Red Hat Certificate System 9

Planning, Installation, and Deployment Guide

Updated for Red Hat Certificate System 9.4

Last Updated: 2018-10-26

Red Hat Certificate System 9 Planning, Installation, and Deployment Guide

Updated for Red Hat Certificate System 9.4

Marc Muehlfeld
Red Hat Customer Content Services
mmuehlfeld@redhat.com

Petr Bokoč
Red Hat Customer Content Services

Filip Hanzelka
Red Hat Customer Content Services

Tomáš Čapek
Red Hat Customer Content Services

Aneta Petrová
Red Hat Customer Content Services

Ella Deon Ballard
Red Hat Customer Content Services

Legal Notice

Abstract

This guide covers the major PKI (Public Key Infrastructure) concepts and decision areas for planning a PKI infrastructure.

Table of Contents

PART I. PLANNING HOW TO DEPLOY RED HAT CERTIFICATE SYSTEM

This section provides an overview of Certificate System, including general PKI principles and specific features of Certificate System and its subsystems. Planning a deployment is vital to designing a PKI infrastructure that adequately meets the needs of your organization.

CHAPTER 1. INTRODUCTION TO PUBLIC-KEY CRYPTOGRAPHY

Public-key cryptography and related standards underlie the security features of many products such as signed and encrypted email, single sign-on, and Transport Layer Security/Secure Sockets Layer (SSL/TLS) communications. This chapter covers the basic concepts of public-key cryptography.

Internet traffic, which passes information through intermediate computers, can be intercepted by a third party:

Eavesdropping

Information remains intact, but its privacy is compromised. For example, someone could gather credit card numbers, record a sensitive conversation, or intercept classified information.

Tampering

Information in transit is changed or replaced and then sent to the recipient. For example, someone could alter an order for goods or change a person's resume.

Impersonation

Information passes to a person who poses as the intended recipient. Impersonation can take two forms:

- *Spoofing.* A person can pretend to be someone else. For example, a person can pretend to have the email address **jdoe@example.net** or a computer can falsely identify itself as a site called **www.example.net**.
- *Misrepresentation.* A person or organization can misrepresent itself. For example, a site called **www.example.net** can purport to be an on-line furniture store when it really receives credit-card payments but never sends any goods.

Public-key cryptography provides protection against Internet-based attacks through:

Encryption and decryption

Encryption and decryption allow two communicating parties to disguise information they send to each other. The sender encrypts, or scrambles, information before sending it. The receiver decrypts, or unscrambles, the information after receiving it. While in transit, the encrypted information is unintelligible to an intruder.

Tamper detection

Tamper detection allows the recipient of information to verify that it has not been modified in transit. Any attempts to modify or substitute data are detected.

Authentication

Authentication allows the recipient of information to determine its origin by confirming the sender's identity.

Nonrepudiation

Nonrepudiation prevents the sender of information from claiming at a later date that the information was never sent.

1.1. ENCRYPTION AND DECRYPTION

Encryption is the process of transforming information so it is unintelligible to anyone but the intended recipient. *Decryption* is the process of decoding encrypted information. A cryptographic algorithm, also called a *cipher*, is a mathematical function used for encryption or decryption. Usually, two related functions are used, one for encryption and the other for decryption.

With most modern cryptography, the ability to keep encrypted information secret is based not on the cryptographic algorithm, which is widely known, but on a number called a *key* that must be used with the algorithm to produce an encrypted result or to decrypt previously encrypted information. Decryption with the correct key is simple. Decryption without the correct key is very difficult, if not impossible.

1.1.1. Symmetric-Key Encryption

With symmetric-key encryption, the encryption key can be calculated from the decryption key and vice versa. With most symmetric algorithms, the same key is used for both encryption and decryption, as shown in Figure 1.1, “Symmetric-Key Encryption”.

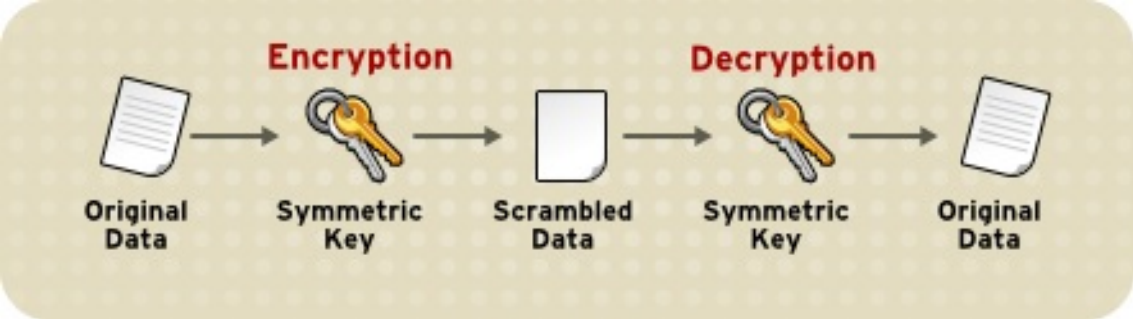


Figure 1.1. Symmetric-Key Encryption

Implementations of symmetric-key encryption can be highly efficient, so that users do not experience any significant time delay as a result of the encryption and decryption.

Symmetric-key encryption is effective only if the symmetric key is kept secret by the two parties involved. If anyone else discovers the key, it affects both confidentiality and authentication. A person with an unauthorized symmetric key not only can decrypt messages sent with that key, but can encrypt new messages and send them as if they came from one of the legitimate parties using the key.

Symmetric-key encryption plays an important role in SSL/TLS communication, which is widely used for authentication, tamper detection, and encryption over TCP/IP networks. SSL/TLS also uses techniques of public-key encryption, which is described in the next section.

1.1.2. Public-Key Encryption

Public-key encryption (also called asymmetric encryption) involves a pair of keys, a public key and a private key, associated with an entity. Each public key is published, and the corresponding private key is kept secret. (For more information about the way public keys are published, see Section 1.3, “Certificates and Authentication”.) Data encrypted with a public key can be decrypted only with the corresponding private key. Figure 1.2, “Public-Key Encryption” shows a simplified view of the way public-key encryption works.

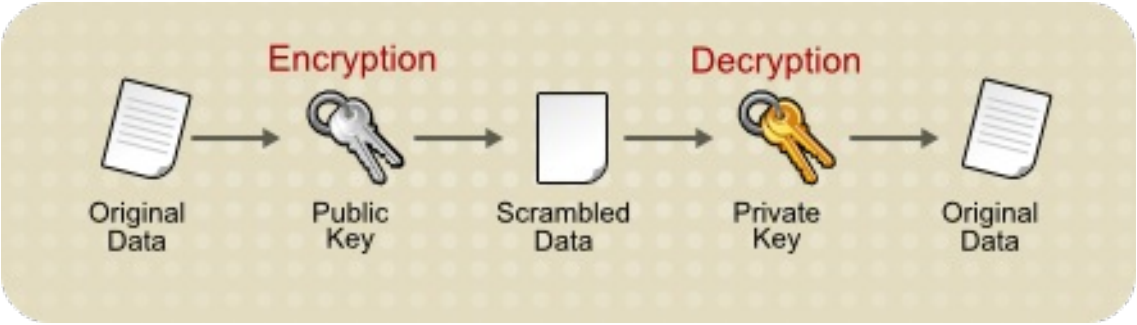


Figure 1.2. Public-Key Encryption

The scheme shown in Figure 1.2, “Public-Key Encryption” allows public keys to be freely distributed, while only authorized people are able to read data encrypted using this key. In general, to send encrypted data, the data is encrypted with that person's public key, and the person receiving the encrypted data decrypts it with the corresponding private key.

Compared with symmetric-key encryption, public-key encryption requires more processing and may not be feasible for encrypting and decrypting large amounts of data. However, it is possible to use public-key encryption to send a symmetric key, which can then be used to encrypt additional data. This is the approach used by the SSL/TLS protocols.

The reverse of the scheme shown in Figure 1.2, “Public-Key Encryption” also works: data encrypted with a private key can be decrypted only with the corresponding public key. This is not a recommended practice to encrypt sensitive data, however, because it means that anyone with the public key, which is by definition published, could decrypt the data. Nevertheless, private-key encryption is useful because it means the private key can be used to sign data with a digital signature, an important requirement for electronic commerce and other commercial applications of cryptography. Client software such as Mozilla Firefox can then use the public key to confirm that the message was signed with the appropriate private key and that it has not been tampered with since being signed. Section 1.2, “Digital Signatures” illustrates how this confirmation process works.

1.1.3. Key Length and Encryption Strength

Breaking an encryption algorithm is finding the key to the access the encrypted data in plain text. For symmetric algorithms, breaking the algorithm usually means trying to determine the key used to encrypt the text. For a public key algorithm, breaking the algorithm usually means acquiring the shared secret information between two recipients.

One method of breaking a symmetric algorithm is to simply try every key within the full algorithm until the right key is found. For public key algorithms, since half of the key pair is publicly known, the other half (private key) can be derived using published, though complex, mathematical calculations. Manually finding the key to break an algorithm is called a brute force attack.

Breaking an algorithm introduces the risk of intercepting, or even impersonating and fraudulently verifying, private information.

The *key strength* of an algorithm is determined by finding the fastest method to break the algorithm and comparing it to a brute force attack.

For symmetric keys, encryption strength is often described in terms of the size or *length* of the keys used to perform the encryption: longer keys generally provide stronger encryption. Key length is measured in bits.

An encryption key is considered full strength if the best known attack to break the key is no faster than a brute force attempt to test every key possibility.

Different types of algorithms — particularly public key algorithms — may require different key lengths to achieve the same level of encryption strength as a symmetric-key cipher. The RSA cipher can use only a subset of all possible values for a key of a given length, due to the nature of the mathematical problem on which it is based. Other ciphers, such as those used for symmetric-key encryption, can use all possible values for a key of a given length. More possible matching options means more security.

Because it is relatively trivial to break an RSA key, an RSA public-key encryption cipher must have a very long key — at least 2048 bits — to be considered cryptographically strong. On the other hand, symmetric-key ciphers are reckoned to be equivalently strong using a much shorter key length, as little as 80 bits for most algorithms. Similarly, public-key ciphers based on the elliptic curve cryptography (ECC), such as the Elliptic Curve Digital Signature Algorithm (ECDSA) ciphers, also require less bits than RSA ciphers.

1.2. DIGITAL SIGNATURES

Tamper detection relies on a mathematical function called a *one-way hash* (also called a *message digest*). A one-way hash is a number of fixed length with the following characteristics:

- The value of the hash is unique for the hashed data. Any change in the data, even deleting or altering a single character, results in a different value.
- The content of the hashed data cannot be deduced from the hash.

As mentioned in Section 1.1.2, “Public-Key Encryption”, it is possible to use a private key for encryption and the corresponding public key for decryption. Although not recommended when encrypting sensitive information, it is a crucial part of digitally signing any data. Instead of encrypting the data itself, the signing software creates a one-way hash of the data, then uses the private key to encrypt the hash. The encrypted hash, along with other information such as the hashing algorithm, is known as a digital signature.

Figure 1.3, “Using a Digital Signature to Validate Data Integrity”illustrates the way a digital signature can be used to validate the integrity of signed data.

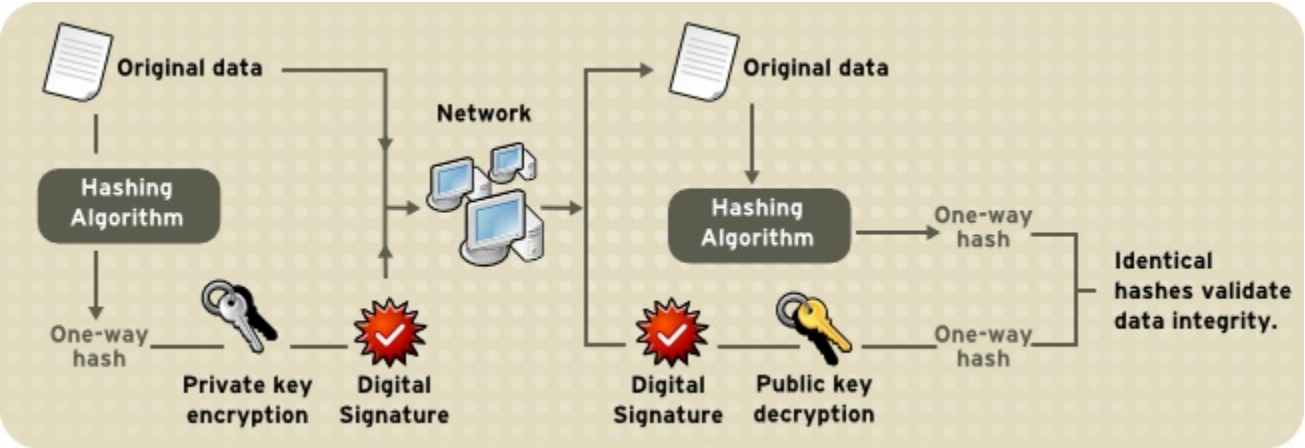


Figure 1.3. Using a Digital Signature to Validate Data Integrity

Figure 1.3, “Using a Digital Signature to Validate Data Integrity”shows two items transferred to the recipient of some signed data: the original data and the digital signature, which is a one-way hash of the original data encrypted with the signer's private key. To

validate the integrity of the data, the receiving software first uses the public key to decrypt the hash. It then uses the same hashing algorithm that generated the original hash to generate a new one-way hash of the same data. (Information about the hashing algorithm used is sent with the digital signature.) Finally, the receiving software compares the new hash against the original hash. If the two hashes match, the data has not changed since it was signed. If they do not match, the data may have been tampered with since it was signed, or the signature may have been created with a private key that does not correspond to the public key presented by the signer.

If the two hashes match, the recipient can be certain that the public key used to decrypt the digital signature corresponds to the private key used to create the digital signature. Confirming the identity of the signer also requires some way of confirming that the public key belongs to a particular entity. For more information on authenticating users, see Section 1.3, “Certificates and Authentication”.

A digital signature is similar to a handwritten signature. Once data have been signed, it is difficult to deny doing so later, assuming the private key has not been compromised. This quality of digital signatures provides a high degree of nonrepudiation; digital signatures make it difficult for the signer to deny having signed the data. In some situations, a digital signature is as legally binding as a handwritten signature.

1.3. CERTIFICATES AND AUTHENTICATION

1.3.1. A Certificate Identifies Someone or Something

A certificate is an electronic document used to identify an individual, a server, a company, or other entity and to associate that identity with a public key. Like a driver's license or passport, a certificate provides generally recognized proof of a person's identity. Public-key cryptography uses certificates to address the problem of impersonation.

To get personal ID such as a driver's license, a person has to present some other form of identification which confirms that the person is who he claims to be. Certificates work much the same way. Certificate authorities (CAs) validate identities and issue certificates. CAs can be either independent third parties or organizations running their own certificate-issuing server software, such as Certificate System. The methods used to validate an identity vary depending on the policies of a given CA for the type of certificate being requested. Before issuing a certificate, a CA must confirm the user's identity with its standard verification procedures.

The certificate issued by the CA binds a particular public key to the name of the entity the certificate identifies, such as the name of an employee or a server. Certificates help prevent the use of fake public keys for impersonation. Only the public key certified by the certificate will work with the corresponding private key possessed by the entity identified by the certificate.

In addition to a public key, a certificate always includes the name of the entity it identifies, an expiration date, the name of the CA that issued the certificate, and a serial number. Most importantly, a certificate always includes the digital signature of the issuing CA. The CA's digital signature allows the certificate to serve as a valid credential for users who know and trust the CA but do not know the entity identified by the certificate.

For more information about the role of CAs, see Section 1.3.6, “How CA Certificates Establish Trust”.

1.3.2. Authentication Confirms an Identity

Authentication is the process of confirming an identity. For network interactions, authentication involves the identification of one party by another party. There are many ways to use authentication over networks. Certificates are one of those way.

Network interactions typically take place between a client, such as a web browser, and a server. *Client authentication* refers to the identification of a client (the person assumed to be using the software) by a server. *Server authentication* refers to the identification of a server (the organization assumed to be running the server at the network address) by a client.

Client and server authentication are not the only forms of authentication that certificates support. For example, the digital signature on an email message, combined with the certificate that identifies the sender, can authenticate the sender of the message. Similarly, a digital signature on an HTML form, combined with a certificate that identifies the signer, can provide evidence that the person identified by that certificate agreed to the contents of the form. In addition to authentication, the digital signature in both cases ensures a degree of nonrepudiation; a digital signature makes it difficult for the signer to claim later not to have sent the email or the form.

Client authentication is an essential element of network security within most intranets or extranets. There are two main forms of client authentication:

Password-based authentication

Almost all server software permits client authentication by requiring a recognized name and password before granting access to the server.

Certificate-based authentication

Client authentication based on certificates is part of the SSL/TLS protocol. The client digitally signs a randomly generated piece of data and sends both the certificate and the signed data across the network. The server validates the signature and confirms the validity of the certificate.

1.3.2.1. Password-Based Authentication

Figure 1.4, “Using a Password to Authenticate a Client to a Server”shows the process of authenticating a user using a user name and password. This example assumes the following:

- The user has already trusted the server, either without authentication or on the basis of server authentication over SSL/TLS.
- The user has requested a resource controlled by the server.
- The server requires client authentication before permitting access to the requested resource.

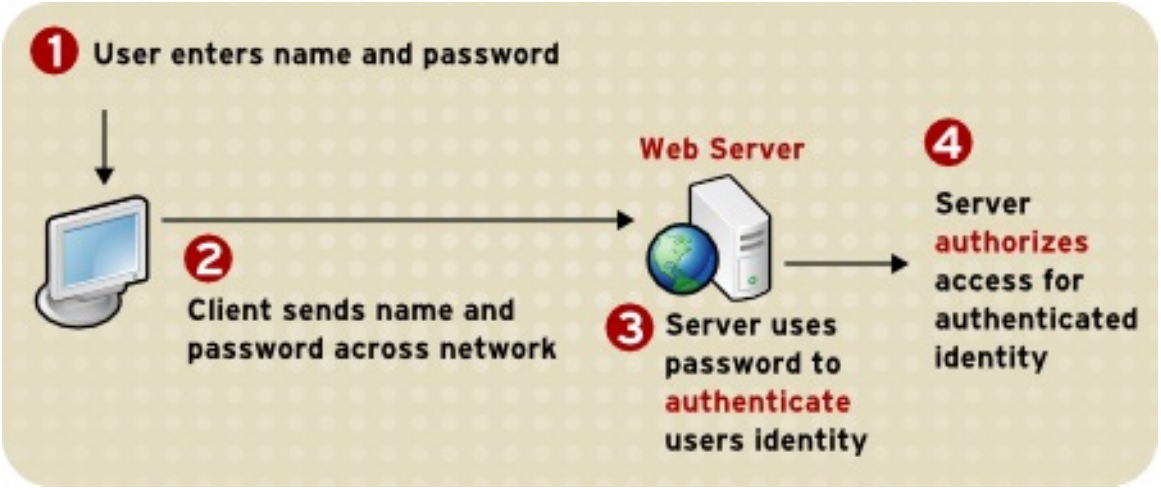


Figure 1.4. Using a Password to Authenticate a Client to a Server

These are the steps in this authentication process:

1. When the server requests authentication from the client, the client displays a dialog box requesting the user name and password for that server.

2. The client sends the name and password across the network, either in plain text or over an encrypted SSL/TLS connection.

3. The server looks up the name and password in its local password database and, if they match, accepts them as evidence authenticating the user's identity.

4. The server determines whether the identified user is permitted to access the requested resource and, if so, allows the client to access it.

With this arrangement, the user must supply a new password for each server accessed, and the administrator must keep track of the name and password for each user.

1.3.2.2. Certificate-Based Authentication

One of the advantages of certificate-based authentication is that it can be used to replace the first three steps in authentication with a mechanism that allows the user to supply one password, which is not sent across the network, and allows the administrator to control user authentication centrally. This is called *single sign-on*.

Figure 1.5, “Using a Certificate to Authenticate a Client to a Server”shows how client authentication works using certificates and SSL/TLS. To authenticate a user to a server, a client digitally signs a randomly generated piece of data and sends both the certificate and the signed data across the network. The server authenticates the user's identity based on the data in the certificate and signed data.

Like Figure 1.4, “Using a Password to Authenticate a Client to a Server”, Figure 1.5, “Using a Certificate to Authenticate a Client to a Server” assumes that the user has already trusted the server and requested a resource and that the server has requested client authentication before granting access to the requested resource.

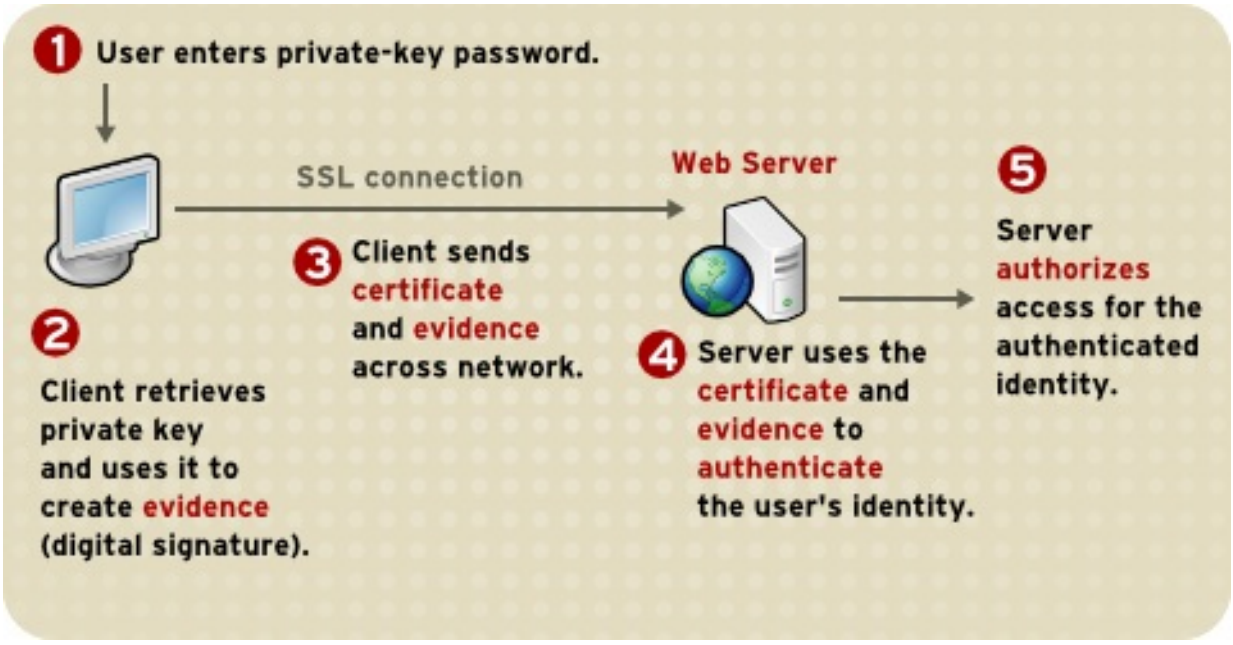


Figure 1.5. Using a Certificate to Authenticate a Client to a Server

Unlike the authentication process in Figure 1.4, “Using a Password to Authenticate a Client to a Server”, the authentication process inFigure 1.5, “Using a Certificate to Authenticate a Client to a Server” requires SSL/TLS. Figure 1.5, “Using a Certificate to Authenticate a Client to a Server” also assumes that the client has a valid certificate that can be used to identify the client to the server. Certificate-based authentication is preferred to password-based authentication because it is based on the user both possessing the private key and knowing the password. However, these two assumptions are true only if unauthorized personnel have not gained access to the user's machine or password, the password for the client software's private key database has been set, and the software is set up to request the password at reasonably frequent intervals.

NOTE

Neither password-based authentication nor certificate-based authentication address security issues related to physical access to individual machines or passwords. Public-key cryptography can only verify that a private key used to sign some data corresponds to the public key in a certificate. It is the user's responsibility to protect a machine's physical security and to keep the private-key password secret.

These are the authentication steps shown in Figure 1.5, “Using a Certificate to Authenticate a Client to a Server”:

1. The client software maintains a database of the private keys that correspond to the public keys published in any certificates issued for that client. The client asks for the password to this database the first time the client needs to access it during a given session, such as the first time the user attempts to access an SSL/TLS-enabled server that requires certificate-based client authentication.

   After entering this password once, the user does not need to enter it again for the rest of the session, even when accessing other SSL/TLS-enabled servers.

2. The client unlocks the private-key database, retrieves the private key for the user's certificate, and uses that private key to sign data randomly-generated from input from both the client and the server. This data and the digital signature are evidence

of the private key's validity. The digital signature can be created only with that private key and can be validated with the corresponding public key against the signed data, which is unique to the SSL/TLS session.

3. The client sends both the user's certificate and the randomly-generated data across the network.

4. The server uses the certificate and the signed data to authenticate the user's identity.

5. The server may perform other authentication tasks, such as checking that the certificate presented by the client is stored in the user's entry in an LDAP directory. The server then evaluates whether the identified user is permitted to access the requested resource. This evaluation process can employ a variety of standard authorization mechanisms, potentially using additional information in an LDAP directory or company databases. If the result of the evaluation is positive, the server allows the client to access the requested resource.

Certificates replace the authentication portion of the interaction between the client and the server. Instead of requiring a user to send passwords across the network continually, single sign-on requires the user to enter the private-key database password once, without sending it across the network. For the rest of the session, the client presents the user's certificate to authenticate the user to each new server it encounters. Existing authorization mechanisms based on the authenticated user identity are not affected.

1.3.3. Uses for Certificates

The purpose of certificates is to establish trust. Their usage varies depending on the kind of trust they are used to ensure. Some kinds of certificates are used to verify the identity of the presenter; others are used to verify that an object or item has not been tampered with.

1.3.3.1. SSL/TLS

The Transport Layer Security/Secure Sockets Layer (SSL/TLS) protocol governs server authentication, client authentication, and encrypted communication between servers and clients. SSL/TLS is widely used on the Internet, especially for interactions that involve exchanging confidential information such as credit card numbers.

SSL/TLS requires an SSL/TLS server certificate. As part of the initial SSL/TLS handshake, the server presents its certificate to the client to authenticate the server's identity. The authentication uses public-key encryption and digital signatures to confirm that the server is the server it claims to be. Once the server has been authenticated, the client and server use symmetric-key encryption, which is very fast, to encrypt all the information exchanged for the remainder of the session and to detect any tampering.

Servers may be configured to require client authentication as well as server authentication. In this case, after server authentication is successfully completed, the client must also present its certificate to the server to authenticate the client's identity before the encrypted SSL/TLS session can be established.

For an overview of client authentication over SSL/TLS and how it differs from password-based authentication, see Section 1.3.2, “Authentication Confirms an Identity”.

1.3.3.2. Signed and Encrypted Email

Some email programs support digitally signed and encrypted email using a widely accepted

protocol known as Secure Multipurpose Internet Mail Extension (S/MIME). Using S/MIME to sign or encrypt email messages requires the sender of the message to have an S/MIME certificate.

An email message that includes a digital signature provides some assurance that it was sent by the person whose name appears in the message header, thus authenticating the sender. If the digital signature cannot be validated by the email software, the user is alerted.

The digital signature is unique to the message it accompanies. If the message received differs in any way from the message that was sent, even by adding or deleting a single character, the digital signature cannot be validated. Therefore, signed email also provides assurance that the email has not been tampered with. This kind of assurance is known as nonrepudiation, which makes it difficult for the sender to deny having sent the message. This is important for business communication. For information about the way digital signatures work, see Section 1.2, “Digital Signatures”.

S/MIME also makes it possible to encrypt email messages, which is important for some business users. However, using encryption for email requires careful planning. If the recipient of encrypted email messages loses the private key and does not have access to a backup copy of the key, the encrypted messages can never be decrypted.

1.3.3.3. Single Sign-on

Network users are frequently required to remember multiple passwords for the various services they use. For example, a user might have to type a different password to log into the network, collect email, use directory services, use the corporate calendar program, and access various servers. Multiple passwords are an ongoing headache for both users and system administrators. Users have difficulty keeping track of different passwords, tend to choose poor ones, and tend to write them down in obvious places. Administrators must keep track of a separate password database on each server and deal with potential security problems related to the fact that passwords are sent over the network routinely and frequently.

Solving this problem requires some way for a user to log in once, using a single password, and get authenticated access to all network resources that user is authorized to use-without sending any passwords over the network. This capability is known as single sign-on.

Both client SSL/TLS certificates and S/MIME certificates can play a significant role in a comprehensive single sign-on solution. For example, one form of single sign-on supported by Red Hat products relies on SSL/TLS client authentication. A user can log in once, using a single password to the local client's private-key database, and get authenticated access to all SSL/TLS-enabled servers that user is authorized to use-without sending any passwords over the network. This approach simplifies access for users, because they do not need to enter passwords for each new server. It also simplifies network management, since administrators can control access by controlling lists of certificate authorities (CAs) rather than much longer lists of users and passwords.

In addition to using certificates, a complete single-sign on solution must address the need to interoperate with enterprise systems, such as the underlying operating system, that rely on passwords or other forms of authentication.

1.3.3.4. Object Signing

Many software technologies support a set of tools called *object signing*. Object signing uses standard techniques of public-key cryptography to let users get reliable information about

code they download in much the same way they can get reliable information about shrink-wrapped software.

Most important, object signing helps users and network administrators implement decisions about software distributed over intranets or the Internet-for example, whether to allow Java applets signed by a given entity to use specific computer capabilities on specific users' machines.

