensions to process signing requests.

Create A CA Directory Structure

On the intermediary CA host, perform the same operations that you performed to create the root CA directory structure, but name the parent directory appropriately so that it is clear that the configuration is for an intermediary, for example:

```
sudo mkdir /root/ca-intermediary
sudo cd /root/ca-intermediary/
sudo mkdir certs db crl newcerts private
sudo chmod 700 private
sudo touch db/index.txt
sudo openssl rand -hex 16  > db/serial
sudo echo 1001 |sudo tee  db/crlnumber
```

Create the intermediary CA Configuration

The intermediary CA configuration is almost identical to the configuration that you created for the CA root, with a few modifications that make it specific to the intermediary. Modifications are indicated in **bold** text in the following example:

```
[default]
name                    =
                              sub-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             = "AU"
organizationName        = "Example Org"
commonName              = "
                              Intermediary CA
                           "

[ca_default]
home                    = .
database                = $home/db/index.txt
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        = 30
default_md              = sha256
policy                  = policy_strict

[policy_strict]
# The root CA should only sign intermediary certificates that match.
# See the POLICY FORMAT section of `man ca`.
countryName             = match
stateOrProvinceName     = optional
organizationName        = match
organizationalUnitName  = optional
commonName              = supplied
emailAddress            = optional

[policy_loose]
# Allow the intermediary CA to sign a more diverse range of certificates.
# See the POLICY FORMAT section of the `ca` manual page.
countryName             = optional
stateOrProvinceName     = optional
localityName            = optional
organizationName        = optional
organizationalUnitName  = optional
commonName              = supplied
emailAddress            = optional

[req]
# Standard Req options
default_bits            = 4096
encrypt_key             = yes
default_md              = sha256
utf8                    = yes
string_mask             = utf8only
prompt                  = no
distinguished_name      = ca_dn
req_extensions          = intermediary_ext

[ca_ext]
# Extensions for a the CA root (`man x509v3_config`).
basicConstraints        = critical,CA:true
keyUsage                = critical,keyCertSign,cRLSign
subjectKeyIdentifier    = hash

[intermediary_ext]
# Extensions for an intermediary CA.
subjectKeyIdentifier = hash
# authorityKeyIdentifier = keyid:always,issuer
basicConstraints = critical, CA:true, pathlen:0
keyUsage = critical, digitalSignature, cRLSign, keyCertSign

[server_ext]
# Extensions for server certificates.
basicConstraints = CA:FALSE
nsCertType = server
nsComment = "OpenSSL Generated Server Certificate"
subjectKeyIdentifier = hash
```

```
authorityKeyIdentifier = keyid,issuer:always
keyUsage = critical, digitalSignature, keyEncipherment
extendedKeyUsage = serverAuth

[client_ext]
# Extensions for client certificates.
basicConstraints = CA:FALSE
nsCertType = client, email
nsComment = "OpenSSL Generated Client Certificate"
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid,issuer
keyUsage = critical, nonRepudiation, digitalSignature, keyEncipherment
extendedKeyUsage = clientAuth, emailProtection

[crl_ext]
# Extension for CRLs.
authorityKeyIdentifier=keyid:always

[ocsp]
# Extension for OCSP signing certificates.
basicConstraints = CA:FALSE
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid,issuer
keyUsage = critical, digitalSignature
extendedKeyUsage = critical, OCSPSigning
```

Note that in the `intermediary_ext` section, a line has been commented out because the intermediary will not issue any further intermediary certificates. The intermediary is unaware of the certificate issuer until the certificate is signed. If you attempt to create the CSR while this line is still included in the configuration, it fails because it cannot determine which certificate issuer to include this metadata in the CSR.

Save the configuration file as `intermediary.conf`.

Create a CSR for the intermediary CA

Create a CSR for the intermediary certificate:

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

This certificate is also a signing certificate, so it is important to protect it with a passphrase to help prevent its unauthorized use and maintain the security of your infrastructure. Enter the passphrase when prompted.

Create a signed certificate for the intermediary CA

Copy the `sub-ca.csr` that you generated in the previous step to the `/root/ca` directory on the system where your root CA is hosted. On the root CA host, run the following commands to generate a signed certificate from the CSR and apply the intermediary signing extension:

```
sudo cd /root/ca
sudo openssl ca -config ca-root.conf -in sub-ca.csr -out newcerts/sub-ca.crt \
-extensions intermediary_ext
```

You are prompted for the root CA passphrase, then presented with the certificate content and prompted to sign it. Check that the certificate contents make sense before you sign it. You can see that the certificate is issued by the Root CA and contains the Intermediary CA in the Subject. You can also see that the correct extensions are applied to the certificate.

After the certificate is signed, you are prompted to update the database.

The newly signed certificate is created as `newcerts/sub-ca.crt`.

Create a certificate chain file

Because no systems are aware of the root CA certificate, you should create a certificate chain that includes the public certificate for the root CA with the newly created intermediary CA certificate. In this way, hosts only need a copy of the chained certificate to validate any certificates that are issued by the intermediary CA. To create the certificate chain, simply join the two public certificates by running the following command on the root CA host:

```
sudo cat root-ca.crt newcerts/sub-ca.crt > newcerts/chained-sub-ca.crt
sudo chmod 444 newcerts/chained-sub-ca.crt
```

Copy the `newcerts/sub-ca.crt` and `newcerts/chained-sub-ca.crt` certificate back to the `/root/ca-intermediary/` directory on the intermediary CA host. You can now use this certificate to process server and client CSRs and to generate CRLs.

When you return a signed certificate for any given CSR, include the `chained-sub-ca.crt` certificate so that it can be installed on the host where the certificate will be used and distributed to any client that needs to validate the signed certificate.

Process CSRs and Sign Certificates

As systems generate CSRs using the process that is described in Creating Certificate Signing Requests With OpenSSL, they must submit them to a CA to be signed.

All subsequent CSR processing for server and client-side certificates should be performed by an intermediary CA that is configured within your environment or by an external third party CA.

To process a CSR, copy it to the */root/ca-intermediary* directory on your intermediary CA host and then use the `openssl ca` command to sign it with the appropriate extension configuration.

For example, to sign a server-side certificate for a CSR named *`www.example.com.csr`* , run the following command:

```
sudo openssl ca -config intermediary.conf -extensions server_ext -days 375 \
-in www.example.com.csr -out newcerts/www.example.com.crt
```

Note that we specify the number of days for which the certificate is valid. For a server-side certificate, the number of days should be limited to a value significantly lower than a CA certificate's validity. It is important to select the correct extensions to apply to the certificate. These extensions map to definitions that are within your configuration file.

You are prompted for the intermediary CA key passphrase and then prompted to sign the certificate and update the database.

You should return the certificate, along with the chained CA certificate, so that these can be distributed to validate the certificate.

Manage a Certificate Revocation List

The certificate revocation list is used to identify certificates that have been issued by a signing CA and revoked. The list also tracks the reason that a certificate was revoked.

Generate the CRL

On each CA host, you should create an empty CRL that can be updated as you need to revoke certificates. For example, on an intermediary CA, you would use the following command:

```
sudo cd /root/ca-intermediary
sudo openssl ca -config intermediary.conf -gencrl -out crl/sub-ca.crl
```

Note that the CRL should be published to the URL that is defined in your configuration file to keep track of certificates that are revoked by the CA. You should configure a web service to serve the `sub-ca.crl`, if possible.

You can check the contents of a CRL as follows:

```
sudo openssl crl -in crl/sub-ca.crl -noout -text
```

If the CRL was just created, it is empty. A new CRL should be created periodically, based on the configuration value that is set in the CA configuration file for `default_crl_days`. By default, it is set for every 30 days.