The objects signed with object signing technology can be applets or other Java code, JavaScript scripts, plug-ins, or any kind of file. The signature is a digital signature. Signed objects and their signatures are typically stored in a special file called a JAR file.

Software developers and others who wish to sign files using object-signing technology must first obtain an object-signing certificate.

1.3.4. Types of Certificates

The Certificate System is capable of generating different types of certificates for different uses and in different formats. Planning which certificates are required and planning how to manage them, including determining what formats are needed and how to plan for renewal, are important to manage both the PKI and the Certificate System instances.

This list is not exhaustive; there are certificate enrollment forms for dual-use certificates for LDAP directories, file-signing certificates, and other subsystem certificates. These forms are available through the Certificate Manager's end-entities page, at **`https://server.example.com:8443/ca/ee/ca`**.

When the different Certificate System subsystems are installed, the basic required certificates and keys are generated; for example, configuring the Certificate Manager generates the CA signing certificate for the self-signed root CA and the internal OCSP signing, audit signing, SSL/TLS server, and agent user certificates. During the KRA configuration, the Certificate Manager generates the storage, transport, audit signing, and agent certificates. Additional certificates can be created and installed separately.

Table 1.1. Common Certificates

Certificate Type	Use	Example
Client SSL/TLS certificates	Used for client authentication to servers over SSL/TLS. Typically, the identity of the client is assumed to be the same as the identity of a person, such as an employee. See Section 1.3.2.2, “Certificate-Based Authentication” for a description of the way SSL/TLS client certificates are used for client authentication. Client SSL/TLS certificates can also be used as part of single sign-on.	A bank gives a customer an SSL/TLS client certificate that allows the bank's servers to identify that customer and authorize access to the customer's accounts. A company gives a new employee an SSL/TLS client certificate that allows the company's servers to identify that employee and authorize access to the company's servers.

Certificate Type	Use	Example
Server SSL/TLS certificates	Used for server authentication to clients over SSL/TLS. Server authentication may be used without client authentication. Server authentication is required for an encrypted SSL/TLS session. For more information, see Section 1.3.3.1, “SSL/TLS”.	Internet sites that engage in electronic commerce usually support certificate-based server authentication to establish an encrypted SSL/TLS session and to assure customers that they are dealing with the web site identified with the company. The encrypted SSL/TLS session ensures that personal information sent over the network, such as credit card numbers, cannot easily be intercepted.
S/MIME certificates	Used for signed and encrypted email. As with SSL/TLS client certificates, the identity of the client is assumed to be the same as the identity of a person, such as an employee. A single certificate may be used as both an S/MIME certificate and an SSL/TLS certificate; see Section 1.3.3.2, “Signed and Encrypted Email”. S/MIME certificates can also be used as part of single sign-on.	A company deploys combined S/MIME and SSL/TLS certificates solely to authenticate employee identities, thus permitting signed email and SSL/TLS client authentication but not encrypted email. Another company issues S/MIME certificates solely to sign and encrypt email that deals with sensitive financial or legal matters.
CA certificates	Used to identify CAs. Client and server software use CA certificates to determine what other certificates can be trusted. For more information, see Section 1.3.6, “How CA Certificates Establish Trust”.	The CA certificates stored in Mozilla Firefox determine what other certificates can be authenticated. An administrator can implement corporate security policies by controlling the CA certificates stored in each user's copy of Firefox.
Object-signing certificates	Used to identify signers of Java code, JavaScript scripts, or other signed files.	Software companies frequently sign software distributed over the Internet to provide users with some assurance that the software is a legitimate product of that company. Using certificates and digital signatures can also make it possible for users to identify and control the kind of access downloaded software has to their computers.

1.3.4.1. CA Signing Certificates

Every Certificate Manager has a CA signing certificate with a public/private key pair it uses to sign the certificates and certificate revocation lists (CRLs) it issues. This certificate is created and installed when the Certificate Manager is installed.

NOTE

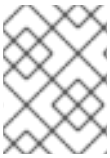
For more information about CRLs, see Section 2.4.4, “Revoking Certificates and Checking Status”.

The Certificate Manager's status as a root or subordinate CA is determined by whether its CA signing certificate is self-signed or is signed by another CA. Self-signed root CAs set the policies they use to issue certificates, such as the subject names, types of certificates that can be issued, and to whom certificates can be issued. A subordinate CA has a CA signing certificate signed by another CA, usually the one that is a level above in the CA hierarchy (which may or may not be a root CA). If the Certificate Manager is a subordinate CA in a CA hierarchy, the root CA's signing certificate must be imported into individual clients and servers before the Certificate Manager can be used to issue certificates to them.

The CA certificate must be installed in a client if a server or user certificate issued by that CA is installed on that client. The CA certificate confirms that the server certificate can be trusted. Ideally, the certificate chain is installed.

1.3.4.2. Other Signing Certificates

Other services, such as the Online Certificate Status Protocol (OCSP) responder service and CRL publishing, can use signing certificates other than the CA certificate. For example, a separate CRL signing certificate can be used to sign the revocation lists that are published by a CA instead of using the CA signing certificate.

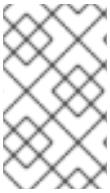


NOTE

For more information about OCSP, see Section 2.4.4, “Revoking Certificates and Checking Status”.

1.3.4.3. SSL/TLS Server and Client Certificates

Server certificates are used for secure communications, such as SSL/TLS, and other secure functions. Server certificates are used to authenticate themselves during operations and to encrypt data; client certificates authenticate the client to the server.



NOTE

CAs which have a signing certificate issued by a third-party may not be able to issue server certificates. The third-party CA may have rules in place which prohibit its subordinates from issuing server certificates.

1.3.4.4. User Certificates

End user certificates are a subset of client certificates that are used to identify users to a server or system. Users can be assigned certificates to use for secure communications, such as SSL/TLS, and other functions such as encrypting email or for single sign-on. Special users, such as Certificate System agents, can be given client certificates to access special services.

1.3.4.5. Dual-Key Pairs

Dual-key pairs are a set of two private and public keys, where one set is used for signing and one for encryption. These dual keys are used to create dual certificates. The dual certificate enrollment form is one of the standard forms listed in the end-entities page of the Certificate Manager.

When generating dual-key pairs, set the certificate profiles to work correctly when generating separate certificates for signing and encryption.

1.3.4.6. Cross-Pair Certificates

The Certificate System can issue, import, and publish cross-pair CA certificates. With cross-pair certificates, one CA signs and issues a cross-pair certificate to a second CA, and the second CA signs and issues a cross-pair certificate to the first CA. Both CAs then store or publish both certificates as a **crossCertificatePair** entry.

Bridging certificates can be done to honor certificates issued by a CA that is not chained to the root CA. By establishing a trust between the Certificate System CA and another CA through a cross-pair CA certificate, the cross-pair certificate can be downloaded and used to trust the certificates issued by the other CA.

1.3.5. Contents of a Certificate

The contents of certificates are organized according to the X.509 v3 certificate specification, which has been recommended by the International Telecommunications Union (ITU), an international standards body.

Users do not usually need to be concerned about the exact contents of a certificate. However, system administrators working with certificates may need some familiarity with the information contained in them.

1.3.5.1. Certificate Data Formats

Certificate requests and certificates can be created, stored, and installed in several different formats. All of these formats conform to X.509 standards.

1.3.5.1.1. Binary

The following binary formats are recognized:

- *DER-encoded certificate*. This is a single binary DER-encoded certificate.
- *PKCS #7 certificate chain*. This is a PKCS #7 **SignedData** object. The only significant field in the **SignedData** object is the certificates; the signature and the contents, for example, are ignored. The PKCS #7 format allows multiple certificates to be downloaded at a single time.
- *Netscape Certificate Sequence*. This is a simpler format for downloading certificate chains in a PKCS #7 **ContentInfo** structure, wrapping a sequence of certificates. The value of the **contentType** field should be **netscape-cert-sequence**, while the content field has the following structure:

```
CertificateSequence ::= SEQUENCE OF Certificate
```

This format allows multiple certificates to be downloaded at the same time.

1.3.5.1.2. Text

Any of the binary formats can be imported in text form. The text form begins with the following line:

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

Following this line is the certificate data, which can be in any of the binary formats described. This data should be base-64 encoded, as described by RFC 1113. The certificate information is followed by this line:

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

1.3.5.2. Distinguished Names

An X.509 v3 certificate binds a distinguished name (DN) to a public key. A DN is a series of name-value pairs, such as **uid=doe**, that uniquely identify an entity. This is also called the certificate *subject name*.

This is an example DN of an employee for Example Corp.:

```
uid=doe, cn=John Doe,o=Example Corp.,c=US
```

In this DN, **uid** is the user name, **cn** is the user's common name, **o** is the organization or company name, and **c** is the country.

DNs may include a variety of other name-value pairs. They are used to identify both certificate subjects and entries in directories that support the Lightweight Directory Access Protocol (LDAP).

The rules governing the construction of DNs can be complex; for comprehensive information about DNs, see *A String Representation of Distinguished Names* at http://www.ietf.org/rfc/rfc4514.txt.

1.3.5.3. A Typical Certificate

Every X.509 certificate consists of two sections:

The data section

This section includes the following information:

- The version number of the X.509 standard supported by the certificate.
- The certificate's serial number. Every certificate issued by a CA has a serial number that is unique among the certificates issued by that CA.
- Information about the user's public key, including the algorithm used and a representation of the key itself.
- The DN of the CA that issued the certificate.

- The period during which the certificate is valid; for example, between 1:00 p.m. on November 15, 2004, and 1:00 p.m. November 15, 2018.
- The DN of the certificate subject, which is also called the subject name; for example, in an SSL/TLS client certificate, this is the user's DN.
- Optional *certificate extensions*, which may provide additional data used by the client or server. For example:
  - the Netscape Certificate Type extension indicates the type of certificate, such as an SSL/TLS client certificate, an SSL/TLS server certificate, or a certificate for signing email
  - the Subject Alternative Name (SAN) extension links a certificate to one or more host names

  Certificate extensions can also be used for other purposes.

The signature section

This section includes the following information:

- The cryptographic algorithm, or cipher, used by the issuing CA to create its own digital signature.
- The CA's digital signature, obtained by hashing all of the data in the certificate together and encrypting it with the CA's private key.

Here are the data and signature sections of a certificate shown in the readable pretty-print format:

```
Certificate:
Data:
   Version: v3 (0x2)
   Serial Number: 3 (0x3)
   Signature Algorithm: PKCS #1 MD5 With RSA Encryption
   Issuer: OU=Example Certificate Authority, O=Example Corp, C=US
   Validity:
    Not Before: Fri Oct 17 18:36:25 1997
    Not  After: Sun Oct 17 18:36:25 1999
   Subject: CN=Jane Doe, OU=Finance, O=Example Corp, C=US
   Subject Public Key Info:
    Algorithm: PKCS #1 RSA Encryption
    Public Key:
       Modulus:
          00:ca:fa:79:98:8f:19:f8:d7:de:e4:49:80:48:e6:2a:2a:86:
          ed:27:40:4d:86:b3:05:c0:01:bb:50:15:c9:de:dc:85:19:22:
          43:7d:45:6d:71:4e:17:3d:f0:36:4b:5b:7f:a8:51:a3:a1:00:
          98:ce:7f:47:50:2c:93:36:7c:01:6e:cb:89:06:41:72:b5:e9:
          73:49:38:76:ef:b6:8f:ac:49:bb:63:0f:9b:ff:16:2a:e3:0e:
          9d:3b:af:ce:9a:3e:48:65:de:96:61:d5:0a:11:2a:a2:80:b0:
          7d:d8:99:cb:0c:99:34:c9:ab:25:06:a8:31:ad:8c:4b:aa:54:
          91:f4:15
       Public Exponent: 65537 (0x10001)
   Extensions:
    Identifier: Certificate Type
```

```
      Critical: no
      Certified Usage:
      TLS Client
    Identifier: Authority Key Identifier
      Critical: no
      Key Identifier:
        f2:f2:06:59:90:18:47:51:f5:89:33:5a:31:7a:e6:5c:fb:36:
        26:c9
  Signature:
   Algorithm: PKCS #1 MD5 With RSA Encryption
  Signature:
6d:23:af:f3:d3:b6:7a:df:90:df:cd:7e:18:6c:01:69:8e:54:65:fc:06:
30:43:34:d1:63:1f:06:7d:c3:40:a8:2a:82:c1:a4:83:2a:fb:2e:8f:fb:
f0:6d:ff:75:a3:78:f7:52:47:46:62:97:1d:d9:c6:11:0a:02:a2:e0:cc:
2a:75:6c:8b:b6:9b:87:00:7d:7c:84:76:79:ba:f8:b4:d2:62:58:c3:c5:
b6:c1:43:ac:63:44:42:fd:af:c8:0f:2f:38:85:6d:d6:59:e8:41:42:a5:
4a:e5:26:38:ff:32:78:a1:38:f1:ed:dc:0d:31:d1:b0:6d:67:e9:46:a8:
d:c4
```

Here is the same certificate in the base-64 encoded format:

```
-----BEGIN CERTIFICATE-----
MIICKzCCAZSgAwIBAgIBAzANBgkqhkiG9w0BAQQFADA3MQswCQYDVQQGEwJVUzER
MA8GA1UEChMITmV0c2NhcGUxFTATBgNVBAsTDFN1cHJpeWEncyBDQTAeFw05NzEw
MTgwMTM2MjVaFw05OTEwMTgwMTM2MjVaMEgxCzAJBgNVBAYTAlVTMREwDwYDVQQK
EwhOZXRzY2FwZTENMAsGA1UECxMEUHViczEXMBUGA1UEAxMOU3Vwcml5YSBTaGV0
dHkwgZ8wDQYJKoZIhvcNAQEFBQADgY0AMIGJAoGBAMr6eZiPGfjX3uRJgEjmKiqG
7SdATYazBcABu1AVyd7chRkiQ31FbXFOGD3wNktbf6hRo6EAmM5/R1AskzZ8AW7L
iQZBcrXpc0k4du+2Q6xJu2MPm/8WKuMOnTuvzpo+SGXelmHVChEqooCwfdiZywyZ
NMmrJgaoMa2MS6pUkfQVAgMBAAGjNjA0MBEGCWCGSAGG+EIBAQQEAwIAgDAfBgNV
HSMEGDAWgBTy8gZZkBhHUfWJM1oxeuZc+zYmyTANBgkqhkiG9w0BAQQFAAOBgQBt
I6/z07Z635DfzX4XbAFpjlRl/AYwQzTSYx8GfcNAqCqCwaSDKvsuj/vwbf91o3j3
UkdGYpcd2cYRCgKi4MwqdWyLtpuHAH18hHZ5uvi00mJYw8W2wUOsY0RC/a/IDy84
hW3WWehBUqVK5SY4/zJ4oTjx7dwNMdGwbWfpRqjd1A==
-----END CERTIFICATE-----
```

1.3.6. How CA Certificates Establish Trust

CAs validate identities and issue certificates. They can be either independent third parties or organizations running their own certificate-issuing server software, such as the Certificate System.

Any client or server software that supports certificates maintains a collection of trusted CA certificates. These CA certificates determine which issuers of certificates the software can trust, or validate. In the simplest case, the software can validate only certificates issued by one of the CAs for which it has a certificate. It is also possible for a trusted CA certificate to be part of a chain of CA certificates, each issued by the CA above it in a certificate hierarchy.

The sections that follow explains how certificate hierarchies and certificate chains determine what certificates software can trust.

1.3.6.1. CA Hierarchies

In large organizations, responsibility for issuing certificates can be delegated to several

different CAs. For example, the number of certificates required may be too large for a single CA to maintain; different organizational units may have different policy requirements; or a CA may need to be physically located in the same geographic area as the people to whom it is issuing certificates.

These certificate-issuing responsibilities can be divided among subordinate CAs. The X.509 standard includes a model for setting up a hierarchy of CAs, shown in Figure 1.6, “Example of a Hierarchy of Certificate Authorities”.

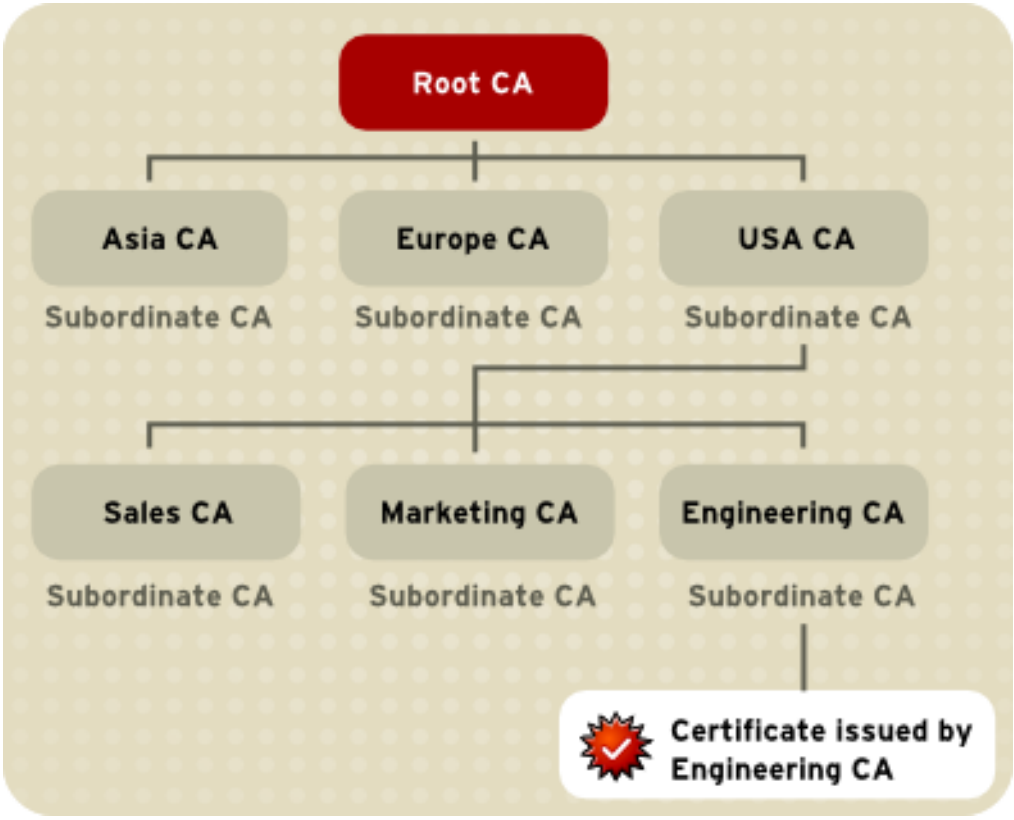


Figure 1.6. Example of a Hierarchy of Certificate Authorities

The root CA is at the top of the hierarchy. The root CA's certificate is a *self-signed certificate*; that is, the certificate is digitally signed by the same entity that the certificate identifies. The CAs that are directly subordinate to the root CA have CA certificates signed by the root CA. CAs under the subordinate CAs in the hierarchy have their CA certificates signed by the higher-level subordinate CAs.

Organizations have a great deal of flexibility in how CA hierarchies are set up; Figure 1.6, “Example of a Hierarchy of Certificate Authorities” shows just one example.

1.3.6.2. Certificate Chains

CA hierarchies are reflected in certificate chains. A *certificate chain* is series of certificates issued by successive CAs. Figure 1.7, “Example of a Certificate Chain” shows a certificate chain leading from a certificate that identifies an entity through two subordinate CA certificates to the CA certificate for the root CA, based on the CA hierarchy shown in Figure 1.6, “Example of a Hierarchy of Certificate Authorities”.

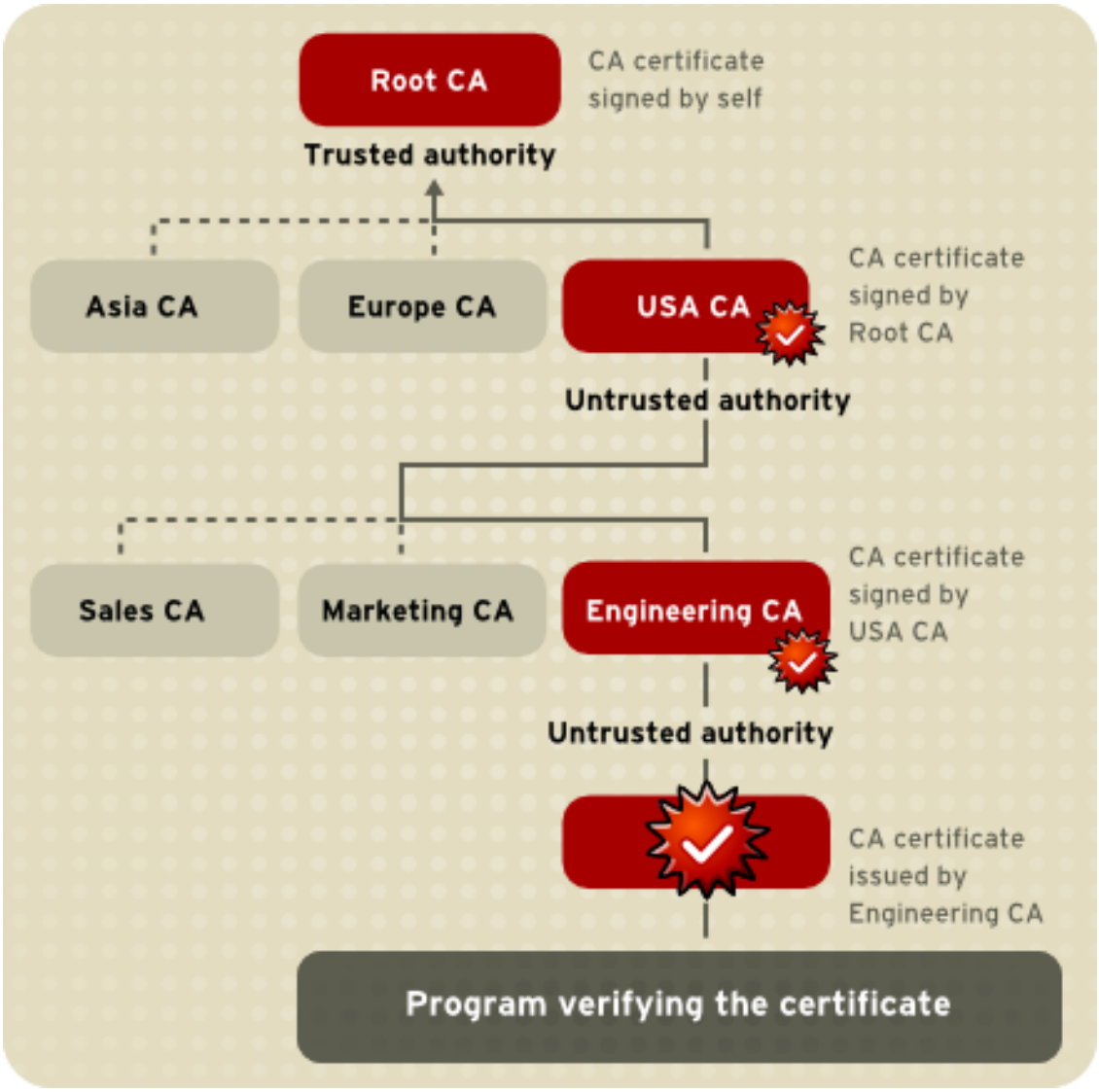


Figure 1.7. Example of a Certificate Chain

A certificate chain traces a path of certificates from a branch in the hierarchy to the root of the hierarchy. In a certificate chain, the following occur:

- Each certificate is followed by the certificate of its issuer.
- Each certificate contains the name (DN) of that certificate's issuer, which is the same as the subject name of the next certificate in the chain.

  In Figure 1.7, “Example of a Certificate Chain”, the **`Engineering CA`** certificate contains the DN of the CA, **`USA CA`**, that issued that certificate.**`USA CA`**'s DN is also the subject name of the next certificate in the chain.
- Each certificate is signed with the private key of its issuer. The signature can be verified with the public key in the issuer's certificate, which is the next certificate in the chain.

  In Figure 1.7, “Example of a Certificate Chain”, the public key in the certificate for the **`USA CA`** can be used to verify the**`USA CA`**'s digital signature on the certificate for the **`Engineering CA`**.

1.3.6.3. Verifying a Certificate Chain

Certificate chain verification makes sure a given certificate chain is well-formed, valid, properly signed, and trustworthy. The following description of the process covers the most important steps of forming and verifying a certificate chain, starting with the certificate being presented for authentication:

1. The certificate validity period is checked against the current time provided by the verifier's system clock.

2. The issuer's certificate is located. The source can be either the verifier's local certificate database on that client or server or the certificate chain provided by the subject, as with an SSL/TLS connection.

3. The certificate signature is verified using the public key in the issuer's certificate.

4. The host name of the service is compared against the Subject Alternative Name (SAN) extension. If the certificate has no such extension, the host name is compared against the subject's CN.

5. The system verifies the Basic Constraint requirements for the certificate, that is, whether the certificate is a CA and how many subsidiaries it is allowed to sign.

6. If the issuer's certificate is trusted by the verifier in the verifier's certificate database, verification stops successfully here. Otherwise, the issuer's certificate is checked to make sure it contains the appropriate subordinate CA indication in the certificate type extension, and chain verification starts over with this new certificate. Figure 1.8, “Verifying a Certificate Chain to the Root CA”presents an example of this process.

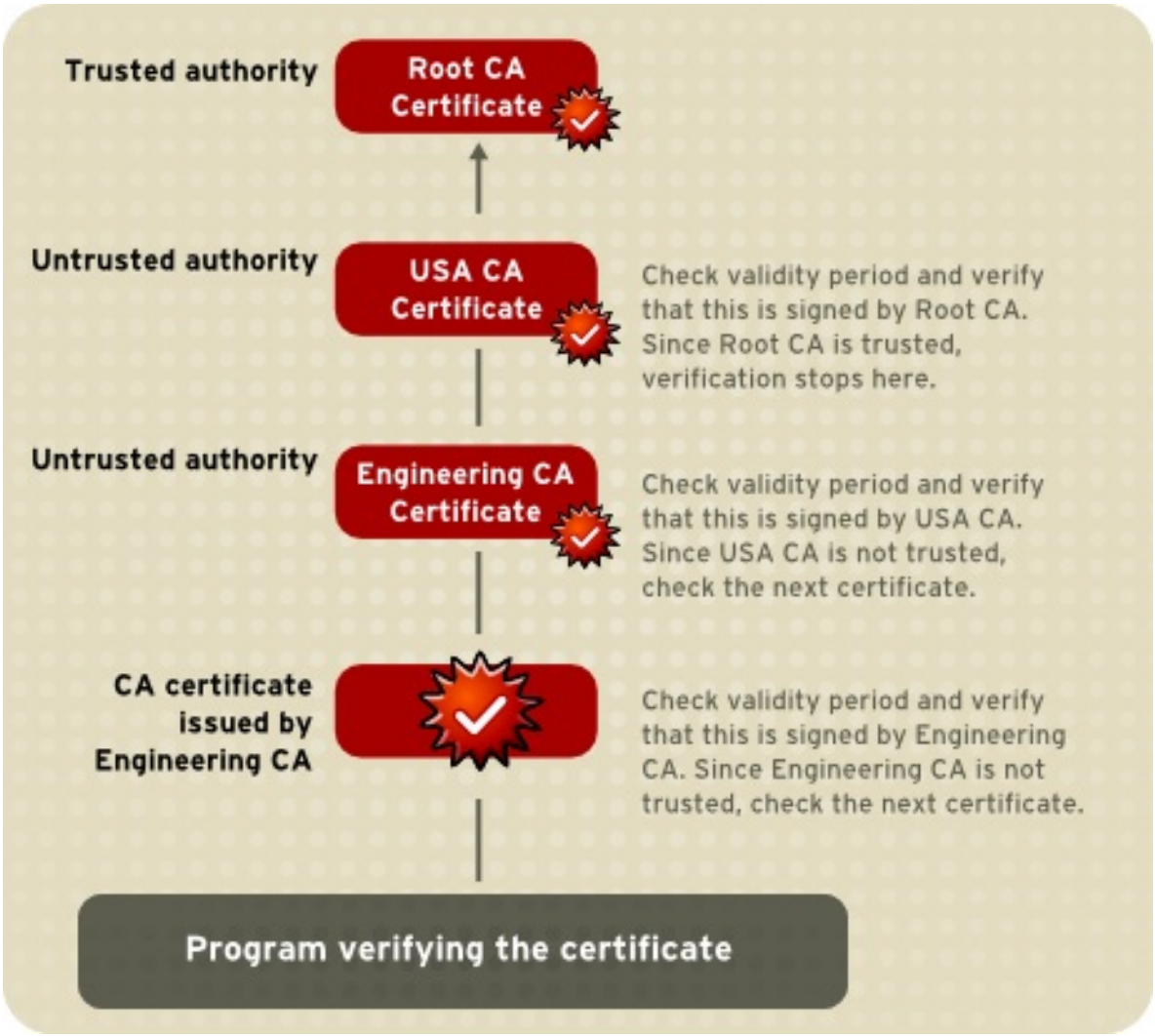


Figure 1.8. Verifying a Certificate Chain to the Root CA

Figure 1.8, “Verifying a Certificate Chain to the Root CA”illustrates what happens when only the root CA is included in the verifier's local database. If a certificate for one of the intermediate CAs, such as **`Engineering CA`**, is found in the verifier's local database, verification stops with that certificate, as shown in Figure 1.9, “Verifying a Certificate Chain to an Intermediate CA”.

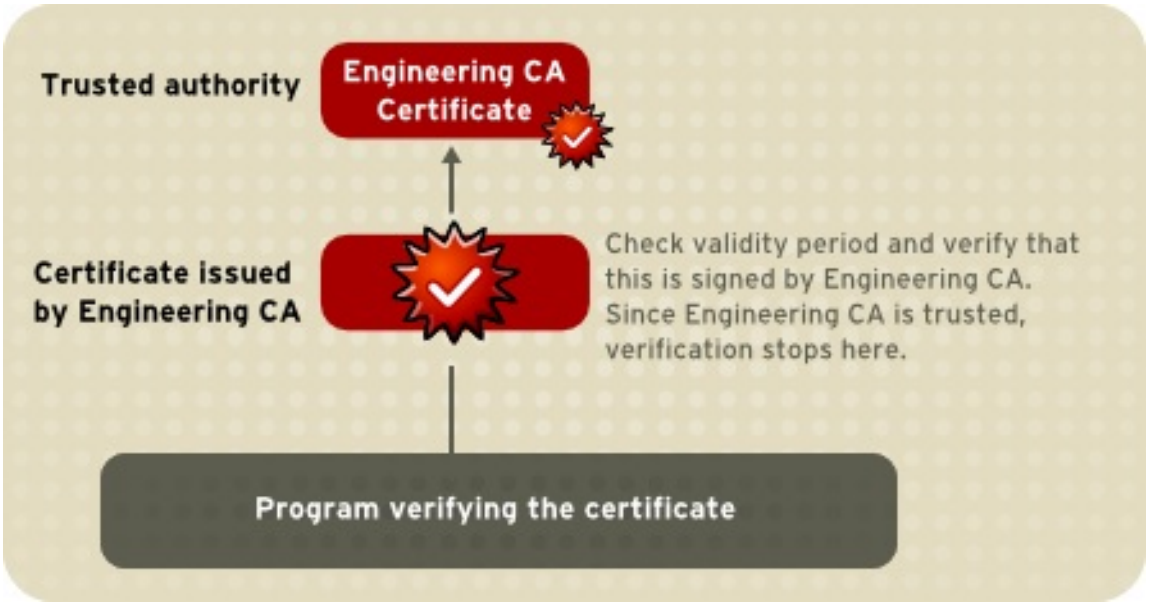


Figure 1.9. Verifying a Certificate Chain to an Intermediate CA

Expired validity dates, an invalid signature, or the absence of a certificate for the issuing CA at any point in the certificate chain causes authentication to fail. Figure 1.10, “A Certificate Chain That Cannot Be Verified” shows how verification fails if neither the root CA certificate nor any of the intermediate CA certificates are included in the verifier's local database.

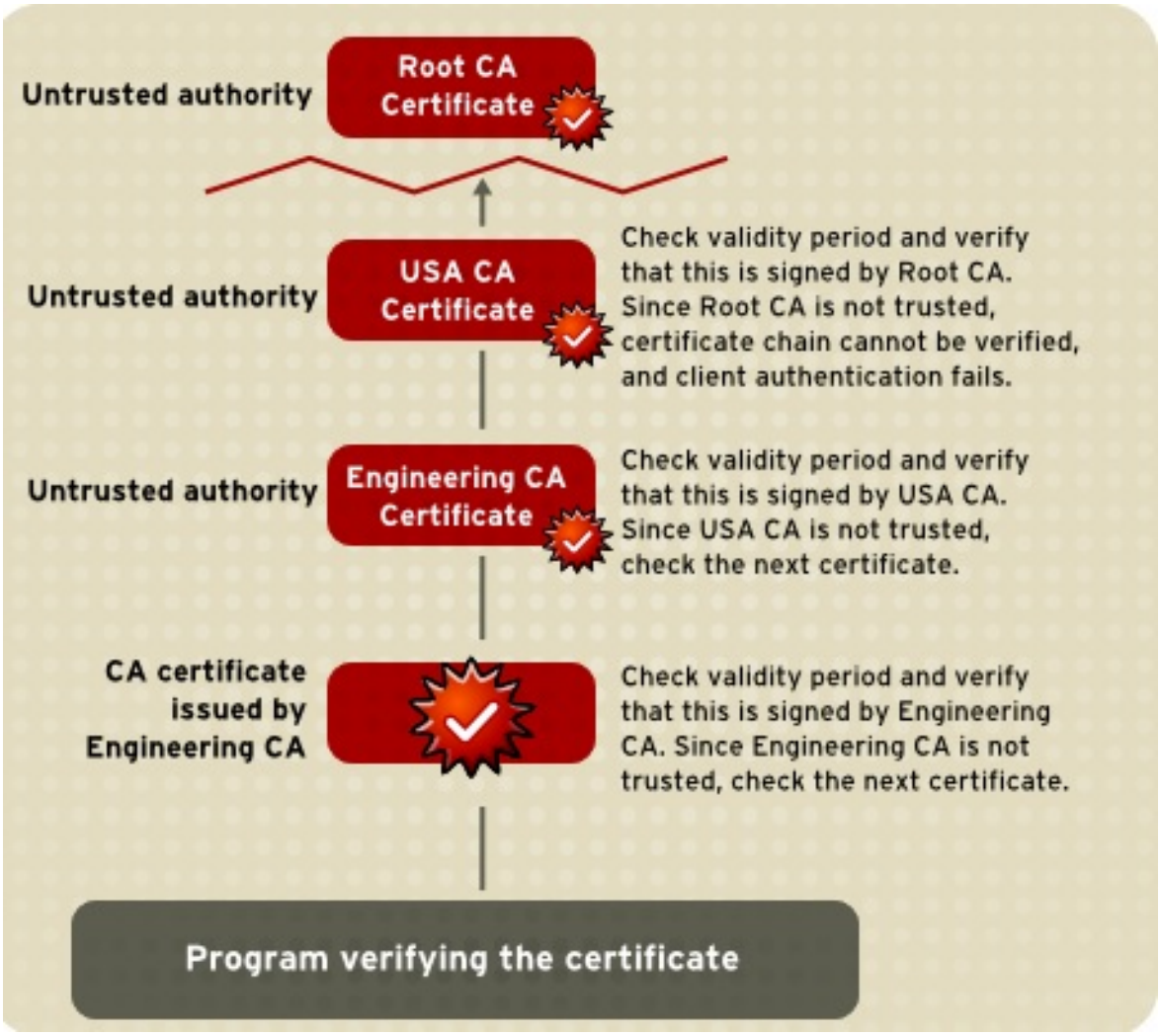


Figure 1.10. A Certificate Chain That Cannot Be Verified

1.3.7. Certificate Status

1.3.7.1. Certificate Revocation List (CRL)

1.3.7.2. Online Certificate Status Protocol (OCSP)

1.4. CERTIFICATE LIFE CYCLE

Certificates are used in many applications, from encrypting email to accessing websites. There are two major stages in the lifecycle of the certificate: the point when it is issued (issuance and enrollment) and the period when the certificates are no longer valid (renewal or revocation). There are also ways to manage the certificate during its cycle. Making information about the certificate available to other applications is *publishing* the certificate and then backing up the key pairs so that the certificate can be recovered if it is lost.

1.4.1. Certificate Issuance

The process for issuing a certificate depends on the CA that issues it and the purpose for which it will be used. Issuing non-digital forms of identification varies in similar ways. The requirements to get a library card are different than the ones to get a driver's license.

Similarly, different CAs have different procedures for issuing different kinds of certificates. Requirements for receiving a certificate can be as simple as an email address or user name and password to notarized documents, a background check, and a personal interview.

Depending on an organization's policies, the process of issuing certificates can range from being completely transparent for the user to requiring significant user participation and complex procedures. In general, processes for issuing certificates should be flexible, so organizations can tailor them to their changing needs.

1.4.2. Certificate Expiration and Renewal

Like a driver's license, a certificate specifies a period of time during which it is valid. Attempts to use a certificate for authentication before or after its validity period will fail. Managing certificate expirations and renewals are an essential part of the certificate management strategy. For example, an administrator may wish to be notified automatically when a certificate is about to expire so that an appropriate renewal process can be completed without disrupting the system operation. The renewal process may involve reusing the same public-private key pair or issuing a new one.

Additionally, it may be necessary to revoke a certificate before it has expired, such as when an employee leaves a company or moves to a new job in a different unit within the company.

Certificate revocation can be handled in several different ways:

Verify if the certificate is present in the directory

Servers can be configured so that the authentication process checks the directory for the presence of the certificate being presented. When an administrator revokes a certificate, the certificate can be automatically removed from the directory, and subsequent authentication attempts with that certificate will fail, even though the certificate remains valid in every other respect.

Certificate revocation list (CRL)

A list of revoked certificates, a CRL, can be published to the directory at regular intervals. The CRL can be checked as part of the authentication process.

Real-time status checking

The issuing CA can also be checked directly each time a certificate is presented for authentication. This procedure is sometimes called real-time status checking.

Online Certificate Status Protocol

The Online Certificate Status Protocol (OCSP) service can be configured to determine the status of certificates.

For more information about renewing certificates, see Section 2.4.2, “Renewing Certificates”. For more information about revoking certificates, including CRLs and OCSP, see Section 2.4.4, “Revoking Certificates and Checking Status”.

1.5. KEY MANAGEMENT

Before a certificate can be issued, the public key it contains and the corresponding private key must be generated. Sometimes it may be useful to issue a single person one certificate

and key pair for signing operations and another certificate and key pair for encryption operations. Separate signing and encryption certificates keep the private signing key only on the local machine, providing maximum nonrepudiation. This also aids in backing up the private encryption key in some central location where it can be retrieved in case the user loses the original key or leaves the company.

Keys can be generated by client software or generated centrally by the CA and distributed to users through an LDAP directory. There are costs associated with either method. Local key generation provides maximum nonrepudiation but may involve more participation by the user in the issuing process. Flexible key management capabilities are essential for most organizations.

Key recovery , or the ability to retrieve backups of encryption keys under carefully defined conditions, can be a crucial part of certificate management, depending on how an organization uses certificates. In some PKI setups, several authorized personnel must agree before an encryption key can be recovered to ensure that the key is only recovered to the legitimate owner in authorized circumstance. It can be necessary to recover a key when information is encrypted and can only be decrypted by the lost key.

CHAPTER 2. INTRODUCTION TO RED HAT CERTIFICATE SYSTEM

Every common PKI operation, such as issuing, renewing, and revoking certificates; archiving and recovering keys; publishing CRLs and verifying certificate status, is carried out by interoperating subsystems within Red Hat Certificate System. The functions of each individual subsystem and the way that they work together to establish a robust and local PKI is described in this chapter.

2.1. A REVIEW OF CERTIFICATE SYSTEM SUBSYSTEMS

Red Hat Certificate System provides five different subsystems, each focusing on different aspects of a PKI deployment:

- A *certificate authority* called *Certificate Manager*. The CA is the core of the PKI; it issues and revokes all certificates. The Certificate Manager is also the core of the Certificate System. By establishing a *security domain* of trusted subsystems, it establishes and manages relationships between the other subsystems.

- A *key recovery authority* (KRA). Certificates are created based on a specific and unique key pair. If a private key is ever lost, then the data which that key was used to access (such as encrypted emails) is also lost because it is inaccessible. The KRA stores key pairs, so that a new, identical certificate can be generated based on recovered keys, and all of the encrypted data can be accessed even after a private key is lost or damaged.

NOTE

In previous versions of Certificate System, KRA was also referred to as the data recovery manager (DRM). Some code, configuration file entries, web panels, and other resources might still use the term DRM instead of KRA.

- An *online certificate status protocol* (OCSP) responder. The OCSP verifies whether a certificate is valid and not expired. This function can also be done by the CA, which has an internal OCSP service, but using an external OCSP responder lowers the load of the issuing CA.

- A *token key service* (TKS). The TKS derives keys based on the token CCID, private information, and a defined algorithm. These derived keys are used by the TPS to format tokens and enroll certificates on the token.

- A *token processing system* (TPS). The TPS interacts directly with external tokens, like smart cards, and manages the keys and certificates on those tokens through a local client, the Enterprise Security Client (ESC). The ESC contacts the TPS when there is a token operation, and the TPS interacts with the CA, KRA, or TKS, as required, then send the information back to the token by way of the Enterprise Security Client.

Even with all possible subsystems installed, the core of the Certificate System is still the CA (or CAs), since they ultimately process all certificate-related requests. The other subsystems connect to the CA or CAs likes spokes in a wheel. These subsystems work together, in tandem, to create a public key infrastructure (PKI). Depending on what subsystems are installed, a PKI can function in one (or both) of two ways:

- A *token management system* or *TMS* environment, which manages smart cards. This requires a CA, TKS, and TPS, with an optional KRA for server-side key generation.
- A traditional *non token management system* or *non-TMS* environment, which manages certificates used in an environment other than smart cards, usually in software databases. At a minimum, a non-TMS requires only a CA, but a non-TMS environment can use OCSP responders and KRA instances as well.

2.2. OVERVIEW OF CERTIFICATE SYSTEM SUBSYSTEMS

2.2.1. Separate versus Shared Instances

Red Hat Certificate System supports deployment of separate PKI instances for all subsystems:

- Separate PKI instances run as a single Java-based Apache Tomcat instance.
- Separate PKI instances contain a single PKI subsystem (CA, KRA, OCSP, TKS, or TPS).
- Separate PKI instances must utilize unique ports if co-located on the same physical machine or virtual machine (VM).

Alternatively, Certificate System supports deployment of a shared PKI instance:

- Shared PKI instances also run as a single Java-based Apache Tomcat instance.
- Shared PKI instances that contain a single PKI subsystem are identical to a separate PKI instance.
- Shared PKI instances may contain any combination of up to one of each type of PKI subsystem:
  - CA only
  - TKS only
  - CA and KRA
  - CA and OCSP
  - TKS and TPS
  - CA, KRA, TKS, and TPS
  - CA, KRA, OCSP, TKS, and TPS
  - etc.
- Shared PKI instances allow all of their subsystems contained within that instance to share the same ports.
- Shared PKI instances must utilize unique ports if more than one is co-located on the same physical machine or VM.

2.2.2. Instance Installation Prerequisites

2.2.2.1. Directory Server Instance Availability

Prior to installation of a Certificate System instance, a local or remote Red Hat Directory Server LDAP instance must be available. For instructions on installing Red Hat Directory Server, see the *Red Hat Directory Server Installation Guide*.

2.2.2.2. PKI Packages

Red Hat Certificate System is composed of packages listed below:

- The following base packages form the core of Certificate System, and are available in base Red Hat Enterprise Linux repositories:
  - pki-core.el7
    - pki-base
    - pki-base-java
    - pki-ca
    - pki-javadoc
    - pki-kra
    - pki-server
    - pki-symkey
    - pki-tools
- The packages listed below are *not* available in the base Red Hat Enterprise Linux subscription channel. To install these packages, you must first use **Subscription Manager** to attach the Red Hat Certificate System subscription pool, and enable the RHCS9 repository. See the Subscription Manager chapter of the Red Hat Enterprise Linux 7 System Administrator's Guide for instructions.
  - pki-console.el7pki
    - pki-console
  - pki-core.el7pki
    - pki-ocsp
    - pki-tks
    - pki-tps
  - redhat-pki.el7pki
    - redhat-pki
  - redhat-pki-theme.el7pki
    - redhat-pki-console-theme
    - redhat-pki-server-theme

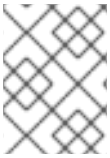
Use a Red Hat Enterprise Linux 7 system (optionally, use one that has been configured with a supported Hardware Security Module listed in Chapter 4, *Supported Platforms, Hardware, and Programs*), and make sure that all packages are up to date before installing Red Hat Certificate System.

To install all Certificate System packages (with the exception of pki-javadoc), use use **Yum** to install the redhat-pki metapackage:

```
# yum install redhat-pki
```

Alternatively, install one or more of the top level PKI subsystem packages as required; see the list above for exact package names. If you use this approach, make sure to also install the redhat-pki-server-theme package, and optionally redhat-pki-console-theme and pki-console to use the PKI Console.

Finally, developers and administrators may also want to install the JSS and PKI javadocs (the jss-javadoc and pki-javadoc).



NOTE

The jss-javadoc package requires you to enable the Server-Optional repository in **Subscription Manager**.

2.2.2.3. Instance Installation and Configuration

The **pkispawn** command line tool is used to install and configure a new PKI instance. It eliminates the need for separate installation and configuration steps, and may be run either interactively, as a batch process, or a combination of both (batch process with prompts for passwords). The utility does not provide a way to install or configure the browser-based graphical interface.

For usage information, use the **pkispawn --help** command.

The **pkispawn** command:

1. Reads in its default **name=value** pairs from a plain text configuration file (**/etc/pki/default.cfg**).

2. Interactively or automatically overrides any pairs as specified and stores the final result as a Python dictionary.

3. Executes an ordered series of *scriptlets* to perform subsystem and instance installation.

4. The configuration scriptlet packages the Python dictionary as a JavaScript Object Notation (JSON) data object, which is then passed to the Java-based configuration servlet.

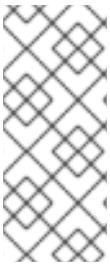
5. The configuration servlet utilizes this data to configure a new PKI subsystem, and then passes control back to the **pkispawn** executable, which finalizes the PKI setup. A copy of the final deployment file is stored in **/var/lib/pki/*instance_name*/<*subsystem*>/registry/<*subsystem*>/deployment.cfg**

See the **pkispawn** man page for additional information.

The default configuration file, **/etc/pki/default.cfg**, is a plain text file containing the default installation and configuration values which are read at the beginning of the process described above. It consists of **name=value** pairs divided into **[DEFAULT]**, **[Tomcat]**, **[CA]**, **[KRA]**, **[OCSP]**, **[TKS]**, and **[TPS]** sections.

If you use the **-s** option with **pkispawn** and specify a subsystem name, then only the section for that subsystem will be read.

The sections have a hierarchy: a **name=value** pair specified in a subsystem section will override the pair in the **[Tomcat]** section, which in turn override the pair in the **[DEFAULT]** section. Default pairs can further be overriden by interactive input, or by pairs in a specified PKI instance configuration file.



NOTE

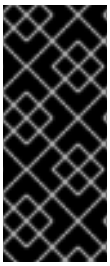
Whenever non-interactive files are used to override default **name=value** pairs, they may be stored in any location and specified at any time. These files are referred to as **myconfig.txt** in the **pkispawn** man pages, but they are also often referred to as **.ini** files, or more generally as PKI instance configuration override files.

See the **pki_default.cfg** man page for more information.

The *Configuration Servlet* consists of Java bytecode stored in **/usr/share/java/pki/pki-certsrv.jar** as **com/netscape/certsrv/system/ConfigurationRequest.class**. The servlet processes data passed in as a JSON object from the configuration scriptlet using **pkispawn**, and then returns to **pkispawn** using Java bytecode served in the same file as **com/netscape/certsrv/system/ConfigurationResponse.class**.

An example of an interactive installation only involves running the **pkispawn** command on a command line as **root**:

```
# pkispawn
```



IMPORTANT

Interactive installation currently only exists for very basic deployments. For example, deployments intent upon using advanced features such as cloning, Elliptic Curve Cryptography (ECC), external CA, Hardware Security Module (HSM), subordinate CA, and others, must provide the necessary override parameters in a separate configuration file.

A non-interactive installation requires a PKI instance configuration override file, and the process may look similar to the following example:

1. ```
   # mkdir -p /root/pki
   ```

2. Use a text editor such as **vim** to create a configuration file named **/root/pki/ca.cfg** with the following contents:

   ```
   [DEFAULT]
   pki_admin_password=<password>
   pki_client_pkcs12_password=<password>
   ```
   ```

```
pki_ds_password=<password>
```

3. ```
   # pkispawn -s CA -f /root/pki/ca.cfg
   ```

See the **pkispawn** man page for various configuration examples.

2.2.2.4. Instance Removal

To remove an existing PKI instance, use the **pkidestroy** command. It can be run interactively or as a batch process. Use **pkidestroy -h** to display detailed usage inforamtion on the command line.

The **pkidestroy** command reads in a PKI subsystem deployment configuration file which was stored when the subsystem was created (**/var/lib/pki/*instance_name*/<*subsystem*>/registry/<*subsystem*>/deployment.cfg**), uses the read-in file in order to remove the PKI subsystem, and then removes the PKI instance if it contains no additional subsystems. See the **pkidestroy** man page for more information.

An interactive removal procedure using **pkidestroy** may look similar to the following:

```
# pkidestroy
Subsystem (CA/KRA/OCSP/TKS/TPS) [CA]:
Instance [pki-tomcat]:

Begin uninstallation (Yes/No/Quit)? Yes

Log file: /var/log/pki/pki-ca-destroy.20150928183547.log
Loading deployment configuration from /var/lib/pki/pki-
tomcat/ca/registry/ca/deployment.cfg.
Uninstalling CA from /var/lib/pki/pki-tomcat.
rm '/etc/systemd/system/multi-user.target.wants/pki-tomcatd.target'

Uninstallation complete.
```

A non-interactive removal procedure may look similar to the following example:

```
# pkidestroy -s CA -i pki-tomcat
Log file: /var/log/pki/pki-ca-destroy.20150928183159.log
Loading deployment configuration from /var/lib/pki/pki-
tomcat/ca/registry/ca/deployment.cfg.
Uninstalling CA from /var/lib/pki/pki-tomcat.
rm '/etc/systemd/system/multi-user.target.wants/pki-tomcatd.target'

Uninstallation complete.
```

2.2.3. Execution Management (systemctl)

2.2.3.1. Starting, Stopping, Restarting, and Obtaining Status

Red Hat Certificate System subsystem instances can be stopped and started using the **systemctl** execution management system tool on Red Hat Enterprise Linux 7:
```

```
# systemctl start pki-tomcatd@instance_name.service
```

```
# systemctl status pki-tomcatd@instance_name.service
```

```
# systemctl stop pki-tomcatd@instance_name.service
```

```
# systemctl restart pki-tomcatd@instance_name.service
```

2.2.3.2. Starting the Instance Automatically

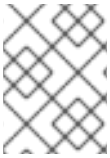
The **systemctl** utility in Red Hat Enterprise Linux 7 manages the automatic startup and shutdown settings for each process on the server. This means that when a system reboots, some services can be automatically restarted. System unit files control service startup to ensure that services are started in the correct order. The **systemd** service and **systemctl** utility are described in the *Red Hat Enterprise Linux System Administrator's Guide*

Certificate System instances can be managed by **systemctl**, so this utility can set whether to restart instances automatically. After a Certificate System instance is created, it is enabled on boot. This can be changed by using **systemctl**:

```
# systemctl disable pki-tomcatd@instance_name.service
```

To re-enable the instance:

```
# systemctl enable pki-tomcatd@instance_name.service
```



NOTE

The **systemctl enable** and **systemctl disable** commands do not immediately start or stop Certificate System.

2.2.4. Process Management (pki-server and pkidaemon)

2.2.4.1. The pki-server Command Line Tool

The primary process management tool for Red Hat Certificate System is **pki-server**. Use the **pki-server --help** command and see the **pki-server** man page for usage information.

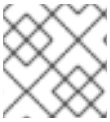
2.2.4.2. Enabling and Disabling an Installed Subsystem Using pki-server

To enable or disable an installed subsystem, use the **pki-server** utility.

```
# pki-server subsystem-disable -i instance_id subsystem_id
```

```
# pki-server subsystem-enable -i instance_id subsystem_id
```

Replace *subsystem_id* with a valid subsystem identifier: **ca**, **kra**, **tks**, **ocsp**, or **tps**.



NOTE

One instance can have only one of each type of subsystem.

For example, to disable the OCSP subsystem on an instance named **pki-tomcat**:

```
# pki-server subsystem-disable -i pki-tomcat ocsp
```

To list the installed subsystems for an instance:

```
# pki-server subsystem-find -i instance_id
```

To show the status of a particular subsystem:

```
# pki-server subsystem-find -i instance_id subsystem_id
```

2.2.4.3. The pkidaemon Command Line Tool

Another process management tool for Red Hat Certificate System is the **pkidaemon** tool:

```
pkidaemon {start|status} instance-type [instance_name]
```

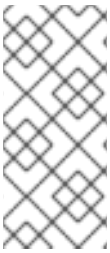
- **pkidaemon status tomcat** - Provides status information such as on/off, ports, URLs of each PKI subsystem of all PKI instances on the system.
- **pkidaemon status tomcat *instance_name*** - Provides status information such as on/off, ports, URLs of each PKI subsystem of a specific instance.
- **pkidaemon start tomcat *instance_name*.service** - Used internally using **systemctl**.

See the **pkidaemon** man page for additional information.

2.2.4.4. Finding the Subsystem Web Services URLs

The CA, KRA, OCSP, TKS, and TPS subsystems have web services pages for agents, as well as regular users and administrators, when appropriate. These web services can be accessed by opening the URL to the subsystem host over the subsystem's secure end user's port. For example, for the CA:

```
https://server.example.com:8443/ca/services
```



NOTE

To get a complete list of all of the interfaces, URLs, and ports for an instance, check the status of the service. For example:

```
pkidaemon status tomcat instance_name
```

The main web services page for each subsystem has a list of available services pages; these are summarized in Table 2.1, “Default Web Services Pages”. To access any service

specifically, access the appropriate port and append the appropriate directory to the URL. For example, to access the CA's end entities (regular users) web services:

```
https://server.example.com:8443/ca/ee/ca
```

If DNS is not configured, then an IPv4 or IPv6 address can be used to connect to the services pages. For example:

```
https://192.0.2.1:8443/ca/services
https://[2001:DB8::1111]:8443/ca/services
```

NOTE

Anyone can access the end user pages for a subsystem. However, accessing agent or admin web services pages requires that an agent or administrator certificate be issued and installed in the web browser. Otherwise, authentication to the web services fails.

Table 2.1. Default Web Services Pages

Port	Used for SSL/TLS	Used for Client Authentication [a]	Web Services	Web Service Location
Certificate Manager				
8080	No		End Entities	ca/ee/ca
8443	Yes	No	End Entities	ca/ee/ca
8443	Yes	Yes	Agents	ca/agent/ca
8443	Yes	No	Services	ca/services
8443	Yes	No	Console	pkiconsole https://*host:port*/ca
Key Recovery Authority				
8080	No		End Entities[b]	kra/ee/kra
8443	Yes	No	End Entities[b]	kra/ee/kra
8443	Yes	Yes	Agents	kra/agent/kra
8443	Yes	No	Services	kra/services

Port	Used for SSL/TLS	Used for Client Authentication [a]	Web Services	Web Service Location
8443	Yes	No	Console	pkiconsole https://*host:port*/kra
Online Certificate Status Manager				
8080	No		End Entities[c]	ocsp/ee/ocsp
8443	Yes	No	End Entities[c]	ocsp/ee/ocsp
8443	Yes	Yes	Agents	ocsp/agent/ocsp
8443	Yes	No	Services	ocsp/services
8443	Yes	No	Console	pkiconsole https://*host:port*/ocsp
Token Key Service				
8080	No		End Entities[b]	tks/ee/tks
8443	Yes	No	End Entities[b]	tks/ee/tks
8443	Yes	Yes	Agents	tks/agent/tks
8443	Yes	No	Services	tks/services
8443	Yes	No	Console	pkiconsole https://*host:port*/tks
Token Processing System				
8080	No		Unsecure Services	tps/tps
8443	Yes		Secure Services	tps/tps

Port	Used for SSL/TLS	Used for Client Authentication [a]	Web Services	Web Service Location
8080	No		Enterprise Security Client Phone Home	tps/phoneHome
8443	Yes		Enterprise Security Client Phone Home	tps/phoneHome
8443	Yes	Yes	Admin, Agent, and Operator Services [d]	tps/ui

[a] Services with a client authentication value of **No** can be reconfigured to require client authentication. Services which do not have either a **Yes** or **No** value cannot be configured to use client authentication.

[b] Although this subsystem type does have end entities ports and interfaces, these end-entity services are not accessible through a web browser, as other end-entity services are.

[c] Although the OCSP does have end entities ports and interfaces, these end-entity services are not accessible through a web browser, as other end-entity services are. End user OCSP services are accessed by a client sending an OCSP request.

[d] The agent, admin, and operator services are all accessed through the same web services page. Each role can only access specific sections which are only visible to the members of that role.

2.2.4.5. Starting the Certificate System Console

The CA, KRA, OCSP, and TKS subsystems have a Java interface which can be accessed to perform administrative functions. For the KRA, OCSP, and TKS, this includes very basic tasks like configuring logging and managing users and groups. For the CA, this includes other configuration settings such as creating certificate profiles and configuring publishing.

The Console is opened by connecting to the subsystem instance over its SSL/TLS port using the **pkiconsole** utility. This utility uses the format:

```
pkiconsole https://server.example.com:admin_port/subsystem_type
```

The *subsystem_type* can be **ca**, **kra**, **ocsp**, or **tks**. For example, this opens the KRA console:

```
pkiconsole https://server.example.com:8443/kra
```

If DNS is not configured, then an IPv4 or IPv6 address can be used to connect to the console. For example:

```
https://192.0.2.1:8443/ca
https://[2001:DB8::1111]:8443/ca
```

2.3. CERTIFICATE SYSTEM ARCHITECTURE OVERVIEW

Although each provides a different service, all RHCS subsystems (CA, KRA, OCSP, TKS, TPS) share a common architecture. The following architectural diagram shows the common architecture shared by all of these subsystems.

2.3.1. Java Application Server

Java application server is a Java framework to run server applications. The Certificate System is designed to run within a Java application server. Currently the only Java application server supported is Tomcat 7 and 8. Support for other application servers may be added in the future. More information can be found at http://tomcat.apache.org/.