Revoke a certificate

Every signed certificate contains the serial number that is issued by the signing CA. You can view this serial number within a certificate as follows:

```
sudo openssl x509 -serial -noout -in server.crt
```

This serial number identifies the certificate within the CA signing database and can also be used to identify the certificate stored by the CA that signed it so that the CA can revoke it.

On the CA where the certificate was issued, you can find the certificate with the matching serial number in the `certs` directory. For example, on an intermediary host, for a certificate with serial number *8F75111A8E33B2D109A8BF079C67C83F*, it would be as follows:

```
sudo cd /root/ca-intermediary
sudo ls certs/8F75111A8E33B2D109A8BF079C67C83F*

certs/8F75111A8E33B2D109A8BF079C67C83F.pem
```

You can also check the details for the certificate in the CA database:

```
sudo grep 8F75111A8E33B2D109A8BF079C67C83F db/index.txt
```

To revoke this certificate, the signing CA must issue the following command:

```
sudo openssl ca -config intermediary.conf -revoke certs/
8F75111A8E33B2D109A8BF079C67C83F.pem \
-crl_reason keyCompromise
```

Note that you should specify the reason for revoking the certificate, as this reason is used in the certificate revocation list. Options include the following: unspecified, keyCompromise, CACompromise, affiliationChanged, superseded, cessationOfOperation, certificateHold, and removeFromCRL. For more information, see the CA(1) manual page.

When a certificate is revoked, the CA database is updated to reflect this change and the status is set to R for the certificate that is listed in the db/index.txt file.

The database file is used to generate the CRL each time it is created. It is good practice generating a new CRL as soon as you revoke a certificate. In this way, this list is kept up to date. See Generate the CRL for more information.

Configure and Run an OCSP Server

The Online Certificate Status Protocol (OCSP) provides an alternative to CRLs and includes its own publishing mechanism. OpenSSL includes an option to run as an OCSP server that can respond to OCSP queries.

Note that OCSP is preferred over CRLs. Usually, it is a good idea to make sure that an OCSP server is running for your CA, particularly if the OCSP URL appears in your configuration, as this URL is included in each certificate that is signed by the CA. Any client software can confirm the revocation status of a certificate by querying the OCSP server.

For any CA, create a key and CSR for the OCSP server:

```
sudo openssl req -new -newkey rsa:2048 -subj "/C=AU/O=Example Org/CN=OCSP
Responder" \
-keyout private/ocsp.key -out ocsp.csr
```

Create a signed certificate from the ocsp.csr CSR file:

```
sudo openssl ca -config intermediary.conf -extensions ocsp -days 187 -in
ocsp.csr \
-out newcerts/ocsp.crt
```

Because the OCSP certificate is responsible for handling revocation, it cannot be revoked. Therefore, it is a good practice to set the validity period on the certificate to a manageable, but relatively short period. In this example, the validity period has been set to 187 days, which means that it needs to be refreshed every 6 months.

To run an OCSP server on the current CA, you can use the tool provided within OpenSSL. For example, you could use the following command:

```
sudo openssl ocsp -port 9080 -index db/index.txt -rsigner newcerts/ocsp.crt \
-rkey private/ocsp.key -CA sub-ca.crt -text
```

Note that the command specifies the CA db/index.txt file directly, which means that as certificates are revoked, the OCSP server becomes aware of them automatically. When you run the command, you are prompted for the OCSP key passphrase. The server continues to run until you kill the process or escape by using a control sequence such as Ctrl-C.

You can test the service by checking the ocsp.crt file. Use the openssl command as follow to run an OCSP query:

```
sudo openssl ocsp -issuer sub-ca.crt -CAfile chained-sub-ca.crt -cert newcerts/
ocsp.crt \
-url http://127.0.0.1:9080

Response verify OK
newcerts/ocsp.crt: good
        This Update: Oct 30 15:48:11 2019 GMT
```

The response in the previous example indicates whether the verification has succeeded and provides a status of `good` if the certificate has not been revoked. A status of `revoked` is returned if it has been revoked.