Each Certificate System instance is a Tomcat server instance. The Tomcat configuration is stored in **`server.xml`**. The following links provide more information about Tomcat configuration;

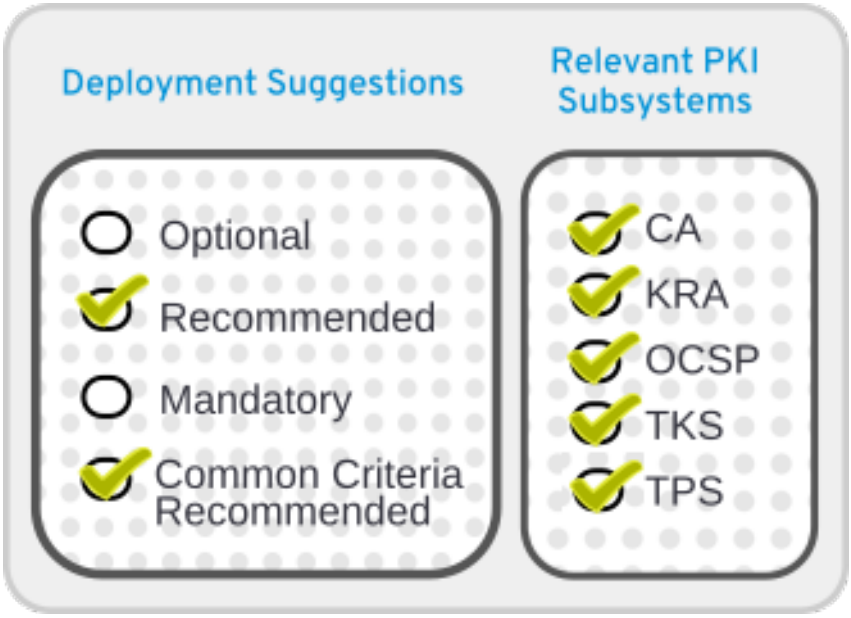
- https://tomcat.apache.org/tomcat-7.0-doc/config/
- https://tomcat.apache.org/tomcat-8.0-doc/config/

Each Certificate System subsystem (such as CA or KRA) is deployed as a web application in Tomcat. The web application configuration is stored in a **`web.xml`** file, which is defined in Java Servlet 3.0 specification. See https://jcp.org/en/jsr/detail?id=315 for details.

The Certificate System configuration itself is stored in **`CS.cfg`**.

See Section 2.3.14, “Instance Layout” for the actual locations of these files.

2.3.2. Java Security Manager



Java services have the option of having a Security Manager which defines unsafe and safe operations for applications to perform. When the subsystems are installed, they have the Security Manager enabled automatically, meaning each Tomcat instance starts with the Security Manager running.

The Security Manager is disabled if the instance is created by running pkispawn and using an override configuration file which specifies the **`pki_security_manager=false`** option under its own Tomcat section.

The Security Manager can be disabled from an installed instance using the following procedure:

1. ```
   # systemctl stop pki-tomcatd@instance_name.service
   ```

   or

   ```
   # systemctl stop pki-tomcatd-nuxwdog@instance_name.service
   ```

   (if using the **nuxwdog** watchdog)

2. Open the **/etc/sysconfig/*instance_name*** file, and set **SECURITY_MANAGER="false"**

3. ```
   # systemctl start pki-tomcatd@instance_name.service
   ```

   or

   ```
   # systemctl start pki-tomcatd-nuxwdog@instance_name.service
   ```

   (if using the **nuxwdog** watchdog)

When an instance is started or restarted, a Java security policy is constructed or reconstructed by **pkidaemon** from the following files:

```
/usr/share/pki/server/conf/catalina.policy
/usr/share/tomcat/conf/catalina.policy
/var/lib/pki/$PKI_INSTANCE_NAME/conf/pki.policy
/var/lib/pki/$PKI_INSTANCE_NAME/conf/custom.policy
```

Then, it is saved into **/var/lib/pki/*instance_name*/conf/catalina.policy**.

2.3.3. Interfaces

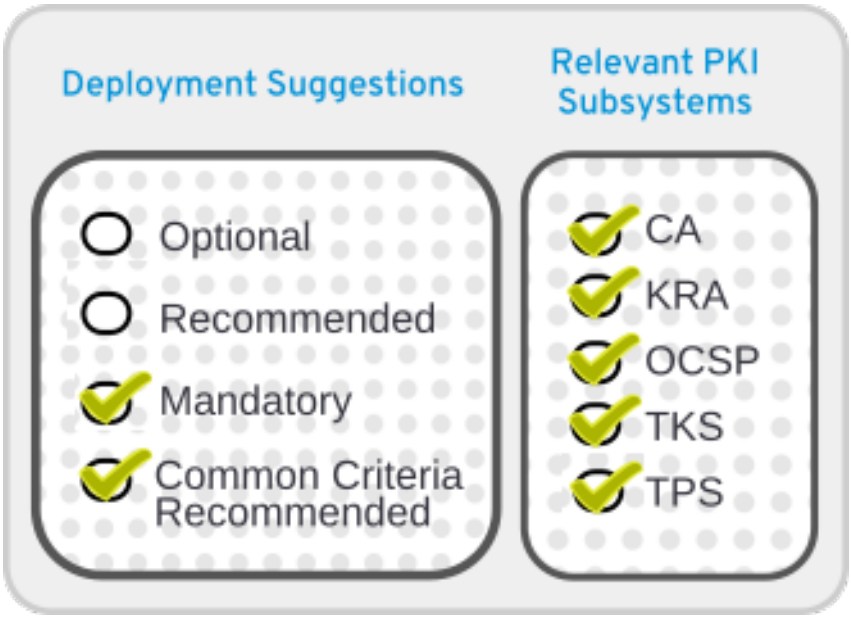
2.3.3.1. Servlet Interface

Each subsystem contains interfaces allowing interaction with various portions of the subsystem. All subsystems share a common administrative interface and have an agent interface that allows for agents to perform the tasks assigned to them. A CA Subsystem has an end-entity interface that allows end-entities to enroll in the PKI. An OCSP Responder subsystem has an end-entity interface allowing end-entities and applications to check for current certificate revocation status. Finally, a TPS has an operator interface.

While the application server provides the connection entry points, Certificate System completes the interfaces by providing the servlets specific to each interface.

The servlets for each subsystem are defined in the corresponding **web.xml** file. The same file also defines the URL of each servlet and the security requirements to access the servlets. See Section 2.3.1, “Java Application Server” for more information.

2.3.3.2. Administrative Interface

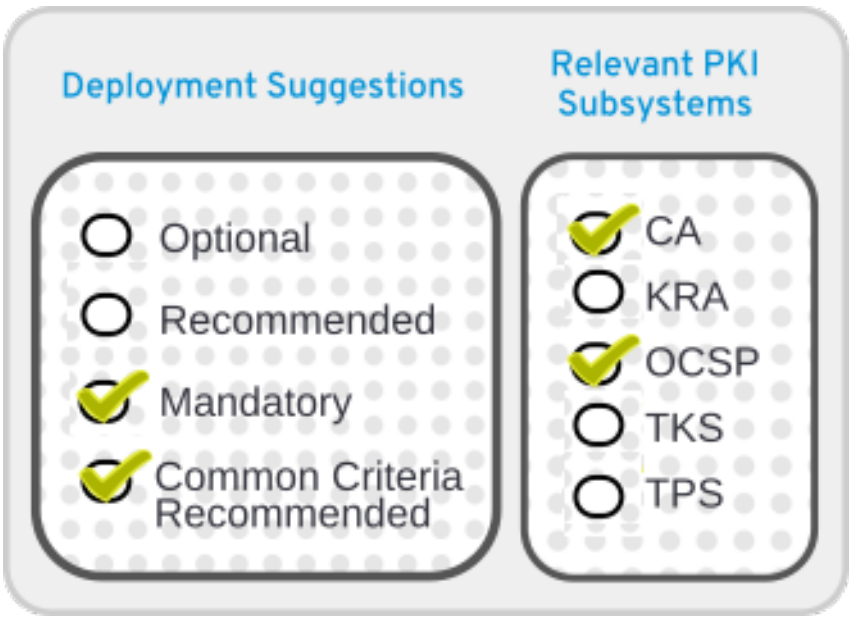


The agent interface provides Java servlets to process HTML form submissions coming from the agent entry point. Based on the information given in each form submission, the agent servlets allow agents to perform agent tasks, such as editing and approving requests for certificate approval, certificate renewal, and certificate revocation, approving certificate profiles. The agent interfaces for a KRA subsystem, or a TKS subsystem, or an OCSP Responder are specific to the subsystems.

In the non-TMS setup, the agent interface is also used for inter-CIMC boundary communication for the CA-to-KRA trusted connection. This connection is protected by SSL client-authentication and differentiated by separate trusted roles called *Trusted Managers*. Like the agent role, the Trusted Managers (pseudo-users created for inter-CIMC boundary connection only) are required to be SSL client-authenticated. However, unlike the agent role, they are not offered any agent capability.

In the TMS setup, inter-CIMC boundary communication goes from TPS-to-CA, TPS-to-KRA, and TPS-to-TKS.

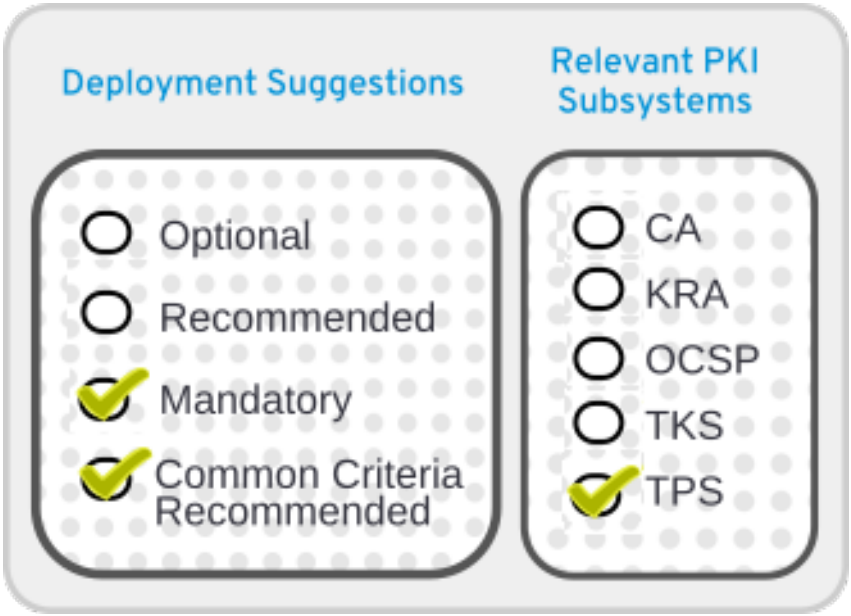
2.3.3.3. End-Entity Interface



For the CA subsystem, the end-entity interface provides Java servlets to process HTML form

submissions coming from the end-entity entry point. Based on the information received from the form submissions, the end-entity servlets allow end-entities to enroll, renew certificates, revoke their own certificates, and pick up issued certificates. The OCSP Responder subsystem's End-Entity interface provides Java servlets to accept and process OCSP requests. The KRA, TKS, and TPS subsystems do not offer any End-Entity services.

2.3.3.4. Operator Interface



The operator interface is only found in the TPS subsystem.

2.3.4. REST Interface

Representational state transfer (REST) is a way to use HTTP to define and organize web services which will simplify interoperability with other applications. Red Hat Certificate System provides a REST interface to access various services on the server.

The REST services in Red Hat Certificate System are implemented using the RESTEasy framework. RESTEasy is actually running as a servlet in the web application, so the RESTEasy configuration can also be found in the **`web.xml`** of the corresponding subsystem. More information about RESTEasy can be found at http://resteasy.jboss.org/.

Each REST service is defined as a separate URL. For example:

- CA certificate service: **`http://<host_name>:<port>/ca/rest/certs/`**
- KRA key service: **`http://<host_name>:<port>/kra/rest/agent/keys/`**
- TKS user service: **`http://<host_name>:<port>/tks/rest/admin/users/`**
- TPS group service: **`http://<host_name>:<port>/tps/rest/admin/groups/`**

Some services can be accessed using plain HTTP connection, but some others may require HTTPS connection for security.

The REST operation is specified as HTTP method (for example, GET, PUT, POST, DELETE). For example, to get the CA users the client will send a GET /ca/rest/users request.

The REST request and response messages can be sent in XML or JSON format. For example:

```
{
 "id":"admin",
 "UserID":"admin",
 "FullName":"Administrator",
 "Email":"admin@example.com",
 ...
}
```

The REST interface can be accessed using tools such as CLI, Web UI, or generic REST client. Certificate System also provides Java, Python, and JavaScript libraries to access the services programmatically.

The REST interface supports two types of authentication methods:

- user name and password
- client certificate

The authentication method required by each service is defined in **/usr/share/pki/ca/conf/auth-method.properties**.

The REST interface may require certain permissions to access the service. The permissions are defined in the ACL resources in LDAP. The REST interface are mapped to the ACL resources in the **/usr/share/pki/*<subsystem>*/conf/acl.properties**.

For more information about the REST interface, see http://www.dogtagpki.org/wiki/REST.

2.3.5. JSS

Java Security Services (JSS) provides a Java interface for security operations performed by NSS. JSS and higher levels of the Certificate System architecture are built with Java Native Interface (JNI), which provides binary compatibility across different versions of the Java Virtual Machine (JVM). This design allows customized subsystem services to be compiled and built just once and run on a range of platforms. JSS supports most of the security standards and encryption technologies supported by NSS. JSS also provides a pure Java interface for ASN.1 types and BER-DER encoding.

2.3.6. Tomcatjss

Java-based subsystems in Red Hat Certificate System use a single JAR file called **tomcatjss** as a bridge between the Tomcat Server HTTP engine and JSS, the Java interface for security operations performed by NSS. **Tomcatjss** is a Java Secure Socket Extension (JSSE) implementation using Java Security Services (JSS) for Tomcat.

The following configuration in the **server.xml** file found in the**pki-tomcat/conf** directory can be used to explain how **Tomcatjss** fits into the entire Certificate System ecosystem. Portions of the Connector entry for the secret port are shown below.

```
<Connector name="Secure"

# Info about the socket itself
 port="8443"
 protocol="org.apache.coyote.http11.Http11Protocol"
 SSLEnabled="true"
 sslProtocol="SSL"
```

```
 scheme="https"
 secure="true"
 connectionTimeout="80000"
 maxHttpHeaderSize="8192"
 acceptCount="100" maxThreads="150" minSpareThreads="25"
 enableLookups="false" disableUploadTimeout="true"
 #Points to our tomcat jss implementation
 sslImplementationName="org.apache.tomcat.util.net.jss.JSSImplementation"
 enableOCSP="false"

#Optional ocsp responder configuration can be enabled here
 ocspResponderURL="http://server.example.com.com:9080/ca/ocsp"
 ocspResponderCertNickname="ocspSigningCert cert-pki-ca"
 ocspCacheSize="1000"
 ocspMinCacheEntryDuration="60"
 ocspMaxCacheEntryDuration="120"
 ocspTimeout="10"
 strictCiphers="true"
 #This configures the client auth scheme we want for this server socket.
Want means that the client auth cert is optional. We could also have made
it “required” to give no choice.
 clientAuth="want"

# A collection of cipher related settings that make sure connections are
secure.
 sslOptions="ssl2=false,ssl3=false,tls=true"
 ssl3Ciphers="-SSL3_FORTEZZA_ ..."
 tlsCiphers="-TLS_ECDH_ECDSA_WITH_AES_128_"
 sslVersionRangeStream="tls1_0:tls1_2"
 sslVersionRangeDatagram="tls1_1:tls1_2"
 sslRangeCiphers="-TLS_ECDH_ECDSA_W"
 serverCertNickFile="/var/lib/pki/pki-tomcat/conf/serverCertNick.conf"
 passwordFile="/var/lib/pki/pki-tomcat/conf/password.conf"
 passwordClass="org.apache.tomcat.util.net.jss.PlainPasswordFile"
 certdbDir="/var/lib/pki/pki-tomcat/alias"
/>
```

Note that this Connector contains the pointer to the tomcatjss implementation, which can be plugged into the **sslImplementation** property of this Connector object.

Tomcatjss implements the interfaces needed to use SSL and to create SSL sockets. The socket factory, which tomcatjss implements, makes use of the various properties listed below to create an SSL server listening socket and return it to tomcat. Tomcatjss itself, makes use of our java JSS system to ultimately communicate with the native NSS cryptographic services on the machine.

Tomcatjss is loaded when the Tomcat server and the Certificate System classes are loaded. The load process is described below:

1. The server is started.

2. Tomcat gets to the point where it needs to create the listening sockets for the Certificate System installation.

3. The **server.xml** file is processed. Configuration in this file tells the system to use a socket factory implemented by Tomcatjss.

4. For each requested socket, Tomcajss reads and processes the included attributes when it creates the socket. The resulting socket will behave as it has been asked to by those parameters.

5. Once the server is running, we have the required set of listening sockets waiting for incoming connections to the Tomcat-based Certificate System.

Note that when the sockets are created at startup, Tomcatjss is the *first* entity in Certificate System that actually deals with the underlying JSS security services. Once the first listening socket is processed, an instance of JSS is created for use going forward.

2.3.7. PKCS#11 Modules

Public-Key Cryptography Standard (PKCS) #11 specifies an API used to communicate with devices that hold cryptographic information and perform cryptographic operations. Because it supports PKCS #11, Certificate System works with a wide range of hardware and software devices intended for such purposes.

One or more PKCS #11 modules must be available to any Certificate System subsystem instance. A *PKCS #11 module* (also called a *cryptographic module* or *cryptographic service provider*) manages cryptographic services such as encryption and decryption using the PKCS #11 interface. PKCS #11 modules can be thought of as drivers for cryptographic devices that can be implemented in either hardware or software.

NSS provides a built-in PKCS #11 module that is used by Certificate System. Network Security Services (NSS) is a set of libraries designed to support cross-platform development of security-enabled communications applications. Applications built with the NSS libraries support the SSL protocol for authentication, tamper detection, and encryption as well as the PKCS #11 interface for cryptographic token interfaces. Red Hat uses NSS to support these features in a wide range of products, including Certificate System.

The NSS document can be found online at http://www.mozilla.org/projects/security/pki/nss/overview.html. NSS tools are frequently used to manage the PKI environment. For more information, see https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/tools.

A PKCS #11 module always has one or more *slots*, which can be implemented as physical hardware slots in some form of physical reader (for example, for smart cards) or as conceptual slots in software. Each slot for a PKCS #11 module can in turn contain a *token*, which is the hardware or software device that actually provides cryptographic services and optionally stores certificates and keys.

2.3.7.1. TLS Ciphers

Each PKI deployment may have policies on how it wishes its clients to communicate with the PKI servers. RHCS allows its TLS ciphers to be configured.

When acting as a TLS server, each Certificate instance's TLS cipher list can be configured from its **`server.xml`** file. When acting as a TLS client (for inter-subsystem communication), the instance's TLS cipher list can be configured in the instance's configuration file, **`CS.cfg`**.

See the *Red Hat Certificate System Administration Guide* for configuration information..

2.3.7.2. NSS Soft Token

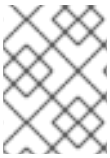
NSS provides two built-in modules with Certificate System:

- *Default NSS Internal PKCS #11 Module.*This comes with two built-in tokens:
  - The Internal Crypto Services token performs all cryptographic operations, such as encryption, decryption, and hashing.
  - The Internal Key Storage token handles all communication with the certificate and key database files (called **certX.db** and **keyX.db**, respectively, where X is a version number) that store certificates and keys.
- *FIPS 140-2 module.* This module complies with the FIPS 140-2 government standard for implementations of cryptographic modules. Many products sold to the US government must comply with one or more of the FIPS standards. The FIPS 140-2 module includes a single, built-in FIPS 140-2 Certificate DB token , which handles both cryptographic operations and communication with the **certX.db** and **keyX.db** files.

2.3.7.3. HSMs

See the **pkispawn** man page for an example of setting up a PKI subsystem to utilize an HSM.

An external token refers to an external hardware device, such as a smart card or hardware security module (HSM), that the Certificate System uses to generate and store its key pairs and certificates. The Certificate System should work with most hardware tokens that are compliant with PKCS #11 but officially, Certificate System 9 only supports certain models of the Gemalto Safenet LunaSA and nCipher nShield HSMs.



NOTE

See your specific HSM vendor documentation for installation and configuration instructions for using the HSM.

PKCS #11 is a standard set of APIs and shared libraries which isolate an application from the details of the cryptographic device. This enables the application to provide a unified interface for PKCS #11-compliant cryptographic devices.

The PKCS #11 module provided by NSS in the Certificate System supports cryptographic devices supplied by many different manufacturers. This module allows the Certificate System to plug in shared libraries supplied by manufacturers of external encryption devices and use them for generating and storing keys and certificates for the Certificate System managers.

Consider using HSMs for generating and storing the key pairs and certificates used by Certificate System. These devices are another security measure to safeguard private keys because hardware tokens are in general considered more secure than software tokens.

Before using external tokens, plan how the external token is going to be used with the subsystem:

- All system keys for a subsystem must be generated on the same token.
- Preferably, the subsystem keys should be generated in an empty HSM slot. If the HSM slot has previously been used to store other keys, then use the HSM vendor's utilities to delete the contents of the slot. The Certificate System has to be able to create certificates and keys on the slot with default nicknames. If not properly cleaned up, the names of these objects may collide with previous instances.

- A single HSM can be used to store certificates and keys for multiple subsystem instances, which may be installed on multiple hosts. When an HSM is used, any certificate nickname for a subsystem must be unique for every subsystem instance managed on the HSM.

NOTE

For KRA to perform wrapping and unwrapping on an HSM for key archival and recovery, the HSM must have the capability to wrap and unwrap private keys. If the HSM does not have this capability, the KRA has to be downgraded to be able to encrypt and decrypt user private keys for its key archival and recovery functionalities.

IMPORTANT

If multiple subsystems share the same networked HSM, make sure the nicknames and certificate subject distinguished names are uniquely specified to avoid conflict.

2.3.7.4. Smart Cards (Clients)

Smart Cards are managed using the Enterprise Security Client, which is documented in a separate guide that describes its architecture, concepts, and provides information on installation, configuration, and administration. See *Managing Smart Cards with the Enterprise Security Client*.

2.3.8. Certificate System Serial Number Management

2.3.8.1. Serial Number Ranges

Certificate request and serial numbers are represented by Java's big integers

By default, due to their efficiency, certificate request numbers, certificate serial numbers, and replica IDs are assigned sequentially for CA subsystems.

Serial number ranges are specifiable for requests, certificates, and replica IDs:

- Current serial number management is based on assigning ranges of sequential serial numbers.
- Instances request new ranges when crossing below a defined threshold.
- Instances store information about a newly acquired range once it is assigned to the instance.
- Instances continue using old ranges until all numbers are exhausted from it, and then it moves to the new range.
- Cloned subsystems synchronize their range assignment through replication conflicts.

For new clones:

- Part of the current range of the master is transferred to a new clone in the process of cloning.

- New clones may request a new range if the transferred range is below the defined threshold.

All ranges are configurable at CA instance installation time by adding a **[CA]** section to the PKI instance override configuration file, and adding the following **name=value** pairs under that section as needed. Default values which already exist in **/etc/pki/default.cfg** are shown in the following example:

```
[CA]
pki_serial_number_range_start=1
pki_serial_number_range_end=10000000
pki_request_number_range_start=1
pki_request_number_range_end=10000000
pki_replica_number_range_start=1
pki_replica_number_range_end=100
```

2.3.8.2. Random Serial Number Management

In addition to sequential serial number management, Red Hat Certificate System provides optional random serial number management. Using random serial numbers is selectable at CA instance installation time by adding a **[CA]** section to the PKI instance override file and adding the following **name=value** pair under that section:

```
[CA]
pki_random_serial_numbers_enable=True
```

If selected, certificate request numbers and certificate serial numbers will be selected randomly within the specified ranges.

2.3.9. Security Domain

A *security domain* is a registry of PKI services. Services such as CAs register information about themselves in these domains so users of PKI services can find other services by inspecting the registry. The security domain service in RHCS manages both the registration of PKI services for Certificate System subsystems and a set of shared trust policies.

See Section 5.3, “Planning Security Domains” for further details.

2.3.10. Passwords and Watchdog (nuxwdog)

In the default setup, an RHCS subsystem instance needs to act as a client and authenticate to some other services, such as an LDAP internal database (unless SSL client authentication is set up, where a certificate will be used for authentication instead), the NSS token database, or sometimes an HSM, with a password. The administrator is prompted to set up this password at the time of installation configuration. This password is then written to the file ***<instance_dir>*/conf/password.conf**. At the same time, an identifying string is stored in the main configuration file **CS.cfg** as part of the parameter **cms.passwordlist**.

The password entries in the password.conf file are in the following format:

```
<name>=<password>
```

E.g.

```
internal=413691159497
```

In cases where an HSM token is used, the following format is used:

```
hardware-<name>=<password>
```

E.g.

```
hardware-internal=413691159497
```

The configuration file, **CS.cfg**, is protected by Red Hat Enterprise Linux, and only accessible by the PKI administrators. No passwords are stored in **CS.cfg**.

During installation, the installer will select and log into either the internal software token or a hardware cryptographic token. The login passphrase to these tokens is also written to **password.conf**.

Configuration at a later time can also place passwords into **password.conf**. LDAP publishing is one example where the newly configured Directory Manager password for the publishing directory is entered into **password.conf**.

Nuxwdog is a lightweight auxiliary daemon process that is used to start, stop, monitor the status of, and reconfigure server programs. It is most useful when users need to be prompted for passwords to start a server, because it caches these passwords securely in the kernel keyring, so that restarts can be done automatically in the case of a server crash.

Once installation is complete, it is possible to remove the **password.conf** file altogether. On restart, the **nuxwdog** watchdog program will prompt the administrator for the required passwords, using the parameter **cms.passwordlist** as a list of passwords for which to prompt. The passwords are then cached by **nuxwdog** in the kernel keyring to allow automated recovery from a server crash. This automated recovery (automatic subsystem restart) happens in case of uncontrolled shutdown (crash). In case of a controlled shutdown by the administrator, administrators are prompted for passwords again.

When using the watchdog service, starting and stopping an RHCS instance are done differently. For details, see the corresponding section in the *Certificate System Administration Guide*.

2.3.11. Internal LDAP Database

Red Hat Certificate System employs Red Hat Directory Server (RHDS) as its internal database for storing information such as certificates, requests, users, roles, ACLs, as well as other miscellaneous internal information. Certificate System communicates with the internal LDAP database either with a password, or securely by means of SSL authentication.

If certificate-based authentication is required between a Certificate System instance and Directory Server, it is important to follow instruction to set up trust between these two entities. Proper **pkispawn** options will also be needed for installing such Certificate System instance.

Red Hat Certificate System allows installation and configuration of a PKI instance to connect directly with a Directory Server instance that is running in Secure Lightweight Directory Access Protocol (LDAPS). The protocol must be enabled in the Directory Server.

To verify that a client can connect securely over LDAPS, use the following command:

```
# /usr/lib64/mozldap/ldapsearch -Z -h server.example.com -p 636 -D
'cn=Directory Manager' -w password -b "dc=example, dc=com" "objectclass=*"
```

Once you verify the connection, export the self-signed CA certificate using the following command:

```
# certutil -L -d /etc/dirsrv/slapd-pki -n "CA certificate" -a >
$HOME/dscacert.pem
```

Then, add the following parameters into the **[DEFAULT]** section of the PKI instance override configuration file:

```
[DEFAULT]
pki_ds_secure_connection=True
pki_ds_secure_connection_ca_pem_file=$HOME/dscacert.pem
pki_ds_ldaps_port=636
```

Finally, execute the **pkispawn** command to create the PKI subsystem.

For more information, see the **pkispawn** man page.

2.3.12. Security-Enhanced Linux (SELinux)

SELinux is a collection of mandatory access control rules which are enforced across a system to restrict unauthorized access and tampering. SELinux is described in more detail in *Red Hat Enterprise Linux 7 SELinux User's and Administrator's Guide*.

Basically, SELinux identifies *objects* on a system, which can be files, directories, users, processes, sockets, or any other resource on a Linux host. These objects correspond to the Linux API objects. Each object is then mapped to a *security context*, which defines the type of object and how it is allowed to function on the Linux server.

Objects can be grouped into domains, and then each domain is assigned the proper rules. Each security context has rules which set restrictions on what operations it can perform, what resources it can access, and what permissions it has.

SELinux policies for the Certificate System are incorporated into the standard system SELinux policies. These SELinux policies apply to every subsystem and service used by Certificate System. By running Certificate System with SELinux in enforcing mode, the security of the information created and maintained by Certificate System is enhanced.

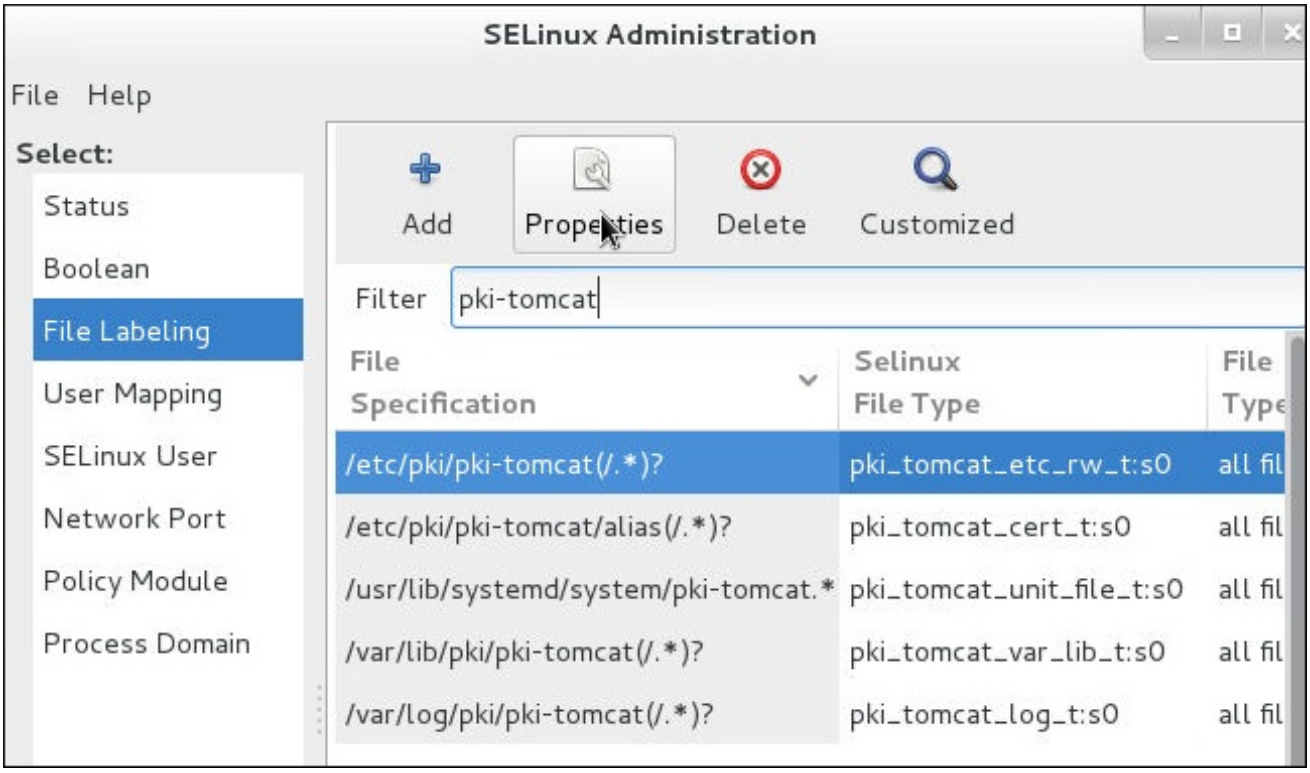


Figure 2.1. CA SELinux Port Policy

The Certificate System SELinux policies define the SELinux configuration for every subsystem instance:

- Files and directories for each subsystem instance are labeled with a specific SELinux context.
- The ports for each subsystem instance are labeled with a specific SELinux context.
- All Certificate System processes are constrained within a subsystem-specific domain.
- Each domain has specific rules that define what actions that are authorized for the domain.
- Any access not specified in the SELinux policy is denied to the Certificate System instance.

For Certificate System, each subsystem is treated as an SELinux object, and each subsystem has unique rules assigned to it. The defined SELinux policies allow Certificate System objects run with SELinux set in enforcing mode.

Every time **pkispawn** is run to configure a Certificate System subsystem, files and ports associated with that subsystem are labeled with the required SELinux contexts. These contexts are removed when the particular subsystems are removed using **pkidestroy**.

The central definition in an SELinux policy is the **pki_tomcat_t** domain. Certificate System instances are Tomcat servers, and the **pki_tomcat_t** domain extends the policies for a standard **tomcat_t** Tomcat domain. All Certificate System instances on a server share the same domain.

When each Certificate System process is started, it initially runs in an unconfined domain (**unconfined_t**) and then transitions into the**pki_tomcat_t** domain. This process then has certain access permissions, such as write access to log files labeled **pki_tomcat_log_t**,

read and write access to configuration files labeled **pki_tomcat_etc_rw_t**, or the ability to open and write to **http_port_t** ports.

The SELinux mode can be changed from enforcing to permissive, or even off, though this is not recommended.

2.3.13. Self-tests

Red Hat Certificate System provides a Self-Test framework which allows the PKI system integrity to be checked during startup or on demand or both. In the event of a non-critical self test failure, the message will be stored in the log file, while in the event of a critical self test failure, the message will be stored in the log file, while the Certificate System subsystem will properly shut down. The administrator is expected to watch the self-test log during the startup of the subsystem if they wish to see the self-test report during startup. They can also view the log after startup.

When a subsystem is shut down due to a self-test failure, it will also be automatically disabled. This is done to ensure that the subsystem does not partially run and produce misleading responses. Once the issue is resolved, the subsystem can be re-enabled by running the following command on the server:

```
# pki-server subsystem-enable <subsystem>
```

2.3.14. Instance Layout

Each Certificate System instance depends on a number of files. Some of them are located in instance-specific folders, while some others are located in a common folder which is shared with other server instances.

For example, the server configuration files are stored in **/etc/pki/*instance_name*/server.xml**, which is instance-specific, but the CA servlets are defined in **/usr/share/pki/ca/webapps/ca/WEB-INF/web.xml**, which is shared by all server instances on the system.

2.3.14.1. File and Directory Locations for Certificate System

Certificate System servers are Tomcat instances which consist of one or more Certificate System subsystems. Certificate System subsystems are web applications that provide specific type of PKI functions. General, shared subsystem information is contained in non-relocatable, RPM-defined shared libraries, Java archive files, binaries, and templates. These are stored in a fixed location.

The directories are instance specific, tied to the instance name. In these examples, the instance name is **pki-tomcat**; the true value is whatever is specified at the time the subsystem is created with **pkispawn**.

The directories contain customized configuration files and templates, profiles, certificate databases, and other files for the subsystem.

Table 2.2. Tomcat Instance Information

Setting	Value
Main Directory	/var/lib/pki/pki-tomcat

Setting	Value
Configuration Directory	/etc/pki/pki-tomcat
Configuration File	/etc/pki/pki-tomcat/server.xml /etc/pki/pki-tomcat/password.conf
Security Databases	/var/lib/pki/pki-tomcat/alias
Subsystem Certificates	SSL server certificate Subsystem certificate [a]
Log Files	/var/log/pki/pki-tomcat
Web Services Files	/usr/share/pki/server/webapps/ROOT - Main page /usr/share/pki/server/webapps/pki/admin - Admin templates /usr/share/pki/server/webapps/pki/js - JavaScript libraries
[a] The subsystem certificate is always issued by the security domain so that domain-level operations that require client authentication are based on this subsystem certificate.	

2.3.14.2. CA Subsystem Information

The directories are instance specific, tied to the instance name. In these examples, the instance name is **pki-tomcat**; the true value is whatever is specified at the time the subsystem is created with **pkispawn**.

Table 2.3. CA Subsystem Information

Setting	Value
Main Directory	/var/lib/pki/pki-tomcat/ca
Configuration Directory	/etc/pki/pki-tomcat/ca
Configuration File	/etc/pki/pki-tomcat/ca/CS.cfg
Subsystem Certificates	CA signing certificate OCSP signing certificate (for the CA's internal OCSP service) Audit log signing certificate
Log Files	/var/log/pki/pki-tomcat/ca
Install Logs	/var/log/pki/pki-ca-spawn.YYYYMMDDhhmmss.log

Setting	Value
Profile Files	/var/lib/pki/pki-tomcat/ca/profiles/ca
Email Notification Templates	/var/lib/pki/pki-tomcat/ca/emails
Web Services Files	/usr/share/pki/ca/webapps/ca/agent - Agent services /usr/share/pki/ca/webapps/ca/admin - Admin services /usr/share/pki/ca/webapps/ca/ee - End user services

2.3.14.3. KRA Subsystem Information

The directories are instance specific, tied to the instance name. In these examples, the instance name is **pki-tomcat**; the true value is whatever is specified at the time the subsystem is created with **pkispawn**.

Table 2.4. KRA Subsystem Information

Setting	Value
Main Directory	/var/lib/pki/pki-tomcat/kra
Configuration Directory	/etc/pki/pki-tomcat/kra
Configuration File	/etc/pki/pki-tomcat/kra/CS.cfg
Subsystem Certificates	Transport certificate Storage certificate Audit log signing certificate
Log Files	/var/log/pki/pki-tomcat/kra
Install Logs	/var/log/pki/pki-kra-spawn.YYYYMMDDhhmmss.log
Web Services Files	/usr/share/pki/kra/webapps/kra/agent - Agent services /usr/share/pki/kra/webapps/kra/admin - Admin services

2.3.14.4. OCSP Subsystem Information

The directories are instance specific, tied to the instance name. In these examples, the instance name is **pki-tomcat**; the true value is whatever is specified at the time the subsystem is created with **pkispawn**.

Table 2.5. OCSP Subsystem Information

Setting	Value
Main Directory	/var/lib/pki/pki-tomcat/ocsp
Configuration Directory	/etc/pki/pki-tomcat/ocsp
Configuration File	/etc/pki/pki-tomcat/ocsp/CS.cfg
Subsystem Certificates	OCSP signing certificate Audit log signing certificate
Log Files	/var/log/pki/pki-tomcat/ocsp
Install Logs	/var/log/pki/pki-ocsp-spawn.YYYYMMDDhhmmss.log
Web Services Files	/usr/share/pki/ocsp/webapps/ocsp/agent - Agent services /usr/share/pki/ocsp/webapps/ocsp/admin - Admin services

2.3.14.5. TKS Subsystem Information

The directories are instance specific, tied to the instance name. In these examples, the instance name is **pki-tomcat**; the true value is whatever is specified at the time the subsystem is created with **pkispawn**.

Table 2.6. TKS Subsystem Information

Setting	Value
Main Directory	/var/lib/pki/pki-tomcat/tks
Configuration Directory	/etc/pki/pki-tomcat/tks
Configuration File	/etc/pki/pki-tomcat/tks/CS.cfg
Subsystem Certificates	Audit log signing certificate
Log Files	/var/log/pki/pki-tomcat/tks
Install Logs	/var/log/pki/pki-tomcat/pki-tks-spawn.YYYYMMDDhhmmss.log

2.3.14.6. TPS Subsystem Information

The directories are instance specific, tied to the instance name. In these examples, the instance name is **pki-tomcat**; the true value is whatever is specified at the time the subsystem is created with **pkispawn**.

Table 2.7. TPS Subsystem Information

Setting	Value
Main Directory	/var/lib/pki/pki-tomcat/tps
Configuration Directory	/etc/pki/pki-tomcat/tps
Configuration File	/etc/pki/pki-tomcat/tps/CS.cfg
Subsystem Certificates	Audit log signing certificate
Log Files	/var/log/pki/pki-tomcat/tps
Install Logs	/var/log/pki/pki-tps-spawn.YYYYMMDDhhhmmss.log
Web Services Files	/usr/share/pki/tps/webapps/tps - TPS services

2.3.14.7. Shared Certificate System Subsystem File Locations

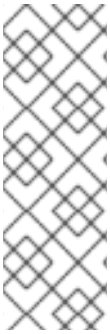
There are some directories used by or common to all Certificate System subsystem instances for general server operations, listed in Table 2.8, “Subsystem File Locations”.

Table 2.8. Subsystem File Locations

Directory Location	Contents
/usr/share/pki	Contains common files and templates used to create Certificate System instances. Along with shared files for all subsystems, there are subsystem-specific files in subfolders: • pki/ca (CA) • pki/kra (KRA) • pki/ocsp (OCSP) • pki/tks (TKS) • pki/tps (TPS)
/usr/bin	Contains the **pkispawn** and **pkidestroy** instance configuration scripts and tools (Java, native, and security) shared by the Certificate System subsystems.
/usr/share/java/pki	Contains Java archive files shared by local Tomcat web applications and shared by the Certificate System subsystems.

2.4. PKI WITH CERTIFICATE SYSTEM

The Certificate System is comprised of subsystems which each contribute different functions of a public key infrastructure. A PKI environment can be customized to fit individual needs by implementing different features and functions for the subsystems.



NOTE

A conventional PKI environment provides the basic framework to manage certificates stored in software databases. This is a *non-TMS* environment, since it does not manage certificates on smart cards. A *TMS* environment manages the certificates on smart cards.

At a minimum, a non-TMS requires only a CA, but a non-TMS environment can use OCSP responders and KRA instances as well.

2.4.1. Issuing Certificates

As stated, the Certificate Manager is the heart of the Certificate System. It manages certificates at every stage, from requests through enrollment (issuing), renewal, and revocation.

The Certificate System supports enrolling and issuing certificates and processing certificate requests from a variety of end entities, such as web browsers, servers, and virtual private network (VPN) clients. Issued certificates conform to X.509 version 3 standards.

2.4.1.1. The Enrollment Process

An end entity enrolls in the PKI environment by submitting an enrollment request through the end-entity interface. There can be many kinds of enrollment that use different enrollment methods or require different authentication methods. Different interfaces can also accept different types of Certificate Signing Requests (CSR).

The Certificate Manager supports different ways to submit CSRs, such as using the graphical interface and command-line tools.

2.4.1.1.1. Enrollment Using the User Interface

For each enrollment through the user interface, there is a separate enrollment page created that is specific to the type of enrollment, type of authentication, and the certificate profiles associated with the type of certificate. The forms associated with enrollment can be customized for both appearance and content. Alternatively, the enrollment process can be customized by creating certificate profiles for each enrollment type. Certificate profiles dynamically-generate forms which are customized by configuring the inputs associated with the certificate profile. Different interfaces can also accept different types of Certificate Signing Requests (CSR).

When an end entity enrolls in a PKI by requesting a certificate, the following events can occur, depending on the configuration of the PKI and the subsystems installed:

1. The end entity provides the information in one of the enrollment forms and submits a request.

   The information gathered from the end entity is customizable in the form depending on the information collected to store in the certificate or to authenticate against the authentication method associated with the form. The form creates a request that is then submitted to the Certificate Manager.

2. The enrollment form triggers the creation of the public and private keys or for dual-key pairs for the request.

3. The end entity provides authentication credentials before submitting the request, depending on the authentication type. This can be LDAP authentication, PIN-based authentication, or certificate-based authentication.

4. The request is submitted either to an agent-approved enrollment process or an automated process.

    - The agent-approved process, which involves no end-entity authentication, sends the request to the request queue in the agent services interface, where an agent must processes the request. An agent can then modify parts of the request, change the status of the request, reject the request, or approve the request.

      Automatic notification can be set up so an email is sent to an agent any time a request appears in the queue. Also, an automated job can be set to send a list of the contents of the queue to agents on a pre configured schedule.

    - The automated process, which involves end-entity authentication, processes the certificate request as soon as the end entity successfully authenticates.

5. The form collects information about the end entity from an LDAP directory when the form is submitted. For certificate profile-based enrollment, the defaults for the form can be used to collect the user LDAP ID and password.

6. The certificate profile associated with the form determine aspects of the certificate that is issued. Depending on the certificate profile, the request is evaluated to determine if the request meets the constraints set, if the required information is provided, and the contents of the new certificate.

7. The form can also request that the user export the private encryption key. If the KRA subsystem is set up with this CA, the end entity's key is requested, and an archival request is sent to the KRA. This process generally requires no interaction from the end entity.

8. The certificate request is either rejected because it did not meet the certificate profile or authentication requirements, or a certificate is issued.

9. The certificate is delivered to the end entity.

    - In automated enrollment, the certificate is delivered to the user immediately. Since the enrollment is normally through an HTML page, the certificate is returned as a response on another HTML page.

    - In agent-approved enrollment, the certificate can be retrieved by serial number or request Id in the end-entity interface.

    - If the notification feature is set up, the link where the certificate can be obtained is sent to the end user.

10. An automatic notice can be sent to the end entity when the certificate is issued or rejected.

11. The new certificate is stored in the Certificate Manager's internal database.

12. If publishing is set up for the Certificate Manager, the certificate is published to a file or an LDAP directory.

13. The internal OCSP service checks the status of certificates in the internal database when a certificate status request is received.

The end-entity interface has a search form for certificates that have been issued and for the CA certificate chain.

By default, the user interface supports CSR in the PKCS #10 and Certificate Request Message Format (CRMF).

2.4.1.1.2. Enrollment Using the Command Line

This section describes the general workflows when enrolling certificates using the command line.

2.4.1.1.2.1. Enrolling with CMC

To enroll a certificate with CMC, proceed as follows:

1. Generate a PKCS #10 or CRMF certificate signing request (CSR) using a utility, such as **PKCS10Client** or **CRMFPopClient**.

   NOTE

   If key archival is enabled in the Key Recovery Agent (KRA), use the **CRMFPopClient** utility with the KRA's transport certificate in Privacy Enhanced Mail (PEM) format set in the **kra.transport** file.

2. Use the **CMCRequest** utility to convert the CSR into a CMC request. The **CMCRequest** utility uses a configuration file as input. This file contains, for example, the path to the CSR and the CSR's format.

   For further details and examples, see the CMCRequest(1) man page.

3. Use the **HttpClient** utility to send the CMC request to the CA. **HttpClient** uses a configuration file with settings, such as the path to the CMC request file and the servlet.

   If the **HttpClient** command succeeds, the utility receives a PKCS #7 chain with CMC status controls from the CA.

   For details about what parameters the utility provides, enter the **HttpClient** command without any parameters.

4. Use the **CMCResponse** utility to check the issuance result of the PKCS #7 file generated by **HttpClient**. If the request is successful, **CMCResponse** displays the certificate chain in a readable format.

   For further details, see the CMCResponse(1) man page.

5. Import the new certificate into the application. For details, follow the instructions of the application to which you want to import the certificate.

NOTE

The certificate retrieved by **HttpClient** is in PKCS #7 format. If the application supports only Base64-encoded certificates, use the **BtoA** utility to convert the certificate.

Additionally, certain applications require a header and footer for certificates in Privacy Enhanced Mail (PEM) format. If these are required, add them manually to the PEM file after you converted the certificate.

2.4.1.1.2.2. CMC Enrollment without POP

In situations when Proof Of Possession (POP) is missing, the **HttpClient** utility receives an **EncryptedPOP** CMC status, which is displayed by the**CMCResponse** command. In this case, enter the **CMCRequest** command again with different parameters in the configuration file.

For details, see the *The CMC Enrollment Process* section in the *Red Hat Certificate System Administration Guide*.

2.4.1.1.2.3. Signed CMC Requests

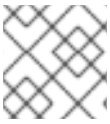
CMC requests can either be signed by a user or a CA agent:

- If an agent signs the request, set the authentication method in the profile to **CMCAuth**.
- If a user signs the request, set the authentication method in the profile to **CMCUserSignedAuth**.

For details, see the *CMC Authentication Plug-ins* section in the *Red Hat Certificate System Administration Guide*.

2.4.1.1.2.4. Unsigned CMC Requests

When the **CMCUserSignedAuth** authentication plug-in is configured in the profile, you must use an unsigned CMC request in combination with the Shared Secret authentication mechanism.



NOTE

Unsigned CMC requests are also called **self-signed CMC requests**.

For details, see the *CMC Authentication Plug-ins* and *The CMC Shared Secret Feature* sections in the *Red Hat Certificate System Administration Guide*.

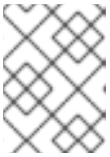
2.4.1.1.2.5. The Shared Secret Workflow

Certificate System provides the Shared Secret authentication mechanism for CMC requests according to RFC 5272. In order to protect the passphrase, an issuance protection certificate must be provided when using the **CMCSharedToken** command. The issuance protection certificate works similar to the KRA transport certificate. For further details, see the CMCSharedToken(1) man page and the*The CMC Shared Secret Feature* section in the *Red Hat Certificate System Administration Guide*.

Shared Secret Create by the End Entity User (Preferred)

The following describes the workflow, if the user generates the shared secret:

1. The end entity user obtains the issuance protection certificate from the CA administrator.
2. The end entity user uses the **CMCSharedToken** utility to generate a shared secret token.



NOTE

The **-p** option sets the passphrase that is shared between the CA and the user, not the password of the token.

3. The end entity user sends the encrypted shared token generated by the **CMCSharedToken** utility to the administrator.
4. The administrator adds the shared token into the **shrTok** attribute in the user's LDAP entry.
5. The end entity user uses the passphrase to set the ***witness.sharedSecret*** parameter in the configuration file passed to the **CMCRequest** utility.

Shared Secret Created by the CA Administrator

The following describes the workflow, if the CA administrator generates the shared secret for a user:

1. The administrator uses the **CMCSharedToken** utility to generate a shared secret token for the user.

NOTE

The **-p** option sets the passphrase that is shared between the CA and the user, not the password of the token.

2. The administrator adds the shared token into the **shrTok** attribute in the user's LDAP entry.
3. The administrator shares the passphrase with the user.
4. The end entity user uses the passphrase to set the ***witness.sharedSecret*** parameter in the configuration file passed to the **CMCRequest** utility.

2.4.1.1.2.6. Simple CMC Requests

Certificate System allows simple CMC requests. However, this process does not support the same level of security requirements as full CMC requests and, therefore, must only be used in a secure environment.

When using simple CMC requests, set the following in the **HttpClient** utility's configuration file:

```
servlet=/ca/ee/ca/profileSubmitCMCSimple?profileId=caECSimpleCMCUserCert
```

2.4.1.1.3. Enrolling Using the `pki` Utility

For details, see:

- The pki-cert(1) man page
- Section 2.7.5, “Command Line Interface (CLI)”

2.4.1.2. Certificate Profiles

The Certificate System uses certificate profiles to configure the content of the certificate, the constraints for issuing the certificate, the enrollment method used, and the input and output forms for that enrollment. A single certificate profile is associated with issuing a particular type of certificate.

A set of certificate profiles is included for the most common certificate types; the profile settings can be modified. Certificate profiles are configured by an administrator, and then sent to the agent services page for agent approval. Once a certificate profile is approved, it is enabled for use. In case of a UI-enrollment, a dynamically-generated HTML form for the certificate profile is used in the end-entities page for certificate enrollment, which calls on the certificate profile. In case of a command line-based enrollment, the certificate profile is called upon to perform the same processing, such as authentication, authorization, input, output, defaults, and constraints. The server verifies that the defaults and constraints set in the certificate profile are met before acting on the request and uses the certificate profile to determine the content of the issued certificate.

The Certificate Manager can issue certificates with any of the following characteristics, depending on the configuration in the profiles and the submitted certificate request:

- Certificates that are X.509 version 3-compliant
- Unicode support for the certificate subject name and issuer name
- Support for empty certificate subject names
- Support for customized subject name components
- Support for customized extensions

By default, the certificate enrollment profiles are stored in ***<instance directory>/ca/profiles/ca*** with names in the format of***<profile id>*****.cfg**. LDAP-based profiles are possible with proper **pkispawn** configuration parameters.

2.4.1.3. Authentication for Certificate Enrollment

Certificate System provides authentication options for certificate enrollment. These include agent-approved enrollment, in which an agent processes the request, and automated enrollment, in which an authentication method is used to authenticate the end entity and then the CA automatically issues a certificate. CMC enrollment is also supported, which automatically processes a request approved by an agent.

2.4.1.4. Dual Key Pairs

The Certificate System supports generating dual key pairs, separate key pairs for signing and encrypting email messages and other data. To support separate key pairs for signing and encrypting data, dual certificates are generated for end entities, and the encryption

keys are archived. If a client makes a certificate request for dual key pairs, the server issues two separate certificates.

2.4.1.5. Cross-Pair Certificates

It is possible to create a trusted relationship between two separate CAs by issuing and storing cross-signed certificates between these two CAs. By using cross-signed certificate pairs, certificates issued outside the organization's PKI can be trusted within the system.

2.4.2. Renewing Certificates

When certificates reach their expiration date, they can either be allowed to lapse, or they can be renewed.

Renewal regenerates a certificate request using the existing key pairs for that certificate, and then resubmits the request to Certificate Manager. The renewed certificate is identical to the original (since it was created from the same profile using the same key material) with one exception — it has a different, later expiration date.

Renewal can make managing certificates and relationships between users and servers much smoother, because the renewed certificate functions precisely as the old one. For user certificates, renewal allows encrypted data to be accessed without any loss.

2.4.3. Publishing Certificates and CRLs

Certificates can be published to files and an LDAP directory, and CRLs to files, an LDAP directory, and an OCSP responder. The publishing framework provides a robust set of tools to publish to all three places and to set rules to define with more detail which types of certificates or CRLs are published where.

2.4.4. Revoking Certificates and Checking Status

End entities can request that their own certificates be revoked. When an end entity makes the request, the certificate has to be presented to the CA. If the certificate and the keys are available, the request is processed and sent to the Certificate Manager, and the certificate is revoked. The Certificate Manager marks the certificate as revoked in its database and adds it to any applicable CRLs.

An agent can revoke any certificate issued by the Certificate Manager by searching for the certificate in the agent services interface and then marking it revoked. Once a certificate is revoked, it is marked revoked in the database and in the publishing directory, if the Certificate is set up for publishing.

If the internal OCSP service has been configured, the service determines the status of certificates by looking them up in the internal database.

Automated notifications can be set to send email messages to end entities when their certificates are revoked by enabling and configuring the certificate revoked notification message.

2.4.4.1. Revoking Certificates

Users can revoke their certificates using:

- The end-entity pages. For details, see the *Certificate Revocation Pages* section in the *Red Hat Certificate System Administration Guide*.
- The **CMCRequest** utility on the command line. For details, see the *Performing a CMC Revocation* section in the *Red Hat Certificate System Administration Guide*.
- The **pki** utility on the command line. For details, see pki-cert(1) man page.

2.4.4.2. Certificate Status

2.4.4.2.1. CRLs

The Certificate System can create certificate revocation lists (CRLs) from a configurable framework which allows user-defined issuing points so a CRL can be created for each issuing point. Delta CRLs can also be created for any issuing point that is defined. CRLs can be issued for each type of certificate, for a specific subset of a type of certificate, or for certificates generated according to a profile or list of profiles. The extensions used and the frequency and intervals when CRLs are published can all be configured.

The Certificate Manager issues X.509-standard CRLs. A CRL can be automatically updated whenever a certificate is revoked or at specified intervals.

2.4.4.2.2. OCSP Services

The Certificate System CA supports the Online Certificate Status Protocol (OCSP) as defined in PKIX standard RFC 2560. The OCSP protocol enables OCSP-compliant applications to determine the state of a certificate, including the revocation status, without having to directly check a CRL published by a CA to the validation authority. The validation authority, which is also called an *OCSP responder*, checks for the application.

1. A CA is set up to issue certificates that include the Authority Information Access extension, which identifies an OCSP responder that can be queried for the status of the certificate.
2. The CA periodically publishes CRLs to an OCSP responder.
3. The OCSP responder maintains the CRL it receives from the CA.
4. An OCSP-compliant client sends requests containing all the information required to identify the certificate to the OCSP responder for verification. The applications determine the location of the OCSP responder from the value of the Authority Information Access extension in the certificate being validated.
5. The OCSP responder determines if the request contains all the information required to process it. If it does not or if it is not enabled for the requested service, a rejection notice is sent. If it does have enough information, it processes the request and sends back a report stating the status of the certificate.

2.4.4.2.2.1. OCSP Response Signing

Every response that the client receives, including a rejection notification, is digitally signed by the responder; the client is expected to verify the signature to ensure that the response came from the responder to which it submitted the request. The key the responder uses to sign the message depends on how the OCSP responder is deployed in a PKI setup. RFC 2560 recommends that the key used to sign the response belong to one of the following:

- The CA that issued the certificate whose status is being checked.
- A responder with a public key trusted by the client. Such a responder is called a *trusted responder*.
- A responder that holds a specially marked certificate issued to it directly by the CA that revokes the certificates and publishes the CRL. Possession of this certificate by a responder indicates that the CA has authorized the responder to issue OCSP responses for certificates revoked by the CA. Such a responder is called a *CA-designated responder* or a *CA-authorized responder*.

The end-entities page of a Certificate Manager includes a form for manually requesting a certificate for the OCSP responder. The default enrollment form includes all the attributes that identify the certificate as an OCSP responder certificate. The required certificate extensions, such as OCSPNoCheck and Extended Key Usage, can be added to the certificate when the certificate request is submitted.

2.4.4.2.2.2. OCSP Responses

The OCSP response that the client receives indicates the current status of the certificate as determined by the OCSP responder. The response could be any of the following:

- *Good or Verified* . Specifies a positive response to the status inquiry, meaning the certificate has not been revoked. It does not necessarily mean that the certificate was issued or that it is within the certificate's validity interval. Response extensions may be used to convey additional information on assertions made by the responder regarding the status of the certificate.
- *Revoked* . Specifies that the certificate has been revoked, either permanently or temporarily.

Based on the status, the client decides whether to validate the certificate.

NOTE

The OCSP responder will never return a response of *Unknown*. The response will always be either *Good* or *Revoked*.

2.4.4.2.2.3. OCSP Services

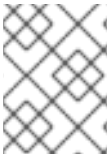
There are two ways to set up OCSP services:

- The OCSP built into the Certificate Manager
- The Online Certificate Status Manager subsystem

In addition to the built-in OCSP service, the Certificate Manager can publish CRLs to an OCSP-compliant validation authority. CAs can be configured to publish CRLs to the Certificate System Online Certificate Status Manager. The Online Certificate Status Manager stores each Certificate Manager's CRL in its internal database and uses the appropriate CRL to verify the revocation status of a certificate when queried by an OCSP-compliant client.

The Certificate Manager can generate and publish CRLs whenever a certificate is revoked and at specified intervals. Because the purpose of an OCSP responder is to facilitate immediate verification of certificates, the Certificate Manager should publish the CRL to the

Online Certificate Status Manager every time a certificate is revoked. Publishing only at intervals means that the OCSP service is checking an outdated CRL.