Debugging and Testing Certificates With OpenSSL

The following are some examples that show how you can use OpenSSL commands to work with existing certificates to debug and test the infrastructure. The examples provided here aren't comprehensive and are intended to supplement the existing OpenSSL manual pages.

Examining Certificates

Display the information contained in an X.509 certificate.

```
sudo openssl x509 -text -noout -in server.crt
```

Display the SHA1 fingerprint of a certificate.

```
sudo openssl x509 -sha1 -noout -fingerprint -in server.crt
```

Display the serial number of a signed certificate:

```
sudo openssl x509 -serial -noout -in server.crt
```

Check That a Private Key Matches a Certificate

The modulus and the public exponent parts of the key and the certificate must match. These values are usually long and difficult to check. The easiest way to compare the modulus in the key and certificate is to create an MD5 hash of each and compare those instead, for example:

```
sudo openssl x509 -noout -modulus -in server.crt | openssl md5
sudo openssl rsa -noout -modulus -in server.key | openssl md5
```

You can equally check the modulus in a CSR to see if it matches a key or certificate as follows:

```
sudo openssl req -noout -modulus -in server.csr | openssl md5
```

Changing Key or Certificate Format

Convert a root certificate to a form that can be published on a website for downloading by a browser:

```
sudo openssl x509 -in cert.pem -out rootcert.crt
```

Convert a base64 encoded certificate (also referred to as PEM or RFC 1421) to binary DER format:

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

Convert the base64 encoded certificates for an entity and its CA to a single PKCS7 format certificate:

```
sudo openssl crl2pkcs7 -nocrl -certfile entCert.cer -certfile CACert.cer -out
certificate.p7b
```

Check Certificate Consistency and Validity

Verify a certificate including the signing authority, signing chain, and period of validity:

```
sudo openssl verify cert.pem
```

Decrypting Keys and Adding or Removing Passphrases

If you create an encrypted key file and decide that the file isn't encrypted or doesn't require a passphrase, you can decrypt it by using the following command:

```
sudo openssl rsa -in private.key -out unencrypted.key

Enter pass phrase for private.key:
writing RSA key
```

You're prompted for the passphrase on the encrypted key, which is stored in *private.key*, and the unencrypted version of the same key is written to the *unencrypted.key* file.

To encrypt an unencrypted key and add a passphrase to protect it, run the following command:

```
sudo openssl rsa -aes256 -in unencrypted.key -out private.key
```

In the previous example, the AES cipher is used with a 256 bit key. The command prompts you to enter a passphrase and to verify it. The new encrypted key file is written to *private.key*.

You can add or remove a passphrase from the private key at any time without affecting its public key counterpart. Adding a passphrase protects the private key from use by an unauthorized or malicious user, but comes with an added inconvenience, in that services that use the private key always require manual intervention to start or restart. If you remove the passphrase from a key, ensure that it's stored with strict permissions and that it's not copied to systems that don't require it.

Using OpenSSL to Test SSL/TLS Configured Services

Test a self-signed certificate by configuring a server that listens on port 443:

```
sudo openssl s_server -accept 443 -cert cert.pem -key prikey.pem -www
```

Test the client side of a connection. This command returns information about the connection including the certificate by which you can directly input HTTP commands:

```
sudo openssl s_client -connect server:443 -CAfile cert.pem
```

Extract a certificate from a server as follows:

```
sudo echo | openssl s_client -connect server:443 2>/dev/null | \
sed -ne '/BEGIN CERT/,/END CERT/p' |sudo tee svrcert.pem
```

Using OpenSSL for File Encryption and Validation

You can also use OpenSSL to encrypt or decrypt any file type and to create digests that can be signed and used to validate the contents and the origin of a file. The following are some examples of how you might use the `openssl` command.