NOTE

If the CRL is large, the Certificate Manager can take a considerable amount of time to publish the CRL.

The Online Certificate Status Manager stores each Certificate Manager's CRL in its internal database and uses it as the CRL to verify certificates. The Online Certificate Status Manager can also use the CRL published to an LDAP directory, meaning the Certificate Manager does not have to update the CRLs directly to the Online Certificate Status Manager.

2.4.5. Archiving, Recovering, and Rotating Keys

To archive private encryption keys and recover them later, the PKI configuration must include the following elements:

- Clients that can generate dual keys and that support the key archival option (using the CRMF/CMMF protocol).
- An installed and configured KRA.
- HTML forms with which end entities can request dual certificates (based on dual keys) and key recovery agents can request key recovery.

Only keys that are used exclusively for encrypting data should be archived; signing keys in particular should never be archived. Having two copies of a signing key makes it impossible to identify with certainty who used the key; a second archived copy could be used to impersonate the digital identity of the original key owner.

Clients that generate single key pairs use the same private key for both signing and encrypting data, so a private key derived from a single key pair cannot be archived and recovered. Clients that can generate dual key pairs use one private key for encrypting data and the other for signing data. Since the private encryption key is separate, it can be archived.

In addition to generating dual key pairs, the clients must also support archiving the encryption key in certificate requests. This option archives keys at the time the private encryption keys are generated as a part of issuing the certificate.

2.4.5.1. Archiving Keys

The KRA automatically archives private encryption keys if archiving is configured.

If an end entity loses a private encryption key or is unavailable to use the private key, the key must be recovered before any data that was encrypted with the corresponding public key can be read. Recovery is possible if the private key was archived when the key was generated.

There are some common situations when it is necessary to recover encryption keys:

- An employee loses the private encryption key and cannot read encrypted mail messages.

- An employee is on an extended leave, and someone needs to access an encrypted document.
- An employee leaves the company, and company officials need to perform an audit that requires gaining access to the employee's encrypted mail.

The KRA stores private encryption keys in a secure key repository in its internal database; each key is encrypted and stored as a key record and is given a unique key identifier.

The archived copy of the key remains wrapped with the KRA's storage key. It can be decrypted, or unwrapped, only by using the corresponding private key pair of the storage certificate. A combination of one or more key recovery (or KRA) agents' certificates authorizes the KRA to complete the key recovery to retrieve its private storage key and use it to decrypt/recover an archived private key.

The KRA indexes stored keys by key number, owner name, and a hash of the public key, allowing for highly efficient searching. The key recovery agents have the privilege to insert, delete, and search for key records.

- When the key recovery agents search by the key ID, only the key that corresponds to that ID is returned.
- When the agents search by user name, all stored keys belonging to that owner are returned.
- When the agents search by the public key in a certificate, only the corresponding private key is returned.

When a Certificate Manager receives a certificate request that contains the key archival option, it automatically forwards the request to the KRA to archive the encryption key. The private key is encrypted by the transport key, and the KRA receives the encrypted copy and stores the key in its key repository. To archive the key, the KRA uses two special key pairs:

- A transport key pair and corresponding certificate.
- A storage key pair.

Figure 2.2, “How the Key Archival Process Works”illustrates how the key archival process occurs when an end entity requests a certificate.

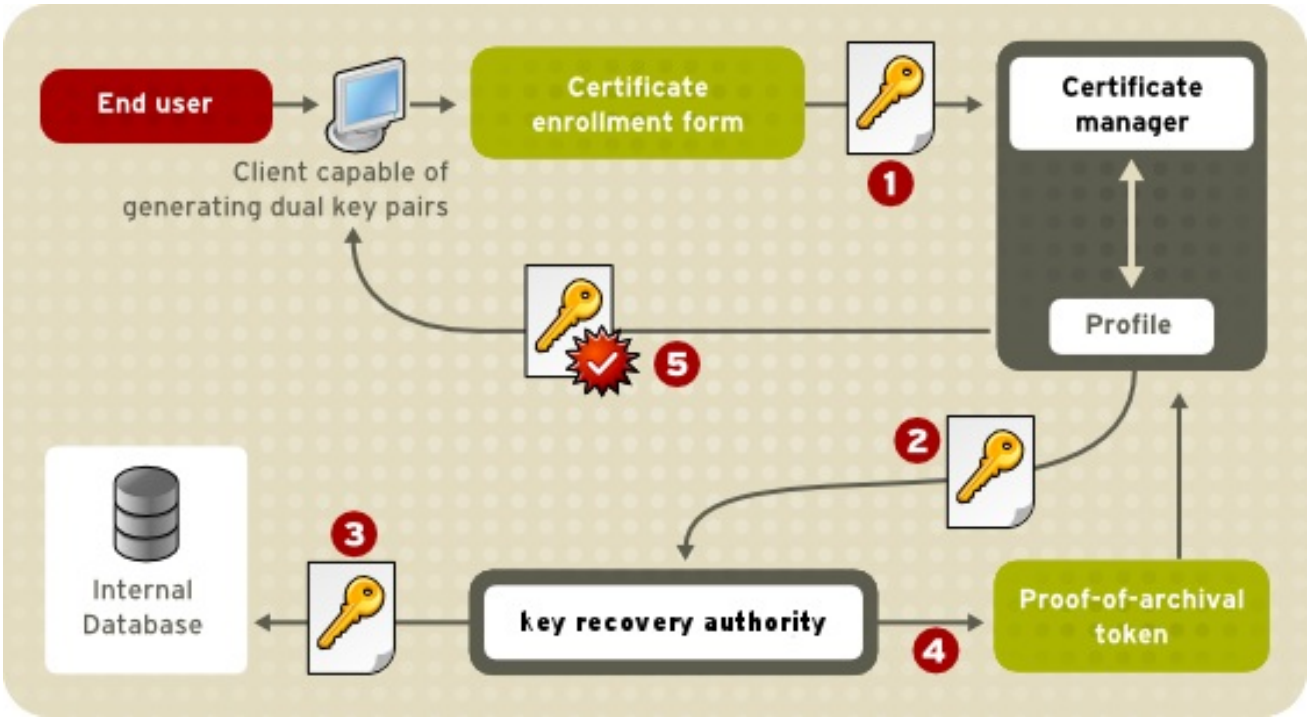


Figure 2.2. How the Key Archival Process Works

1. The client requests and generates a dual key pair.

    1. The end entity, using a client which can generate dual key pairs, submits a request through the Certificate Manager enrollment form.

    2. The client detects the JavaScript in the enrollment form and exports only the private encryption key, not the private signing key.

    3. The Certificate Manager detects the key archival option in the request and asks the client for the private encryption key.

    4. The client encrypts the private encryption key with the public key from the KRA's transport certificate embedded in the enrollment form.

2. After approving the certificate request and issuing the certificate, the Certificate Manager sends it to the KRA for storage, along with the public key. The Certificate Manager waits for verification from the KRA that the private key has been received and stored and that it corresponds to the public encryption key.

3. The KRA decrypts it with the private key. After confirming that the private encryption key corresponds to the public encryption key, the KRA encrypts it again with its public key pair of the storage key before storing it in its internal database.

4. Once the private encryption key has been successfully stored, the KRA uses the private key of its transport key pair to sign a token confirming that the key has been successfully stored; the KRA then sends the token to the Certificate Manager.

5. The Certificate Manager issues two certificates for the signing and encryption key pairs and returns them to the end entity.

Both subsystems subject the request to configured certificate profile constraints at appropriate stages. If the request fails to meet any of the profile constraints, the subsystem rejects the request.

2.4.5.2. Recovering Keys

The KRA supports *agent-initiated key recovery*. Agent-initiated recovery is when designated recovery agents use the key recovery form on the KRA agent services page to process and approve key recovery requests. With the approval of a specified number of agents, an organization can recover keys when the key's owner is unavailable or when keys have been lost.

Through the KRA agent services page, key recovery agents can collectively authorize and retrieve private encryption keys and associated certificates in a PKCS #12 package, which can then be imported into the client.

In key recovery authorization, one of the key recovery agents informs all required recovery agents about an impending key recovery. All recovery agents access the KRA key recovery page. One of the agents initiates the key recovery process. The KRA returns a notification to the agent includes a recovery authorization reference number identifying the particular key recovery request that the agent is required to authorize. Each agent uses the reference number and authorizes key recovery separately.

There are two different paths for key recovery.

- *Asynchronous recovery* means that each step of the recovery process — the initial request and each subsequent approval or rejection — is stored in the KRA's internal database, under the key entry. The data for the recovery process can be retrieved even if the original browser session is closed or the KRA is shut down. Agents search for the key to recover, without using a reference number.
- *Synchronous recovery* means that when the first agent initiates the key recovery process, the process persists and the original browser must remain open until the entire process is complete. When the agent starts the recovery process, the KRA returns a reference number. All subsequent agents use the **`Authorize Recovery`** area and that referral link to access the thread. Continuous updates on the approval status are sent to the initiating agent so they can check the status.

  With synchronous recovery, the page that the first agent used to initiate the key recovery request keeps refreshing until all agents required to authorize have performed the authorization. It is important that the first agent does not close this browser session until the authorization is complete. Otherwise, the key recovery request needs to be started again.

IMPORTANT

The synchronous key recovery mechanism has been deprecated in Red Hat Certificate System 9. Red Hat recommends to use asynchronous key recovery instead.

These two recovery options are illustrated in Figure 2.3, “Async and Sync Recovery, Side by Side”.

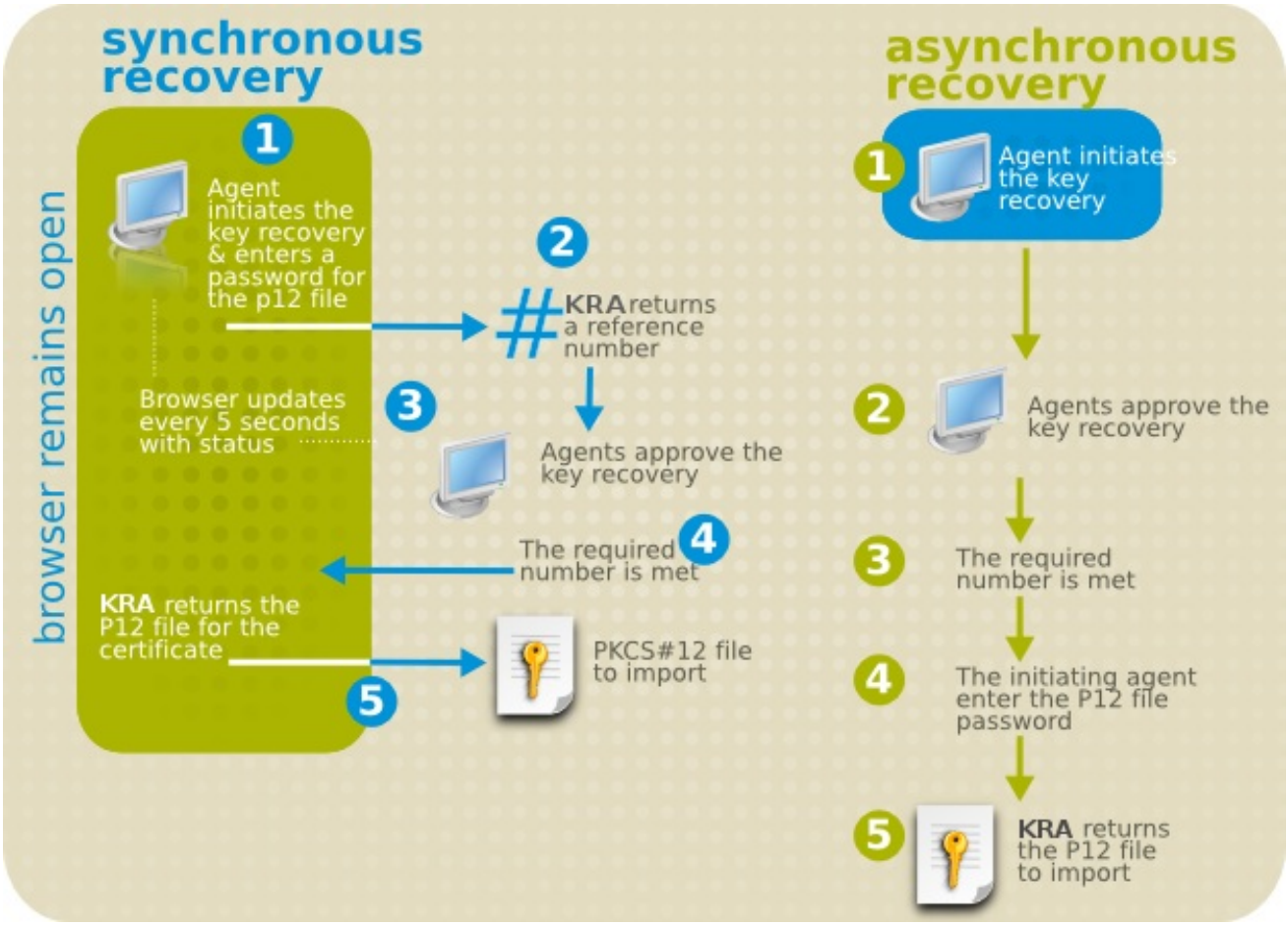


Figure 2.3. Async and Sync Recovery, Side by Side

The KRA informs the agent who initiated the key recovery process of the status of the authorizations.

When all of the authorizations are entered, the KRA checks the information. If the information presented is correct, it retrieves the requested key and returns it along with the corresponding certificate in the form of a PKCS #12 package to the agent who initiated the key recovery process.

WARNING

The PKCS #12 package contains the private key. To minimize the risk of key compromise, the recovery agent must use a secure method to deliver the PKCS #12 package and password to the key recipient. The agent should use a good password to encrypt the PKCS #12 package and set up an appropriate delivery mechanism.

The *key recovery agent scheme* configures the KRA to recognize to which group the key recovery agents belong and specifies how many of these agents are required to authorize a key recovery request before the archived key is restored.

Starting with version 8.1, Certificate System began using an *m-of-n ACL-based recovery scheme*, rather than a secret-splitting-based recovery scheme. In versions of Certificate

System older than 7.1, the password for the storage token was split and protected by individual recovery agent passwords. Now, Certificate System uses its existing access control scheme to ensure recovery agents are appropriately authenticated over SSL/TLS and requires that the agent belong to a specific recovery agent group, by default the Key Recovery Authority Agents Group. The recovery request is executed only when *m-of-n* (a required number of) recovery agents have granted authorization to the request.

IMPORTANT

The above information refers to using a web browser, such as Firefox. However, functionality critical to KRA usage is no longer included in Firefox version 31.6 that was released on Red Hat Enterprise Linux 7 platforms. In such cases, it is necessary to use the **pki** utility to replicate this behavior. For more information, see the pki(1) and pki-key(1) man pages.

Apart from storing symmetric keys, KRA can also store secrets similar to symmetric keys, such as volume encryption secrets, or even passwords and passphrases. The **pki** utility supports options that enable storing and retrieving these other types of secrets.

2.4.5.3. KRA Transport Key Rotation

KRA transport rotation allows for seamless transition between CA and KRA subsystem instances using a current and a new transport key. This allows KRA transport keys to be periodically rotated for enhanced security by allowing both old and new transport keys to operate during the time of the transition; individual subsystem instances take turns being configured while other clones continue to serve with no downtime.

In the KRA transport key rotation process, a new transport key pair is generated, a certificate request is submitted, and a new transport certificate is retrieved. The new transport key pair and certificate have to be included in the KRA configuration to provide support for the second transport key. Once KRA supports two transport keys, administrators can start transitioning CAs to the new transport key. KRA support for the old transport key can be removed once all CAs are moved to the new transport key.

To configure KRA transport key rotation:

1. Generate a new KRA transport key and certificate
2. Transfer the new transport key and certificate to KRA clones
3. Update the CA configuration with the new KRA transport certificate
4. Update the KRA configuration to use only the new transport key and certificate

After this, the rotation of KRA transport certificates is complete, and all the affected CAs and KRAs use the new KRA certificate only. For more information on how to perform the above steps, see the procedures below.

Generating the new KRA transport key and certificate

1. Request the KRA transport certificate.
   a. Stop the KRA:

```
systemctl stop pki-tomcatd@pki-kra.service
```

b. Go to the KRA NSS database directory:

```
cd /etc/pki/pki-kra/alias
```

c. Create a subdirectory and save all the NSS database files into it. For example:

```
mkdir nss_db_backup
cp *.db nss_db_backup
```

d. Create a new request by using the **PKCS10Client** utility. For example:

```
PKCS10Client -p password -d '.' -o 'req.txt' -n 'CN=KRA
Transport 2 Certificate,O=example.com Security Domain'
```

Alternatively, use the **certutil** utility. For example:

```
certutil -d . -R -k rsa -g 2048 -s 'CN=KRA Transport 2
Certificate,O=example.com Security Domain' -f password-file -a
-o transport-certificate-request-file
```

e. Submit the transport certificate request on the **Manual Data Recovery Manager Transport Certificate Enrollment** page of the CA End-Entity page.

f. Wait for the agent approval of the submitted request to retrieve the certificate by checking the request status on the End-Entity retrieval page.

2. Approve the KRA transport certificate through the CA Agent Services interface.

3. Retrieve the KRA transport certificate.

a. Go to the KRA NSS database directory:

```
cd /etc/pki/pki-kra/alias
```

b. Wait for the agent approval of the submitted request to retrieve the certificate by checking the request status on the End-Entity retrieval page.

c. Once the new KRA transport certificate is available, paste its Base64-encoded value into a text file, for example a file named ***cert-serial_number*.txt**. Do not include the header (**-----BEGIN CERTIFICATE-----**) or the footer (**-----END CERTIFICATE-----**).

4. Import the KRA transport certificate.

a. Go to the KRA NSS database directory:

```
cd /etc/pki/pki-kra/alias
```

b. Import the transport certificate into the KRA NSS database:

```
certutil -d . -A -n 'transportCert-serial_number cert-pki-kra
KRA' -t 'u,u,u' -a -i cert-serial_number.txt
```

5. Update the KRA transport certificate configuration.

    a. Go to the KRA NSS database directory:

    ```
    cd /etc/pki/pki-kra/alias
    ```

    b. Verify that the new KRA transport certificate is imported:

    ```
    certutil -d . -L
    certutil -d . -L -n 'transportCert-serial_number cert-pki-kra
    KRA'
    ```

    c. Open the **/var/lib/pki/pki-kra/kra/conf/CS.cfg** file and add the following line:

    ```
    kra.transportUnit.newNickName=transportCert-serial_number cert-
    pki-kra KRA
    ```

Propagating the new transport key and certificate to KRA clones

1. Start the KRA:

    ```
    systemctl start pki-tomcatd@pki-kra.service
    ```

2. Extract the new transport key and certificate for propagation to clones.

    a. Go to the KRA NSS database directory:

    ```
    cd /etc/pki/pki-kra/alias
    ```

    b. Stop the KRA:

    ```
    systemctl stop pki-tomcatd@pki-kra.service
    ```

    c. Verify that the new KRA transport certificate is present:

    ```
    certutil -d . -L
    certutil -d . -L -n 'transportCert-serial_number cert-pki-kra
    KRA'
    ```

    d. Export the KRA new transport key and certificate:

    ```
    pk12util -o transport.p12 -d . -n 'transportCert-serial_number
    cert-pki-kra KRA'
    ```

    e. Verify the exported KRA transport key and certificate:

    ```
    pk12util -l transport.p12
    ```

3. Perform these steps on each KRA clone:

a. Copy the **transport.p12** file, including the transport key and certificate, to the KRA clone location.

b. Go to the clone NSS database directory:

```
cd /etc/pki/pki-kra/alias
```

c. Stop the KRA clone:

```
systemctl stop pki-tomcatd@pki-kra.service
```

d. Check the content of the clone NSS database:

```
certutil -d . -L
```

e. Import the new transport key and certificate of the clone:

```
pk12util -i transport.p12 -d .
```

f. Add the following line to the **/var/lib/pki/pki-kra/kra/conf/CS.cfg** file on the clone:

```
kra.transportUnit.newNickName=transportCert-serial_number cert-
pki-kra KRA
```

g. Start the KRA clone:

```
systemctl start pki-tomcatd@pki-kra.service
```

Updating the CA configuration with the new KRA transport certificate

1. Format the new KRA transport certificate for inclusion in the CA.

   a. Obtain the ***cert-serial_number*.txt** KRA transport certificate file created when retrieving the KRA transport certificate in the previous procedure.

   b. Convert the Base64-encoded certificate included in ***cert-serial_number*.txt** to a single-line file:

```
tr -d '\n' < cert-serial_number.txt > cert-one-line-
serial_number.txt
```

2. Do the following for the CA and all its clones corresponding to the KRA above:

   a. Stop the CA:

```
systemctl stop pki-tomcatd@pki-ca.service
```

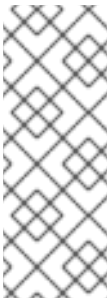
   b. In the **/var/lib/pki/pki-ca/ca/conf/CS.cfg** file, locate the certificate included in the following line:

```
ca.connector.KRA.transportCert=certificate
```

- Replace that certificate with the one contained in ***cert-one-line-serial_number*.txt**.

c. Start the CA:

```
systemctl start pki-tomcatd@pki-ca.service
```



NOTE

While the CA and all its clones are being updated with the new KRA transport certificate, the CA instances that have completed the transition use the new KRA transport certificate, and the CA instances that have not yet been updated continue to use the old KRA transport certificate. Because the corresponding KRA and its clones have already been updated to use both transport certificates, no downtime occurs.

Updating the KRA configuration to use only the new transport key and certificate

For the KRA and each of its clones, do the following:

1. Go to the KRA NSS database directory:

```
cd /etc/pki/pki-kra/alias
```

2. Stop the KRA:

```
systemctl stop pki-tomcatd@pki-kra.service
```

3. Verify that the new KRA transport certificate is imported:

```
certutil -d . -L
certutil -d . -L -n 'transportCert-serial_number cert-pki-kra KRA'
```

4. Open the **/var/lib/pki/pki-kra/kra/conf/CS.cfg** file, and look for the ***nickName*** value included in the following line:

```
kra.transportUnit.nickName=transportCert cert-pki-kra KRA
```

Replace the ***nickName*** value with the ***newNickName*** value included in the following line:

```
kra.transportUnit.newNickName=transportCert-serial_number cert-
pki-kra KRA
```

As a result, the **CS.cfg** file includes this line:

```
kra.transportUnit.nickName=transportCert-serial_number cert-pki-
kra KRA
```

5. Remove the following line from **/var/lib/pki/pki-kra/kra/conf/CS.cfg**:

-

```
kra.transportUnit.newNickName=transportCert-serial_number cert-
pki-kra KRA
```

6. Start the KRA:

```
systemctl start pki-tomcatd@pki-kra.service
```

2.5. SMART CARD TOKEN MANAGEMENT WITH CERTIFICATE SYSTEM

A *smart card* is a hardware cryptographic device containing cryptographic certificates and keys. It can be employed by the user to participate in operations such as secure website access and secure mail. It can also serve as an authentication device to log in to various operating systems such as Red Hat Enterprise Linux. The management of these cards or tokens throughout their entire lifetime in service is accomplished by the *Token Management System* (TMS).

A TMS environment requires a Certificate Authority (CA), Token Key Service (TKS), and Token Processing System (TPS), with an optional Key Recovery Authority (KRA) for server-side key generation and key archival and recovery. Online Certificate Status Protocol (OCSP) can also be used to work with the CA to serve online certificate status requests. This chapter provides an overview of the TKS and TPS systems, which provide the smart card management functions of Red Hat Certificate System, as well as Enterprise Security Client (ESC), that works with TMS from the user end.

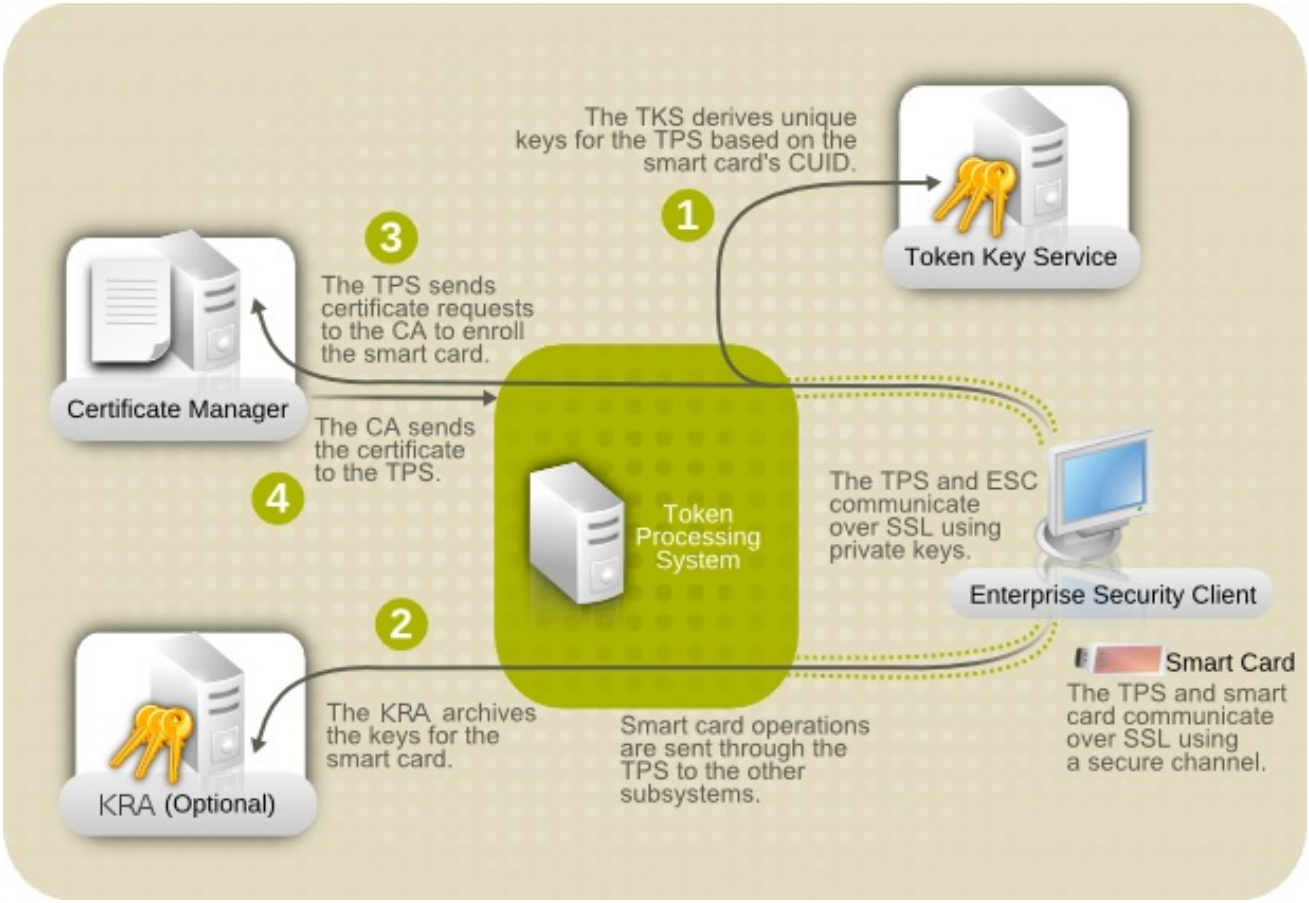


Figure 2.4. How the TMS Manages Smart Cards

2.5.1. Token Key Service (TKS)

The Token Key Service (TKS) is responsible for managing one or more master keys. It maintains the **master keys** and is the only entity within the TMS that has access to the key materials. In an operational environment, each valid smart card token contains a set of symmetric keys that are derived from both the master key and the ID that is unique to the card (CUID).

Initially, a default (unique only per manufacturer master key) set of symmetric keys is initialized on each smart card by the manufacturer. This default set should be changed at the deployment site by going through a **Key Changeover** operation to generate the new master key on TKS. As the sole owner to the master key, when given the CUID of a smart card, TKS is capable of deriving the set of symmetric keys residing on that particular smart card, which would then allow TKS to establish a session-based **Secure Channel** for secure communication between TMS and each individual smart card.

NOTE

Because of the sensitivity of the data that the TKS manages, the TKS should be set behind the firewall with restricted access.

2.5.1.1. Master Keys and Key Sets

The TKS supports multiple smart card key sets. Each smart card vendor creates different default (developer) static key sets for their smart card token stocks, and the TKS is equipped with the static key set (per manufacturer) to kickstart the format process of a blank token.

During the format process of a blank smart card token, a Java applet and the uniquely derived symmetric key set are injected into the token. Each master key (in some cases referred to as **keySet**) that the TKS supports is to have a set of entries in the TKS configuration file (**`CS.cfg`**). Each TPS profile contains a configuration to direct its enrollment to the proper TKS keySet for the matching key derivation process that would essentially be responsible for establishing the Secure Channel secured by a set of session-specific keys between TMS and the smart card token.

On TKS, master keys are defined by named keySets for references by TPS. On TPS, depending on the enrollment type (internal or external registration), The keySet is either specified in the TPS profile, or determined by the keySet Mapping Resolver.

2.5.1.2. Key Ceremony (Shared Key Transport)

A **Key Ceremony** is a process for transporting highly sensitive keys in a secure way from one location to another. In one scenario, in a highly secure deployment environment, the master key can be generated in a secure vault with no network to the outside. Alternatively, an organization might want to have TKS and TPS instances on different physical machines. In either case, under the assumption that no one single person is to be trusted with the key, Red Hat Certificate System TMS provides a utility called **`tkstool`** to manage the secure key transportation.