Encrypt a file by using Blowfish:

```
sudo openssl enc -blowfish -salt -in file -out file.enc
```

Decrypt a Blowfish-encrypted file:

```
sudo openssl enc -d -blowfish -in file.enc -out file.dec
```

Create an SHA1 digest of a file:

```
sudo openssl dgst -sha1 file
```

Sign the SHA1 digest of a file using the private key stored in the file `prikey.pem`:

```
sudo openssl dgst -sha1 -sign prikey.pem -out file.sha1 file
```

Verify the signed digest for a file using the public key stored in the file `pubkey.pem`:

```
sudo openssl dgst -sha1 -verify pubkey.pem -signature file.sha1 file
```

More information About OpenSSL

For more information about OpenSSL, see the `openssl(1)`, `ciphers(1)`, `dgst(1)`, `enc(1)`, `req(1)`, `s_client(1)`, `s_server(1)`, `verify(1)`, and `x509(1)` manual pages.

3

Setting Up TLS/SSL with Other Tools

This chapter describes some other tools available for setting up TLS/SSL that you may consider.

There are various reasons to choose one tool over another. For example, some tools are more light weight than others, or some are targeted to specific environments, such as the keytool for Java. Some not only cover certificates infrastructure management but also include other features, APIs, and libraries for developing applications that enable various other secure network protocols and security standards. Although this book does not provide details about such features, you can find more information about them in corresponding manual pages and documentation from opensource projects.

GnuTLS

This chapter describes the `certtool` GnuTLS certificate tool that is available in Oracle Linux and how to use it to create certificate signing requests, self-signed certificates, and privately owned CA certificates. GnuTLS is a library implementing the SSL, TLS and DTLS protocols and technologies around them for the purpose of securing communications. It includes an application programming interface (API) written in C language to access the secure communications protocols and additional APIs to parse and write structures such as X.509, PKCS #12, and OpenPGP.

To use certtool, install the gnutls-utils package avaiable from the Application Stream repository.

```
sudo dnf install gnutls-utils
```

> **Note:**
>
> If you are using Oracle Linux 7, you can install the package from the `ol7_latest` yum repository by using `yum` instead of `dnf`.

The following examples show how to use the `certtool` command to create certificate signing requests, self-signed certificates, and privately owned CA certificates.

- To generate a private key, do the following.

  ```
  sudo certtool --generate-privkey --outfile private_key_file
  ```

  In the previous example, *private_key_file* is the name of the private key file.

- To generate a CSR, do the following:

  ```
  sudo certtool --generate-request --load-privkey private_key_file --outfile csr_file
  ```

  In the previous example, *csr_file* is the name of the CSR file.

- To generate a self-signed certificate, do the following:

  ```
  sudo certtool --generate-self-signed --load-privkey 
  private_key_file --outfile self_signed_certificate_file
  ```

For more information, see the `certtool(1)` manual page and the GnuTLS open source project at https://www.gnutls.org/.

NSS

This chapter describes the `certutil` Network Security Service (NSS) certificate tool that is available in Oracle Linux and how to use it to create Certificate Signing Requests (CSRs), self-signed certificates, and privately owned CA certificates with NSS database files which store certificates and private keys for applications.

NSS is a set of libraries designed to support cross-platform development of security-enabled client and server applications. Applications built with NSS can support SSL v2 and v3, TLS, PKCS #5, PKCS #7, PKCS #11, PKCS #12, S/MIME, X.509 v3 certificates, and other security standards.

Before you can use `certutil` to manage certificates, CSRs, and keys, you must have access to the NSS database files. You can use the legacy security databases files (cert8.db for certificates, key3.db for keys, and secmod.db for PKCS #11 module information) or the new SQLite databases files (cert9.db for certificates, key4.db for keys, and pkcs11.txt for PKCS #11 modules). This section provides examples from the new database files.

You can also use the related `pk12util` command to export and import certificates and keys from a PKCS #12 file to an NSS database or the reverse.

To use `certutil` and `pk12util`, install the nss-tools package avaiable from the Application Stream repository.

```
sudo dnf install nss-tools
```

> **Note:**
>
> If you are using Oracle Linux 7, you can install the package from the `ol7_latest` yum repository by using `yum` instead of `dnf`.

The following examples show how to use the `certutil` and and `pk12util` commands.

- To create an NSS database, do the following:

  ```
  certutil -N -d database_directory
  ```

  In the previous example, `database_directory` is the home directory where you want to create the the cert9.db, key4.db, and pkcs11.txt NSS database files. For example the following creates the database in a folder called nssdb in the home directory of the terminal's user account:

```
certutil -N -d ~/nssdb
```

- To generate a self-signed certificate, do the following:

  ```
  certutil -d database_directory -S -s subject -n nickname -x -t trust_args
  -o file
  ```

  In the previous example,

  - -S Indicates that you want to create an individual certificate and add it to a certificate database.
  - -s Indicates that you want to specify a distinguished name where *subject* uses the distinguished name format defined in https://www.rfc-editor.org/rfc/rfc1485.html.
  - -n Indicates that you want to specify a nickname where *nickname* is the nickname for the entity you are creating.
  - -x Indicates you want to generate the signature for a certificate being created or added to a database, rather than obtaining a signature from a separate CA.
  - -t Indicates you want to add trust arguments where *trust_args* are the trust attributes that you want to apply to the certificate. There are three available trust categories for each certificate, expressed in the order SSL, email, object signing for each trust setting. In each category position, use none, any, or all of the attribute codes. Valid codes are:
    - p - Valid peer
    - P - Trusted peer (includes p)
    - c - Valid CA
    - C - Trusted CA (includes c)
    - T - Trusted CA for client authentication (ssl server only)

  For example, the following creates a self-signed certificate for the www1.example.com common name with the example_test nickname. The trust attributes are C (Trusted CA) for each category.

  ```
  certutil -d ~/nssdb/ -S -s 'CN=www1.example.com, O=Example Organization,
  L=Ottawa, C=CA' -n example_test -x -t C,C,C
  ```

- To add existing certificates or certificates generated elsewhere, do the following:

  ```
  certutil -A -n nickname -t trust_args -d database_directory -i input-file
  ```

  In the previous example,

  - -A Indicates that you want to add a certificate to a certificate database.
  - -i Indicates that you want to provide an input file, such as a certificate file, for example, a PEM file.

For example,

```
certutil -A -n "CN=My SSL Certificate" -t C,C,C -d ~/nssdb/ -i ~/
tls-ca-bundle.pem
```

- To get list of all certificates, do the following:

  ```
  certutil -L -d database_directory
  ```

  For example,

  ```
  certutil -L -d ~/nssdb/

  Certificate Nickname                                         Trust
  Attributes
                                                               SSL,S/
  MIME,JAR/XPI

  example_test
  Cu,Cu,Cu
  CN=My SSL Certificate                                        C,C,C
  ```

  When listing certificates, the trust tags may include the u flag indicating that a private key is associated with the certificate.

- To delete a certificate from your database, do the following:

  ```
  certutil -D -d database_directory -n nickname
  ```

  In the previous example, -D indicates that you want to delete a specific certificate from your database.

- To get a list of all keys, do the following:

  ```
  certutil certutil -K -d database_directory
  ```

  For example,

  ```
  certutil -K -d ~/nssdb/
  certutil: Checking token "NSS Certificate DB" in slot "NSS User
  Private Key and Certificate Services"
  Enter Password or Pin for "NSS Certificate DB":
  < 0> rsa      35f4555f329c1490b3570c9d36e1ec56f2329f08   NSS
  Certificate DB:example_test
  < 1> rsa      303936d20b3522e9293b75db3dc48f77c1871767   NSS
  Certificate DB:example_test2
  ```

- To show a public key in PEM format, do the following:

  ```
  certutil -L -d database_directory -a -n nickname
  ```

  For example,

  ```
  certutil -L -d ~/nssdb/ -a -n example_test
  -----BEGIN CERTIFICATE-----
  ...[certificate text]
  -----END CERTIFICATE-----
  ```

- To export a certificate and key into a single PKCS #12 file, do the following:

  ```
  pk12util -o certs.p12 -n example_test -d sql:database_directory
  ```

- To modify a certificate, use the -M option. For example, the following changes the trust arguments from C, C, C, to P,P,P for the example_test certificate:

  ```
  $ certutil -d database_directory -M -t "P,P,P" -n example_test
  ```

For more information, see the `certutil(1)` and `pk12util(1)` manual pages and the NSS open source project at https://firefox-source-docs.mozilla.org/security/nss/index.html.