2.5.1.3. Key Update (Key Changeover)

When Global Platform-compliant smart cards are created at the factory, the manufacturer will burn a set of default symmetric keys onto the token. The TKS is initially configured to use these symmetric keys (one KeySet entry per vendor in the TKS configuration).

However, since these symmetric keys are not unique to the smart cards from the same stock, and because these are well-known keys, it is strongly encouraged to replace these symmetric keys with a set that is unique per token, not shared by the manufacturer, to restrict the set of entities that can manipulate the token.

The changing over of the keys takes place with the assistance of the Token Key Service subsystem. One of the functions of the TKS is to oversee the Master Keys from which the previously discussed smart card token keys are derived. There can be more than one master key residing under the control of the TKS.

IMPORTANT

When this key changeover process is done on a token, the token may become unusable in the future since it no longer has the default key set enabled. The key is essentially only as good as long as the TPS and TKS system that provisioned the token is valid. Because of this, it is essential to keep all the master keys, even if any of them are outdated.

You can disable the old master keys in TKS for better control, but do not delete them unless disabled tokens are part of your plan. There is support to revert the token keys back to the original key set, which is viable if the token is to be reused again in some sort of a testing scenario.

2.5.1.4. APDUs and Secure Channels

The Red Hat Certificate System Token Management System (TMS) supports the *GlobalPlatform* smart card specification, in which the Secure Channel implementation is done with the Token Key System (TKS) managing the master key and the Token Processing System (TPS) communicating with the smart card (tokens) with *Application Protocol Data Units* (APDUs).

There are two types of APDUs:

- **Command APDUs**, sent by the TPS to smart cards
- **Response APDUs**, sent by smart cards to the TPS as response to command APDUs

The initiation of the APDU commands may be triggered when clients take action and connect to the Certificate System server for requests. A secure channel begins with an **InitializeUpdate** APDU sent from TPS to the smart card token, and is fully established with the **ExternalAuthenticate** APDU. Then, both the token and TMS would have established a set of shared secrets, called session keys, which are used to encrypt and authenticate the communication. This authenticated and encrypted communication channel is called Secure Channel.

Because TKS is the only entity that has access to the master key which is capable of deriving the set of unique symmetric on-token smart card keys, the Secure Channel provides the adequately safeguarded communication between TMS and each individual token. Any disconnection of the channel will require reestablishment of new session keys for a new channel.

2.5.2. Token Processing System (TPS)

The Token Processing System (TPS) is a registration authority for smart card certificate enrollment. It acts as a conduit between the user-centered Enterprise Security Client (ESC), which interacts with client side smart card tokens, and the Certificate System back end

subsystems, such as the Certificate Authority (CA) and the Key Recovery Authority (KRA).

In TMS, the TPS is required in order to manage smart cards, as it is the only TMS entity that understands the APDU commands and responses. TPS sends commands to the smart cards to help them generate and store keys and certificates for a specific entity, such as a user or device. Smart card operations go through the TPS and are forwarded to the appropriate subsystem for action, such as the CA to generate certificates or the KRA to generate, archive, or recover keys.

2.5.2.1. Coolkey Applet

Red Hat Certificate System includes the **Coolkey** Java applet, written specifically to run on TMS-supported smart card tokens. The **Coolkey** applet connects to a PKCS#11 module that handles the certificate and key related operations. During a token format operation, this applet is injected onto the smart card token using the Secure Channel protocol, and can be updated per configuration.

2.5.2.2. Token Operations

The TPS in Red Hat Certificate System is available to provision smart cards on the behalf of end users of the smart cards. The Token Processing System provides support for the following major token operations:

- **Token Format** - The format operation is responsible for installing the proper Coolkey applet onto the token. The applet provides a platform where subsequent cryptographic keys and certificates can be later placed.
- **Token Enrollment** - The enrollment operation results in a smart card populated with required cryptographic keys and cryptographic certificates. This material allows the user of the smart card to participate in operations such as secure web site access and secure mail. Two types of enrollments are supported, which is configured globally:
  - **Internal Registration** - Enrollment by TPS profiles determined by the profile **Mapping Resolver**.
  - **External Registration** - Enrollment by TPS profiles determined by the entries in the user’s LDAP record.
- **Token PIN Reset** - The token PIN reset operation allows the user of the token to specify a new PIN that is used to log into the token, making it available for performing cryptographic operations.

The following other operations can be considered supplementary or inherent operations to the main ones listed above. They can be triggered per relevant configuration or by the state of the token.

- **Key Generation** - Each PKI certificate is comprised of a public/private key pair. In Red Hat Certificate System, the generation of the keys can be done in two ways, depending on the **TPS profile** configuration:
  - **Token Side Key Generation** - The PKI key pairs are generated on the smart card token. Generating the key pairs on the token side does *not* allow for key archival.

    - **Server Side Key Generation** - The PKI key pairs are generated on the TMS server side. The key pairs are then sent back to the token using Secure Channel. Generating the key pairs on the server side allows for key archival.

- **Certificate Renewal** - This operation allows a previously enrolled token to have the certificates currently on the token reissued while reusing the same keys. This is useful in situations where the old certificates are due to expire and you want to create new ones but maintain the original key material.

- **Certificate Revocation** - Certificate revocation can be triggered based on TPS profile configuration or based on token state.

  Normally, only the CA which issued a certificate can revoke it, which could mean that retiring a CA would make it impossible to revoke certain certificates. However, it is possible to route revocation requests for tokens to the retired CA while still routing all other requests such as enrollment to a new, active CA. This mechanism is called **Revocation Routing**.

- **Token Key Changeover** - The key changeover operation, triggered by a format operation, results in the ability to change the internal keys of the token from the default developer key set to a new key set controlled by the deployer of the Token Processing System. This is usually done in any real deployment scenario since the developer key set is better suited to testing situations.

- **Applet Update** - During the course of a TMS deployment, the Coolkey smart card applet can be updated or downgraded if required.

2.5.2.3. TPS Profiles

The Certificate System Token Processing System subsystem facilitates the management of smart card tokens. Tokens are provisioned by the TPS such that they are taken from a blank state to either a Formatted or Enrolled condition. A Formatted token is one that contains the **CoolKey** applet supported by TPS, while an Enrolled token is personalized (a process called *binding*) to an individual with the requisite certificates and cryptographic keys. This fully provisioned token is ready to use for crytptographic operations.

The TPS can also manage *Profiles*. The notion of a token Profile is related to:

- The steps taken to Format or Enroll a token.

- The attributes contained within the finished token after the operation has been successfully completed.

The following list contains some of the quantities that make up a unique token profile:

- How does the TPS connect to the user's authentication LDAP database?

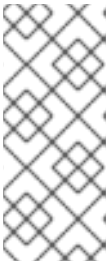
- Will user authentication be required for this token operation? If so, what authentication manager will be used?

- How does the TPS connect to a Certificate System CA from which it will obtain certificates?

- How are the private and public keys generated on this token? Are they generated on the token side or on the server side?

- What key size (in bits) is to be used when generating private and public keys?

- Which certificate enrollment profile (provisioned by the CA) is to be used to generate the certificates on this token?



NOTE

This setting will determine the final structure of the certificates to be written to the token. Different certificates can be created for different uses, based on extensions included in the certificate. For example, one certificate can specialize in data encryption, and another one can be used for signature operations.

- What version of the Coolkey applet will be required on the token?
- How many certificates will be placed on this token for an enrollment operation?

These above and many others can be configured for each token type or profile. A full list of available configuration options is available in the *Red Hat Certificate System Administration Guide*.

Another question to consider is how a given token being provisioned by a user will be mapped to an individual token profile. There are two types of registration:

- **Internal Registration** - In this case, the TPS profile (`tokenType`) is determined by the profile *Mapping Resolver*. This filter-based resolver can be configured to take any of the data provided by the token into account and determine the target profile.
- **External Registration** - When using external registration, the profile (in name only - actual profiles are still defined in the TPS in the same fashion as those used by the internal registration) is specified in each user's LDAP record, which is obtained during authentication. This allows the TPS to obtain key enrollment and recovery information from an external registration Directory Server where user information is stored. This gives you the control to override the enrollment, revocation, and recovery policies that are inherent to the TPS internal registration mechanism. The user LDAP record attribute names relevant to external registration are configurable.

  External registration can be useful when the concept of a "group certificate" is required. In that case, all users within a group can have a special record configured in their LDAP profiles for downloading a shared certificate and keys.

The registration to be used is configured globally per TPS instance.

2.5.2.4. Token Database

The Token Processing System makes use of the LDAP token database store, which is used to keep a list of active tokens and their respective certificates, and to keep track of the current state of each token. A brand new token is considered *Uninitialized*, while a fully enrolled token is *Enrolled*. This data store is constantly updated and consulted by the TPS when processing tokens.

2.5.2.4.1. Token States and Transitions

The Token Processing System stores states in its internal database in order to determine the current token status as well as actions which can be performed on the token.

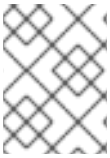
2.5.2.4.1.1. Token States

The following table lists all possible token states:

Table 2.9. Possible Token States

Name	Code	Label
FORMATTED	0	Formatted (uninitialized)
DAMAGED	1	Physically damaged
PERM_LOST	2	Permanently lost
SUSPENDED	3	Suspended (temporarily lost)
ACTIVE	4	Active
TERMINATED	6	Terminated
UNFORMATTED	7	Unformatted

The command line interface displays token states using the Name listed above. The graphical interface uses the Label instead.



NOTE

The above table contains no state with code **5**, which previously belonged to a state that was removed.

2.5.2.4.1.2. Token State Transitions Done Using the Graphical or Command Line Interface

Each token state has a limited amount of next states it can transition into. For example, a token can change state from **FORMATTED** to **ACTIVE** or **DAMAGED**, but it can never transition from **FORMATTED** to **UNFORMATTED**.

Furthermore, the list of states a token can transition into is different depending on whether the transition is triggered manually using a command line or the graphical interface, or automatically using a token operation. The list of allowed manual transitions is stored in the **tokendb.allowedTransitions** property, and the **tps.operations.allowedTransitions** property controls allowed transitions triggered by token operations.

The default configurations for both manual and token operation-based transitions are stored in the **/usr/share/pki/tps/conf/CS.cfg** configuration file.

2.5.2.4.1.2.1. Token State Transitions Using the Command Line or Graphical Interface

All possible transitions allowed in the command line or graphical interface are described in the TPS configuration file using the **tokendb.allowedTransitions** property:

```
tokendb.allowedTransitions=0:1,0:2,0:3,0:6,3:2,3:6,4:1,4:2,4:3,4:6,6:7
```

-

The property contains a comma-separated list of transitions. Each transition is written in the format of ***<current code>:<new code>***. The codes are described inTable 2.9, “Possible Token States”. The default configuration is preserved in **/usr/share/pki/tps/conf/CS.cfg**.

The following table describes each possible transition in more detail:

Table 2.10. Possible Manual Token State Transitions

Transition	Current State	Next State	Description
0:1	FORMATTED	DAMAGED	This token has been physically damaged.
0:2	FORMATTED	PERM_LOST	This token has been permanently lost.
0:3	FORMATTED	SUSPENDED	This token has been suspended (temporarily lost).
0:6	FORMATTED	TERMINATED	This token has been terminated.
3:2	SUSPENDED	PERM_LOST	This suspended token has been permanently lost.
3:6	SUSPENDED	TERMINATED	This suspended token has been terminated.
4:1	ACTIVE	DAMAGED	This token has been physically damaged.
4:2	ACTIVE	PERM_LOST	This token has been permanently lost.
4:3	ACTIVE	SUSPENDED	This token has been suspended (temporarily lost).
4:6	ACTIVE	TERMINATED	This token has been terminated.
6:7	TERMINATED	UNFORMATTED	Reuse this token.

The following transitions are generated automatically depending on the token's original state. If a token was originally **FORMATTED** and then became **SUSPENDED**, it can only return to the **FORMATTED** state. If a token was originally **ACTIVE** and then became **SUSPENDED**, it can only return to the **ACTIVE** state.

Table 2.11. Token State Transitions Triggered Automatically

Transition	Current State	Next State	Description
3:0	SUSPENDED	FORMATTED	This suspended (temporarily lost) token has been found.
3:4	SUSPENDED	ACTIVE	This suspended (temporarily lost) token has been found.

2.5.2.4.1.3. Token State Transitions using Token Operations

All possible transitions that can be done using token operations are described in the TPS configuration file using the **`tokendb.allowedTransitions`** property:

```
tps.operations.allowedTransitions=0:0,0:4,4:4,4:0,7:0
```

The property contains a comma-separated list of transitions. Each transition is written in the format of ***<current code>:<new code>***. The codes are described inTable 2.9, “Possible Token States”. The default configuration is preserved in **`/usr/share/pki/tps/conf/CS.cfg`**.

The following table describes each possible transition in more detail:

Table 2.12. Possible Token State Transitions using Token Operations

Transition	Current State	Next State	Description
0:0	FORMATTED	FORMATTED	This allows reformatting a token or upgrading applet/key in a token.
0:4	FORMATTED	ACTIVE	This allows enrolling a token.
4:4	ACTIVE	ACTIVE	This allows re-enrolling an active token. May be useful for external registration.
4:0	ACTIVE	FORMATTED	This allows formatting an active token.

Transition	Current State	Next State	Description
7:0	UNFORMATTED	FORMATTED	This allows formatting a blank or previously used token.

2.5.2.4.1.4. Token State and Transition Labels

The default labels for token states and transitions are stored in the **/usr/share/pki/tps/conf/token-states.properties** configuration file. By default, the file has the following contents:

```
# Token states
UNFORMATTED         = Unformatted
FORMATTED           = Formatted (uninitialized)
ACTIVE              = Active
SUSPENDED           = Suspended (temporarily lost)
PERM_LOST           = Permanently lost
DAMAGED             = Physically damaged
TEMP_LOST_PERM_LOST = Temporarily lost then permanently lost
TERMINATED          = Terminated

# Token state transitions
FORMATTED.DAMAGED        = This token has been physically damaged.
FORMATTED.PERM_LOST      = This token has been permanently lost.
FORMATTED.SUSPENDED      = This token has been suspended (temporarily
lost).
FORMATTED.TERMINATED     = This token has been terminated.
SUSPENDED.ACTIVE         = This suspended (temporarily lost) token has
been found.
SUSPENDED.PERM_LOST      = This suspended (temporarily lost) token has
become permanently lost.
SUSPENDED.TERMINATED     = This suspended (temporarily lost) token has
been terminated.
SUSPENDED.FORMATTED      = This suspended (temporarily lost) token has
been found.
ACTIVE.DAMAGED           = This token has been physically damaged.
ACTIVE.PERM_LOST         = This token has been permanently lost.
ACTIVE.SUSPENDED         = This token has been suspended (temporarily
lost).
ACTIVE.TERMINATED        = This token has been terminated.
TERMINATED.UNFORMATTED   = Reuse this token.
```

2.5.2.4.1.5. Customizing Allowed Token State Transitions

To customize the list of token state transition, edit the following properties in **/var/lib/pki/*instance_name*/*tps*/*conf*/*CS.cfg***:

- **tokendb.allowedTransitions** to customize the list of allowed transitions performed using the command line or graphical interface
- **tps.operations.allowedTransitions** to customize the list of allowed transitions using token operations

Transitions can be removed from the default list if necessary, but new transitions cannot be added unless they were in the default list. The defaults are stored in **/usr/share/pki/tps/conf/CS.cfg**.

2.5.2.4.1.6. Customizing Token State and Transition Labels

To customize token state and transition labels, copy the default **/usr/share/pki/tps/conf/token-states.properties** into your instance folder (**/var/lib/pki/*instance_name*/tps/conf/CS.cfg**), and change the labels listed inside as needed.

Changes will be effective immediately, the server does not need to be restarted. The TPS user interface may require a reload.

To revert to default state and label names, delete the edited **token-states.properties** file from your instance folder.

2.5.2.4.1.7. Token Activity Log

Certain TPS activities are logged. Possible events in the log file are listed in the table below.

Table 2.13. TPS Activity Log Events

Activity	Description
add	A token was added.
format	A token was formatted.
enrollment	A token was enrolled.
recovery	A token was recovered.
renewal	A token was renewed.
pin_reset	A token PIN was reset.
token_status_change	A token status was changed using the command line or graphical interface.
token_modify	A token was modified.
delete	A token was deleted.
cert_revocation	A token certificate was revoked.
cert_unrevocation	A token certificate was unrevoked.

2.5.2.4.2. Token Policies

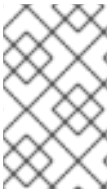
In case of internal registration, each token can be governed by a set of token policies. The default policies are:

```
RE_ENROLL=YES;RENEW=NO;FORCE_FORMAT=NO;PIN_RESET=NO;RESET_PIN_RESET_TO_NO=
NO;RENEW_KEEP_OLD_ENC_CERTS=YES
```

All TPS operations under internal registration are subject to the policies specified in the token's record. If no policies are specified for a token, the TPS uses the default set of policies.

2.5.2.5. Mapping Resolver

The *Mapping Resolver* is an extensible mechanism used by the TPS to determine which token profile to assign to a specific token based on configurable criteria. Each mapping resolver instance can be uniquely defined in the configuration, and each operation can point to various defined mapping resolver instance.



NOTE

The mapping resolver framework provides a platform for writing custom plug-ins. However instructions on how to write a plug-in is outside the scope of this document.

FilterMappingResolver is the only mapping resolver implementation provided with the TPS by default. It allows you to define a set of *mappings* and a target result for each mapping. Each mapping contains a set of filters, where:

- If the input filter parameters pass *all* filters within a mapping, the**target** value is assigned.
- If the input parameters fail a filter, that mapping is skipped and the next one in order is tried.
- If a filter has no specified value, it always passes.
- If a filter does have a specified value, then the input parameters must match exactly.
- The order in which mappings are defined is important. The first mapping which passes is considered resolved and is returned to the caller.

The input filter parameters are information received from the smart card token with or without extensions. They are run against the **FilterMappingResolver** according to the above rules. The following input filter parameters are supported by **FilterMappingResolver**:

- **appletMajorVersion** - The major version of the Coolkey applet on the token.
- **appletMinorVersion** - The minor version of the Coolkey applet on the token.
- **keySet** or **tokenType**
  - **keySet** - can be set as an extension in the client request. Must match the value in the filter if the extension is specified. The keySet mapping resolver is meant for determining keySet value when using external registration. The Key Set

Mapping Resolver is necessary in the external registration environment when multiple key sets are supported (for example, different smart card token vendors). The keySet value is needed for identifying the master key on TKS, which is crucial for establishing Secure Channel. When a user's LDAP record is populated with a set tokenType (TPS profile), it does not know which card will end up doing the enrollment, and therefore keySet cannot be predetermined. The **keySetMappingResolver** helps solve the issue by allowing the keySet to be resolved before authentication.

    - **tokenType** - okenType can be set as an extension in the client request. It must match the value in the filter if the extension is specified. tokenType (also referred to as TPS Profile) is determined at this time for the internal registration environment.

- **tokenATR** - The token's Answer to Reset (ATR).
- **tokenCUID** - "start" and "end" define the range the Card Unique IDs (CUID) of the token must fall in to pass this filter.

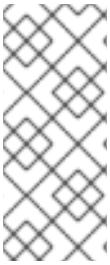
2.5.2.6. TPS Roles

The TPS supports the following roles by default:

- **TPS Administrator** - this role is allowed to:
    - Manage TPS tokens
    - View TPS certificates and activities
    - Manage TPS users and groups
    - Change general TPS configuration
    - Manage TPS authenticators and connectors
    - Configure TPS profiles and profile mappings
    - Configure TPS audit logging
- **TPS Agent** - this role is allowed to:
    - Configure TPS tokens
    - View TPS certificates and activities
    - Change the status of TPS profiles
- **TPS Operator** - this role is allowed to:
    - View TPS tokens, certificates, and activities

2.5.3. TKS/TPS Shared Secret

During TMS installation, a shared symmetric key is established between the Token Key Service and the Token Processing System. The purpose of this key is to wrap and unwrap session keys which are essential to Secure Channels.



NOTE

The shared secret key is currently only kept in a software cryptographical database. There are plans to support keeping the key on a Hardware Security Module (HSM) devices in a future release of Red Hat Certificate System. Once this functionality is implemented, you will be instructed to run a Key Ceremony using **tkstool** to transfer the key to the HSM.

2.5.4. Enterprise Security Client (ESC)

The *Enterprise Security Client* is an HTTP client application, similar to a web browser, that communicates with the TPS and handles smart card tokens from the client side. While an HTTPS connection is established between the ESC and the TPS, an underlying Secure Channel is also established between the token and the TMS within each TLS session.

2.6. RED HAT CERTIFICATE SYSTEM SERVICES

Certificate System has a number of different features for administrators to use which makes it easier to maintain the individual subsystems and the PKI as a whole.

2.6.1. Notifications

When a particular event occurs, such as when a certificate is issued or revoked, then a notification can be sent directly to a specified email address. The notification framework comes with default modules that can be enabled and configured.

2.6.2. Jobs

Automated jobs run at defined intervals.

2.6.3. Logging

The Certificate System and each subsystem produce extensive system and error logs that record system events so that the systems can be monitored and debugged. All log records are stored in the local file system for quick retrieval. Logs are configurable, so logs can be created for specific types of events and at the required logging level.

Certificate System allows logs to be signed digitally before archiving them or distributing them for auditing. This feature enables log files to be checked for tampering after being signed.

2.6.4. Auditing

The Certificate System maintains audit logs for all events, such as requesting, issuing and revoking certificates and publishing CRLs. These logs are then signed. This allows authorized access or activity to be detected. An outside auditor can then audit the system if required. The assigned auditor user account is the only account which can view the signed audit logs. This user's certificate is used to sign and encrypt the logs. Audit logging is configured to specify the events that are logged.

2.6.5. Self-Tests

The Certificate System provides the framework for system self-tests that are automatically run at startup and can be run on demand. A set of configurable self-tests are already

included with the Certificate System.

2.6.6. Users, Authorization, and Access Controls

Certificate System users can be assigned to groups, and they then have the privileges of whichever group they are members. A user only has privileges for the instance of the subsystem in which the user is created and the privileges of the group to which the user is a member.

Authentication is the means that Certificate System subsystems use to verify the identity of clients, whether they are authenticating to a certificate profile or to one of the services interfaces. There are a number of different ways that a client can authentication, including simple user name/password, SSL/TLS client authentication, LDAP authentication, NIS authentication, or CMC. Authentication can be performed for any access to the subsystem; for certificate enrollments, for example, the profile defines how the requestor authenticates to the CA.

Once the client is identified and authenticated, then the subsystems perform an *authorization* check to determine what level of access to the subsystem that particular user is allowed.

Authorization is tied to group, or role, permissions, rather than directly to individual users. The Certificate System provides an authorization framework for creating groups and assigning access control to those groups. The default access control on preexisting groups can be modified, and access control can be assigned to individual users and IP addresses. Access points for authorization have been created for the major portions of the system, and access control rules can be set for each point.

The Certificate System is configured by default with three user types with different access levels to the system:

- *Administrators*, who can perform any administrative or configuration task for a subsystem.
- *Agents*, who perform PKI management tasks, like approving certificate requests, managing token enrollments, or recovering keys.
- *Auditors*, who can view and configure audit logs.

Additionally, when a security domain is created, the CA subsystem which hosts the domain is automatically granted the role of *Security Domain Administrator*, which gives the subsystem the ability to manage the security domain and the subsystem instances within it. Other security domain administrator roles can be created for the different subsystem instances.

2.7. RED HAT CERTIFICATE SYSTEM USER INTERFACES

There are three different interfaces for managing certificates and subsystems, depending on the user type: administrators, agents, and end users.

2.7.1. Administrative Consoles

The administrative interface is used to manage the subsystem itself. This includes adding users, configuring logs, managing profiles and plug-ins, and the internal database, among many other functions. This interface is also the only interface that does not directly deal with certificates, tokens, or keys, meaning it is not used for managing the *PKI*, only the

servers.

There are two types of administrative consoles, Java-based and HTML-based. Although the interface is different, both are accessed using a server URL and the administrative port number.

2.7.1.1. The Java Administrative Console for CA, OCSP, KRA, and TKS Subsystems

The Java console is used by four subsystems: the CA, OCSP, KRA, and TKS. The console is accessed using a locally-installed **pkiconsole** utility. It can access any subsystem because the command requires the host name, the subsystem's administrative SSL/TLS port, and the specific subsystem type.

```
pkiconsole https://server.example.com:admin_port/subsystem_type
```

This opens a console, as in Figure 2.5, “Certificate System Console”.

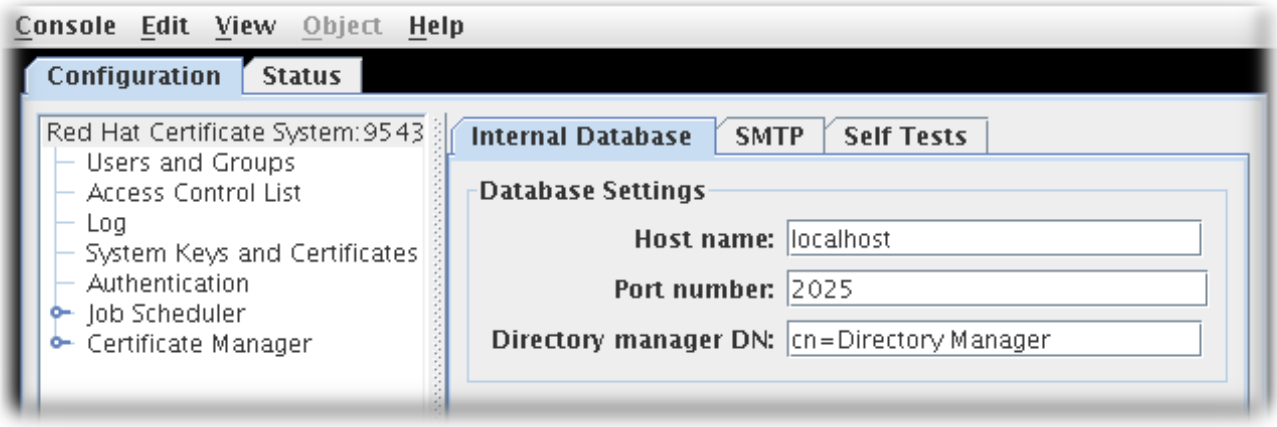


Figure 2.5. Certificate System Console

The **Configuration** tab controls all of the setup for the subsystem, as the name implies. The choices available in this tab are different depending on which subsystem type the instance is; the CA has the most options since it has additional configuration for jobs, notifications, and certificate enrollment authentication.

All subsystems have four basic options:

- Users and groups
- Access control lists
- Log configuration
- Subsystem certificates (meaning the certificates issued to the subsystem for use, for example, in the security domain or audit signing)

The **Status** tab shows the logs maintained by the subsystem.

2.7.1.2. The Administrative Interface for TPS

The TPS subsystems use HTML-based administrative interface. It is accessed by entering the host name and secure port as the URL, authenticating with the administrator's certificate, and clicking the appropriate **`Administrators`** link.

NOTE

There is a single SSL/TLS port for TPS subsystems which is used for both administrator and agent services. Access to those services is restricted by certificate-based authentication.

The HTML admin interface is much more limited than the Java console; the primary administrative function is managing the subsystem users; all other administrative tasks are done by manually editing the **`CS.cfg`** file.

The TPS only allows operations to manage users for the TPS subsystem. However, the TPS admin page can also list tokens and display all activities (including normally-hidden administrative actions) performed on the TPS.

Figure 2.6. TPS Admin Page

2.7.2. Agent Interfaces

The agent services pages are where almost all of the certificate and token management tasks are performed. These services are HTML-based, and agents authenticate to the site using a special agent certificate.