Java

Most Java applications use the keystore that's supplied with JDK to store cryptographic keys, X.509 certificate chain information, and trusted certificates. The default JDK keystore in Oracle Linux is the `/etc/pki/java/cacerts` file. You can use the `keytool` command to generate, install and manage certificates in the Java keystore.

The following examples show how you might use the `keytool` command.

- List the contents of the keystore, `/etc/pki/java/cacerts`:

  ```
  sudo keytool -list [-v] -keystore /etc/pki/java/cacerts
  ```

  The default keystore password is `changeit`. Oracle strongly recommends that you change the password as soon as possible. If specified, the verbose option `-v` displays detailed information.

- Change the password for a keystore, for example, `/etc/pki/java/cacerts`:

  ```
  sudo keytool -storepasswd -keystore /etc/pki/java/cacerts
  ```

- Create a keystore (`keystore.jks`) to achieve the following:
  - Manage public and private key pairs and certificates from entities that you trust.
  - Generate a public and private key pair by using the RSA algorithm and a key length of 3072 bits.
  - Create a self-signed certificate that includes the public key and the specified distinguished name information.

```
sudo keytool -genkeypair -alias mycert -keyalg RSA -keysize 3072 \
-dname "CN=www.unserdom.com, OU=Eng, O=Unser Dom Corp, C=US, ST=Ca,
L=Sunnydale" \
-alias engineering -keypass pkpassword -keystore keystore.jks \
-storepass storepassword -validity 100
```

In the command, *pkpassword* is the private key password and *storepassword* is the keystore password. In this example, the certificate is valid for 100 days and is associated with the private key in a keystore entry that has the alias `engineering`.

- Print the contents of a certificate file in a human-readable form:

  ```
  sudo keytool -printcert [-v] -file cert.cer
  ```

  If specified, the verbose option `-v` displays detailed information.

- Generate a CSR in the file `carequest.csr` for submission to a CA:

  ```
  sudo keytool -certreq -file carequest.csr
  ```

  The CA signs and returns a certificate or a certificate chain that authenticates your public key.

- Import the root certificate or certificate chain for the CA from the `ACME.cer` file into the `keystore.jks` keystore and assign it the alias `acmeca`:

  ```
  sudo keytool -importcert -alias acmeca [-trustcacerts] -file ACME.cer \
  -keystore keystore.jks -storepass storepassword
  ```

  If specified, the `-trustcacerts` option instructs `keytool` to add the certificate only if it can validate the chain of trust against the existing root CA certificates in the `cacerts` keystore. Alternatively, you can use the `keytool -printcert` command to check that the certificate's fingerprint matches the fingerprint that the CA publishes.

- Import the signed certificate for the organization after you have received it from the CA:

  ```
  sudo keytool -importcert -v -trustcacerts -alias acmeca -file ACMEdom.cer \
  -keystore keystore.jks -storepass storepassword
  ```

  In this example, the file containing the certificate is `ACMEdom.cer`. The `-alias` option specifies the entry for the first entity in the CA's root certificate chain. The signed certificate is added to the front of the chain and becomes the entity that's addressed by the alias name.

- Delete the certificate with the alias *aliasname* from the `keystore.jks` keystore:

  ```
  sudo keytool -delete -alias aliasname -keystore keystore.jks -storepass
  storepassword
  ```

- Export the certificate with the alias *aliasname* as a binary PKCS7 format file, which includes the supporting certificate chain as well as the issued certificate:

  ```
  sudo keytool -exportcert -noprompt -alias aliasname -file output.p7b \
  -keystore keystore.jks -storepass storepassword
  ```

- Export the certificate with the alias *aliasname* as a base64 encoded text file (also referred to as PEM or RFC 1421).

```
sudo keytool -exportcert -noprompt -rfc -alias aliasname -file output.pem \
-keystore keystore.jks -storepass storepassword
```

For a certificate chain, the file includes only the first certificate in the chain, which authenticates the public key of the aliased entity.

For more information, see the `keytool(1)` manual page.

4

Managing System Certificates

Oracle Linux stores certificates that are trusted at a system-wide level within the `/etc/pki/ca-trust/` and `/usr/share/pki/ca-trust-source/` directories. Typically, the CA certificates of major third-party CAs are included within the system-wide trust store to enable applications to work correctly. By storing trusted certificates in a central location, a wide range of applications can use these trusted certificates to validate and authenticate certificate chains. For example, when an application needs to validate a certificate, it uses the certificates within the system-wide trust to confirm whether the certificate either matches a trusted certificate, or is signed by one.

A certificate, such as a CA certificate, that's stored on a system as a trusted certificate is often referred to as a trust anchor. This distinguishes the certificate from one for which trust is derived, typically by walking through a certificate chain until a trust anchor is found. You can add any public certificate to the system trust as a trust anchor so that it can be validated immediately.

Commonly trusted third-party CA certificates are selected by the Mozilla Foundation and are included in the `ca-certificates` package. These certificates are installed into the system trust store as anchors for general use.

Using the trust Command To Manage System Certificates

The `trust` command can simplify system certificate management. This command is available in the `p11-kit-trust` package and is installed by default on most Oracle Linux systems.

See the `trust(1)` manual page for more information.

Listing Certificates in the System Trust

To list certificates that are currently trusted, run:

```
trust list
```

Output similar to the following is displayed:

```
pkcs11:id=%37%7F%3E%3E%99%71%60%CA%24%D4%91%13%79%D0%74%29%B4%A8%24%D8;type=cert
    type: certificate
    label: A-CERT ADVANCED
    trust: anchor
    category: authority

pkcs11:id=%4B%3C%8C%1D%85%E9%6F%AD;type=cert
    type: certificate
    label: A-Trust-Qual-01
    trust: anchor
    category: authority
...
```

Note that each certificate in the system trust is allocated a `pkcs11:id` value that can be used to identify a particular certificate for other trust operations.

Adding a Certificate as a Trust Anchor

To add a certificate to the system trust anchors, run:

```
sudo trust anchor /path/to/public.cert
```

Substitute */path/to/public.cert* with the path to the certificate file that you wish to add to the system trust.

When you run this command, the certificate is added to the `/etc/pki/ca-trust/source/` directory and the system trust is refreshed. The certificate is immediately trusted as an anchor.

Typically, you only add certificates from providers that you trust and which aren't already available in the system trust. You can also add self-signed certificates that you might generate for demonstration purposes or for particular internal or developer tooling.

Removing a Certificate From the System Trust Anchors

To remove a certificate from the system trust anchors, run:

```
sudo trust anchor --remove pkcs11:id=<ID>
```

Use the matching `pkcs11:id` value to provide the *<ID>* of the certificate that you want to remove. Alternately, if you have a copy of the certificate available, run:

```
sudo trust anchor --remove /path/to/public.cert
```

The system trust store is updated immediately.

Manually Updating Trusted Certificates

You can manually add a certificate to the system trust store by copying the certificate to either the `/usr/share/pki/ca-trust-source/anchors/` or `/etc/pki/ca-trust/source/anchors/` directories. This operation doesn't have immediate effect and you must run the `update-ca-trust` command to refresh the system trust store after you make manual updates to these directories.

For example:

```
sudo cp /path/to/public.cert /etc/pki/ca-trust/source/anchors
sudo update-ca-trust
```

See the `update-ca-trust(8)` manual page for more information.