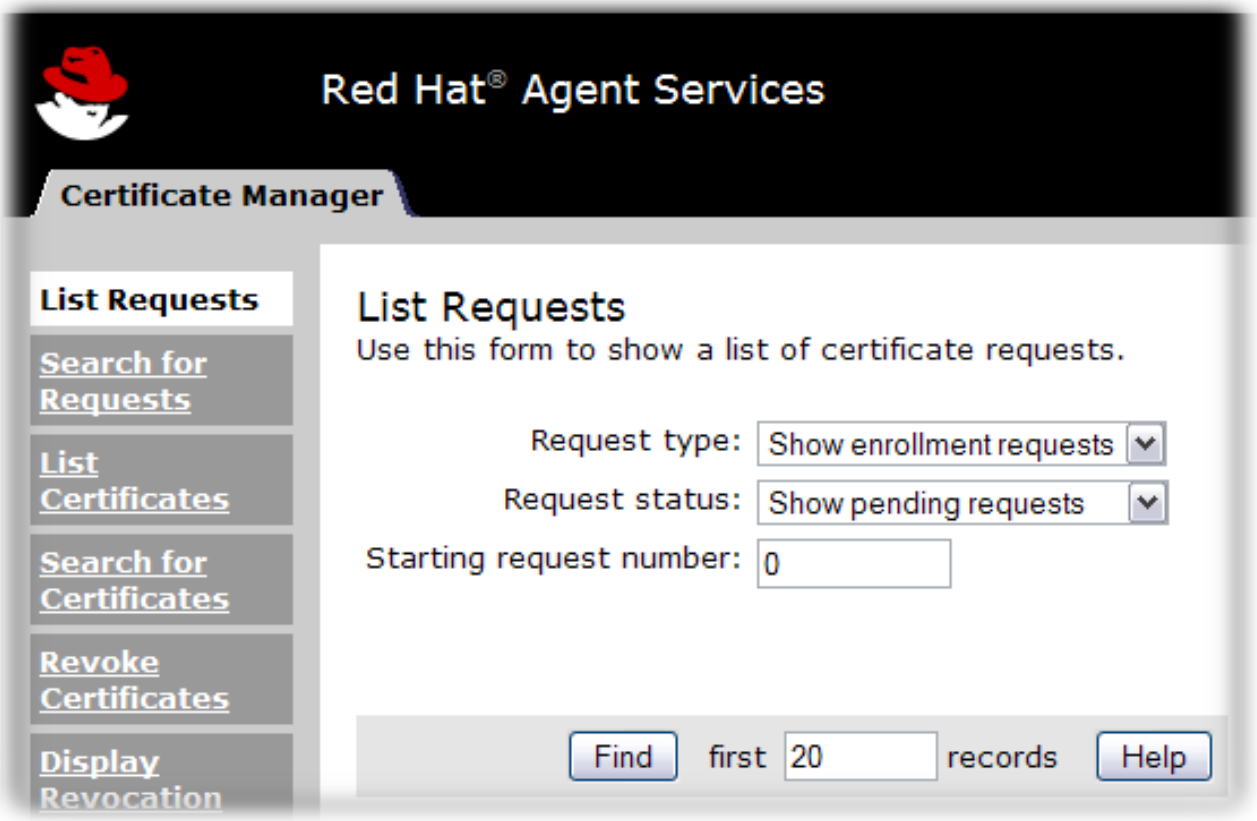


Figure 2.7. Certificate Manager's Agent Services Page

The operations vary depending on the subsystem:

- The Certificate Manager agent services include approving certificate requests (which issues the certificates), revoking certificates, and publishing certificates and CRLs. All certificates issued by the CA can be managed through its agent services page.
- The TPS agent services, like the CA agent services, manages all of the tokens which have been formatted and have had certificates issued to them through the TPS. Tokens can be enrolled, suspended, and deleted by agents. Two other roles (operator and admin) can view tokens in web services pages, but cannot perform any actions on the tokens.
- KRA agent services pages process key recovery requests, which set whether to allow a certificate to be issued reusing an existing key pair if the certificate is lost.
- The OCSP agent services page allows agents to configure CAs which publish CRLs to the OCSP, to load CRLs to the OCSP manually, and to view the state of client OCSP requests.

The TKS is the only subsystem without an agent services page.

2.7.3. End User Pages

The CA and TPS both process direct user requests in some way. That means that end users

have to have a way to connect with those subsystems. The CA has end-user, or *end-entities*, HTML services. The TPS uses the Enterprise Security Client.

The end-user services are accessed over standard HTTP using the server's host name and the standard port number; they can also be accessed over HTTPS using the server's host name and the specific end-entities SSL/TLS port.

For CAs, each type of SSL/TLS certificate is processed through a specific online submission form, called a *profile*. There are about two dozen certificate profiles for the CA, covering all sorts of certificates — user SSL/TLS certificates, server SSL/TLS certificates, log and file signing certificates, email certificates, and every kind of subsystem certificate. There can also be custom profiles.

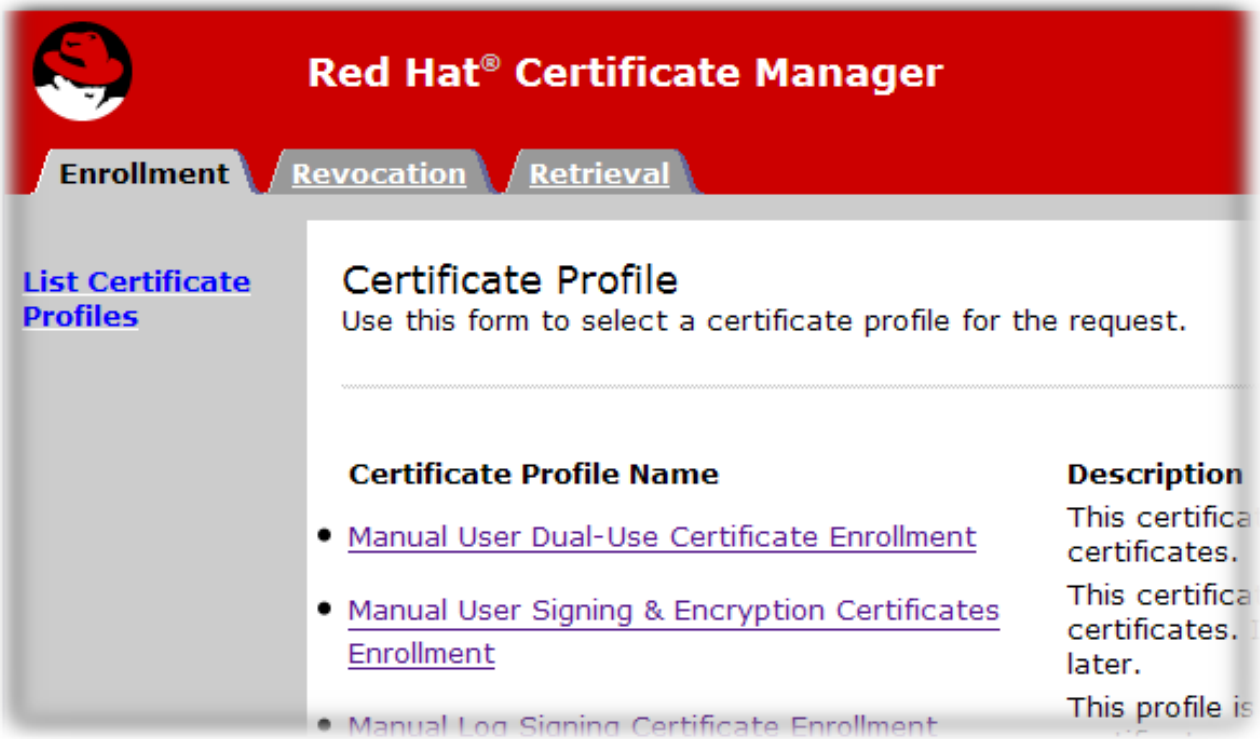


Figure 2.8. Certificate Manager's End-Entities Page

End users retrieve their certificates through the CA pages when the certificates are issued. They can also download CA chains and CRLs and can revoke or renew their certificates through those pages.

2.7.4. Enterprise Security Client

The **Enterprise Security Client** is a tool for Red Hat Certificate System which simplifies managing smart cards. End users can use security tokens (smart cards) to store user certificates used for applications such as single sign-on access and client authentication. End users are issued the tokens containing certificates and keys required for signing, encryption, and other cryptographic functions.

The **Enterprise Security Client** is the third part of Certificate System's complete token management system. Two subsystems — the Token Key Service (TKS) and Token Processing System (TPS) — are used to process token-related operations. The **Enterprise Security Client** is the interface which allows the smart card and user to access the token management system.

After a token is enrolled, applications such as Mozilla Firefox and Thunderbird can be

configured to recognize the token and use it for security operations, like client authentication and S/MIME mail. Enterprise Security Client provides the following capabilities:

- Supports JavaCard 2.1 or higher cards and Global Platform 2.01-compliant smart cards like Safenet's 330J smart card.
- Supports Gemalto TOP IM FIPS CY2 tokens, both the smart card and GemPCKey USB form factor key.
- Supports SafeNet Smart Card 650 (SC650).
- Enrolls security tokens so they are recognized by TPS.
- Maintains the security token, such as re-enrolling a token with TPS.
- Provides information about the current status of the token or tokens being managed.
- Supports server-side key generation so that keys can be archived and recovered on a separate token if a token is lost.

The Enterprise Security Client is a client for end users to register and manage keys and certificates on smart cards or tokens. This is the final component in the Certificate System token management system, with the Token Processing System (TPS) and Token Key Service (TKS).

The Enterprise Security Client provides the user interface of the token management system. The end user can be issued security tokens containing certificates and keys required for signing, encryption, and other cryptographic functions. To use the tokens, the TPS must be able to recognize and communicate with them. Enterprise Security Client is the method for the tokens to be enrolled.

Enterprise Security Client communicates over an SSL/TLS HTTP channel to the back end of the TPS. It is based on an extensible Mozilla XULRunner framework for the user interface, while retaining a legacy web browser container for a simple HTML-based UI.

After a token is properly enrolled, web browsers can be configured to recognize the token and use it for security operations. Enterprise Security Client provides the following capabilities:

- Allows the user to enroll security tokens so they are recognized by the TPS.
- Allows the user to maintain the security token. For example, Enterprise Security Client makes it possible to re-enroll a token with the TPS.
- Provides support for several different kinds of tokens through default and custom token profiles. By default, the TPS can automatically enroll user keys, device keys, and security officer keys; additional profiles can be added so that tokens for different uses (recognized by attributes such as the token CUID) can automatically be enrolled according to the appropriate profile.
- Provides information about the current status of the tokens being managed.

2.7.5. Command Line Interface (CLI)

Red Hat Certificate System provides a client command-line interface (CLI) to access various services on the server. The client CLI can be invoked as follows:

```
$ pki [CLI options] <command> [command parameters]
```

Note that the command line options must be placed before the command, and the command parameters after the command.

To use the command line interface for the first time, specify a new password and use the following command:

```
$ pki -c <password> client-init
```

This will create a new client NSS database in the **~/.dogtag/nssdb** directory. The password must be specified in all CLI operations that uses the client NSS database.

Some commands may require client certificate authentication. To import an existing client certificate and its key into the client NSS database, specify the PKCS #12 file and the password, and execute the following command:

```
$ pki -c <password> pkcs12-import --pkcs12-file <file> --pkcs12-password
<password>
```

To execute a command using the client certificate, specify the certificate nickname and the client NSS database password:

```
$ pki -n <nickname> -c ≤password> <command> [command parameters]
```

The command line interface supports a number of commands organized in a hierarchical structure. To list the top-level commands, execute the **pki** command without any additional commands or parameters:

```
$ pki
```

Some commands have subcommands. To list them, execute **pki** with the command name and no additional options. For example:

```
$ pki ca
```

```
$ pki tps
```

To view command usage information, use the **--help** option:

```
$ pki --help
```

```
$ pki ca-cert-find --help
```

To view manual pages, specify the command line **help** command:

```
$ pki help
```

```
$ pki help ca-cert-find
```

2.8. ABOUT CLONING

Planning for *high availability* reduces unplanned outages and other problems by making one or more subsystem clones available. When a host machine goes down, the cloned subsystems can handle requests and perform services, taking over from the master (original) subsystem seamlessly and keeping uninterrupted service.

Using cloned subsystems also allows systems to be taken offline for repair, troubleshooting, or other administrative tasks without interrupting the services of the overall PKI system.

NOTE

All of the subsystems except the TPS can be cloned.

Cloning is one method of providing scalability to the PKI by assigning the same task, such as handling certificate requests, to separate instances on different machines. The internal databases for the master and its clones are replicated between each other, so the information about certificate requests or archived keys on one subsystem is available on all the others.

Typically, master and cloned instances are installed on different machines, and those machines are placed behind a *load balancer*. The load balancer accepts HTTP and HTTPS requests made to the Certificate System subsystems and directs those requests appropriately between the master and cloned instances. In the event that one machine fails, the load balancer transparently redirects all requests to the machine that is still running until the other machine is brought back online.

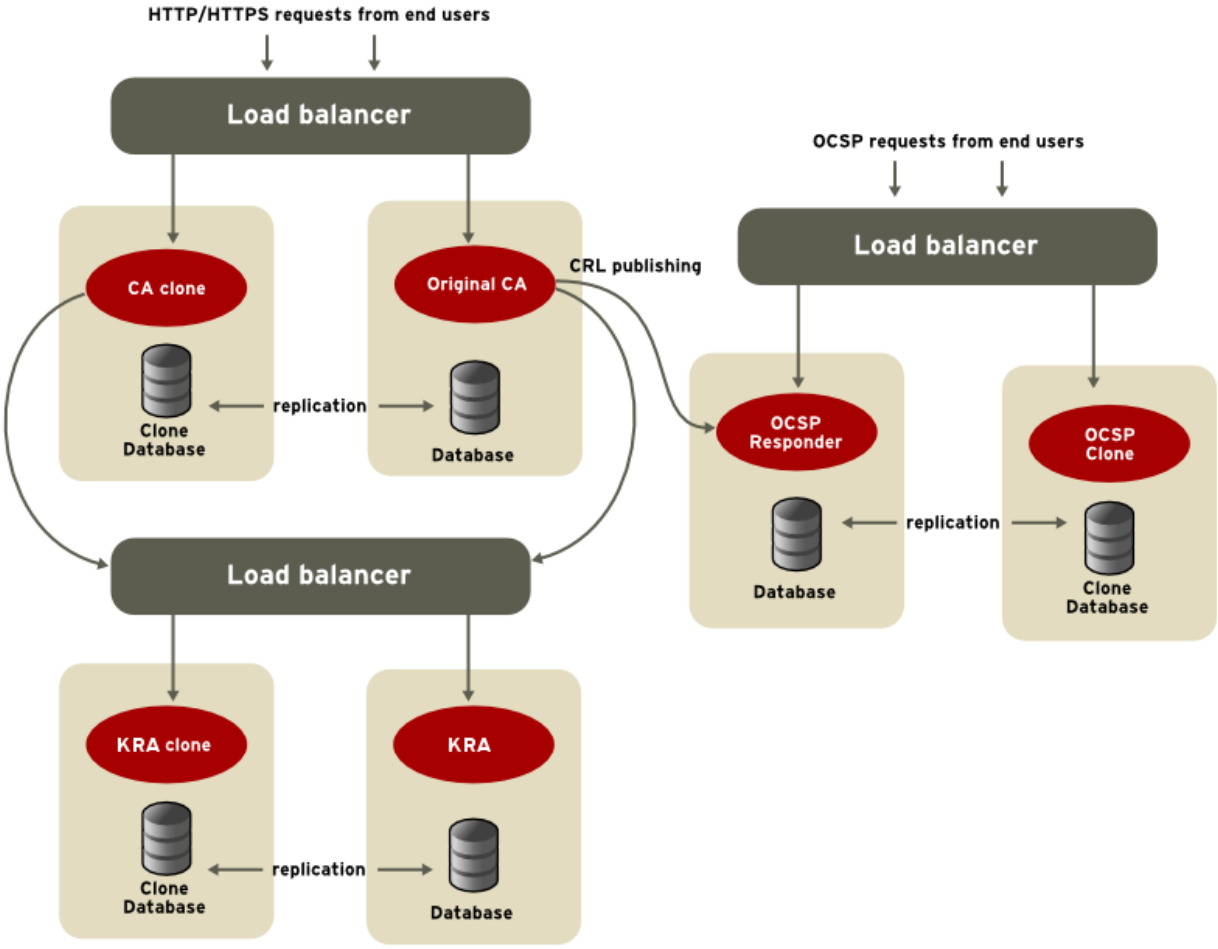


Figure 2.9. Cloning Example

The load balancer in front of a Certificate System subsystem is what provides the actual failover support in a high availability system. A load balancer can also provide the following advantages as part of a Certificate System subsystem:

- DNS round-robin, a feature for managing network congestion that distributes load across several different servers.
- Sticky SSL/TLS, which makes it possible for a user returning to the system to be routed the same host used previously.

Normally, when a system is cloned, the configuration servlet sets up the replication agreements between the internal LDAP databases. However, some users may want to establish and manage their own replication agreements. The pkispawn executable has been modified to allow this by adding a **[Tomcat]** section to the PKI instance override configuration file and adding the following two **name=value** pairs under that section:

```
[Tomcat]
pki_clone_setup_replication=False
pki_clone_reindex_data=False
```

IMPORTANT

Data must be replicated *before* **pkispawn** is invoked when specifying this kind of installation.

2.8.1. Cloning for CAs

Cloned instances have the exact same private keys as the master, so their certificates are identical. For CAs, that means that the CA signing certificates are identical for the original master CA and its cloned CAs. From the perspectives of clients, these look like a single CA.

Every CA, both cloned and master, can issue certificates and process revocation requests.

The main issue with managing replicated CAs is how to assign serial numbers to the certificates they issue. Different replicated CAs can have different levels of traffic, using serial numbers at different rates, and it is imperative that no two replicated CAs issue certificates with the same serial number. These serial number ranges are assigned and managed dynamically by using a shared, replicated entry that defines the ranges for each CA and the next available range to reassign when one CA range runs low.

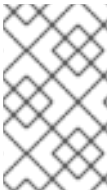
The serial number ranges with cloned CAs are fluid. All replicated CAs share a common configuration entry which defines the next available range. When one CA starts running low on available numbers, it checks this configuration entry and claims the next range. The entry is automatically updated, so that the next CA gets a new range.

The ranges are defined in ***begin*Number*** and ***end*Number*** attributes, with separate ranges defined for requests and certificate serial numbers. For example:

```
dbs.beginRequestNumber=1
dbs.beginSerialNumber=1
dbs.enableSerialManagement=true
dbs.endRequestNumber=9980000
dbs.endSerialNumber=ffe0000
dbs.replicaCloneTransferNumber=5
```

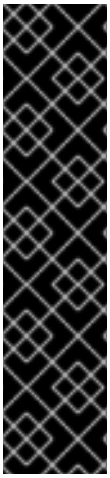
Only one replicated CA can generate, cache, and publish CRLs; this is the CRL CA. CRL requests submitted to other replicated CAs are immediately redirected to the CRL CA. While other replicated CAs can revoke, display, import, and download CRLs previously generated by the CRL CA, synchronization problems might occur if they generate new CRLs. For instructions on how to move CRL generation to a different replicated CA, see Section 10.7.1, “Converting CA Clones and Masters”.

Master CAs also manage the relationships and information sharing between the cloned CAs by monitoring replication changes to the internal databases.



NOTE

If a CA which is a security domain master is cloned, then that cloned CA is also a security domain master. In that case, both the original CA and its clone share the same security domain configuration.



IMPORTANT

As covered in Section 2.8.7, “Custom Configuration and Clones”, the data within the LDAP database is replicated among a master and clones, but the *configuration files* for the different instances are not replicated. This means that any changes which affect the **CS.cfg** file — such as adding a KRA connection or creating a custom profile — are not copied into a clone's configuration if the change occurs *after* the clone is created.

Any changes to the CA configuration should either be made to the master *before* any clones are created (so the custom changes are included in the cloning process) or any configuration changes must be copied over manually to the clone instances after they are created.

2.8.2. Cloning for KRAs

With KRAs, all keys archived in one KRA are replicated to the internal databases of the other KRAs. This allows a key recovery to be initiated on any clone KRA, regardless of which KRA the key was originally archived on.

After a key recovery is processed, the record of the recovery is stored in the internal database of all of the cloned KRAs.

With synchronous key recovery, although the recovery process can be initiated on any clone, it must be completed on the same single KRA on which it was initiated. This is because a recovery operation is recorded in the replicated database only after the appropriate number of approvals have been obtained from the KRA agents. Until then, the KRA on which the recovery is initiated is the only one which knows about the recovery operation.

IMPORTANT

The synchronous key recovery mechanism has been deprecated in Red Hat Certificate System 9. Red Hat recommends to use asynchronous key recovery instead.

2.8.3. Cloning for Other Subsystems

There is no real operational difference between replicated TKSs; the information created or maintained on one is replicated along the other servers.

For OCSPs, only one replicated OCSP receives CRL updates, and then the published CRLs are replicated to the clones.

2.8.4. Cloning and Key Stores

Cloning a subsystem creates two server processes performing the same functions: another new instance of the subsystem is created and configured to use the same keys and certificates to perform its operations. Depending on where the keys are stored for the master clone, the method for the clone to access the keys is very different.

If the keys and certificates are stored in the internal software token, then they must be exported from the master subsystem when it is first configured. When configuring the master instance, it is possible to backup the system keys and certificates to a PKCS #12

file by specifying the ***pki_backup_keys*** and ***pki_backup_password*** parameters in the **pkispawn** configuration file. See the **BACKUP PARAMETERS** section in the pki_default.cfg(5) man page for more details.

If the keys were not backed up during the initial configuration, you can extract them to a PKCS #12 file using the **PKCS12Export** utility, as described in Section 10.1, "Backing up Subsystem Keys from a Software Database".

Then copy the PKCS #12 file over to the clone subsystem, and define its location and password in the **pkispawn** configuration file using the ***pki_clone_pkcs12_password*** and ***pki_clone_pkcs12_path*** parameters For more information, see the **Installing a Clone** section in the pkispawn(8) man page. In particular, make sure that the PKCS#12 file is accessible by the **pkiuser** user and that it has the correct SELinux label.

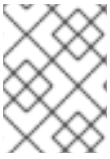
If the keys and certificates are stored on a hardware token, then the keys and certificates must be copied using hardware token specific utilities or referenced directly in the token:

- Duplicate all the required keys and certificates, except the SSL/TLS server key and certificate to the clone instance. Keep the nicknames for those certificates the same. Additionally, copy all the necessary trusted root certificates from the master instance to the clone instance, such as chains or cross-pair certificates.
- If the token is network-based, then the keys and certificates simply need to be available to the token; the keys and certificates do not need to be copied.
- When using a network-based hardware token, make sure the high-availability feature is enabled on the hardware token to avoid single point of failure.

2.8.5. LDAP and Port Considerations

As mentioned in Section 2.8, "About Cloning", part of the behavior of cloning is to replicate information between replicated subsystems, so that they work from an identical set of data and records. This means that the LDAP servers for the replicated subsystems need to be able to communicate.

If the Directory Server instances are on different hosts, then make sure that there is appropriate firewall access to allow the Directory Server instances to connect with each other.



NOTE

Cloned subsystems and their masters must use separate LDAP servers while they replicate data between common suffixes.

A subsystem can connect to its internal database using either SSL/TLS over an LDAPS port or over a standard connection over an LDAP port. When a subsystem is cloned, the clone instance uses the same connection method (SSL/TLS or standard) as its master to connect to the database. With cloning, there is an additional database connection though: the master Directory Server database to the clone Directory Server database. For that connection, there are *three* connection options:

- If the master uses SSL/TLS to connect to its database, then the clone uses SSL/TLS, and the master/clone Directory Server databases use SSL/TLS connections for replication.
- If the master uses a standard connection to its database, then the clone must use a

standard connection, and the Directory Server databases *can* use unencrypted connections for replication.

- If the master uses a standard connection to its database, then the clone must use a standard connection, *but* there is an option to use Start TLS for the master/clone Directory Server databases for replication. Start TLS opens a secure connection over a standard port.

NOTE

To use Start TLS, the Directory Server must still be configured to accept SSL/TLS connections. This means that prior to configuring the clone, a server certificate and a CA certificate must be installed on the Directory Server, and SSL/TLS must be enabled.

Whatever connection method (secure or standard) used by the master must be used by the clone and must be properly configured for the Directory Server databases prior to configuring the clone.

IMPORTANT

Even if the clone connects to the master over a secure connection, the standard LDAP port (389 by default) must still be open and enabled on the LDAP server while cloning is configured.

For secure environments, the standard LDAP port can be disabled on the master's Directory Server instance once the clone is configured.

2.8.6. Replica ID Numbers

Cloning is based on setting up a replication agreement between the Directory Server for the master instance and the Directory Server for the cloned instance.

Servers involved together with replication are in the same replication *topology*. Every time a subsystem instance is cloned, it is added to the overall topology. Directory Server discerns between different servers in the topology based on their *replica ID number*. This replica ID must be unique among all of the servers in the topology.

As with the serial number ranges used for requests and certificates (covered in Section 2.8.1, “Cloning for CAs”), every subsystem is assigned a range of allowed replica IDs. When the subsystem is cloned, it assigns one of the replica IDs from its range to the new clone instance.

```
dbs.beginReplicaNumber=1
dbs.endReplicaNumber=95
```

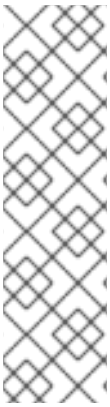
The replica ID range can be refreshed with new numbers if an instance begins to exhaust its current range.

2.8.7. Custom Configuration and Clones

After a clone is created, *configuration* changes are not replicated between clones or between a master and a clone. The instance configuration is in the **CS.cfg** file — outside the replicated database.

For example, there are two CAs, a master and a clone. A new KRA is installed which is associated, at its configuration, with the master CA. The CA-KRA connector information is stored in the master CA's **`CS.cfg`** file, but this connector information is not added to the clone CA configuration. If a certificate request that includes a key archival is submitted to the master CA, then the key archival is forwarded to the KRA using the CA-KRA connector information. If the request is submitted to the clone CA, no KRA is recognized, and the key archival request is disallowed.

Changes made to the configuration of a master server or to a clone server are not replicated to other cloned instances. Any critical settings must be added manually to clones.



NOTE

You can set all required, custom configuration for a master server before configuring any clones. For example, install all KRAs so that all the connector information is in the master CA configuration file, create any custom profiles, or configure all publishing points for a master OCSP responder. Note that if the LDAP profiles are stored in the Directory Server, they are replicated and kept in sync across servers.

Any custom settings in the master instance will be included in the cloned instances *at the time they are cloned* (but not after).

CHAPTER 3. SUPPORTED STANDARDS AND PROTOCOLS

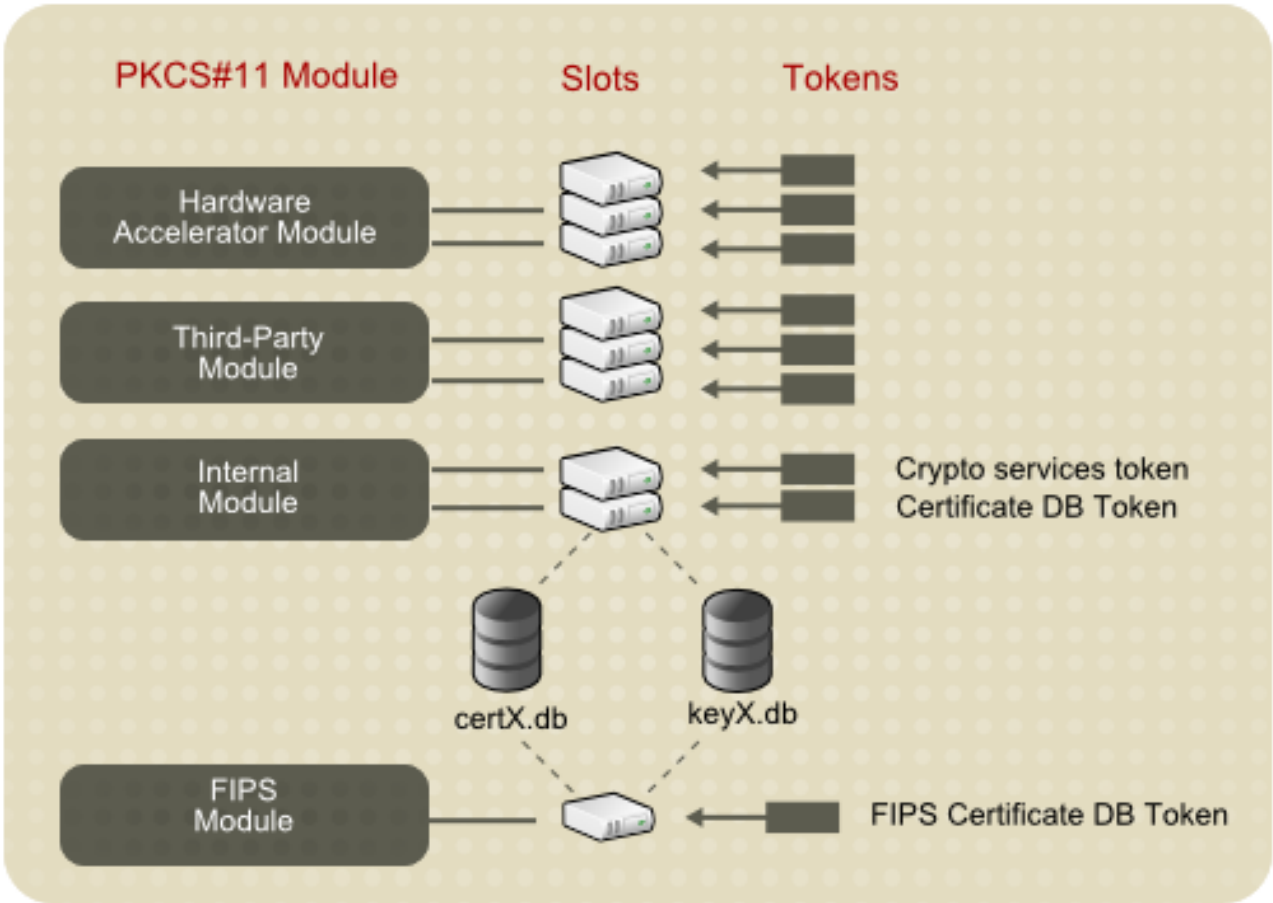
Red Hat Certificate System is based on many public and standard protocols and RFCs, to ensure the best possible performance and interoperability. The major standards and protocols used or supported by Certificate System 9 are outlined in this chapter, to help administrators plan their client services effectively.

3.1. PKCS #11

Public-Key Cryptography Standard (PKCS) #11 specifies an API used to communicate with devices that hold cryptographic information and perform cryptographic operations. Because it supports PKCS #11, Certificate System is compatible with a wide range of hardware and software devices.

At least one PKCS #11 module must be available to any Certificate System subsystem instance. A PKCS #11 module (also called a cryptographic module or cryptographic service provider) manages cryptographic services such as encryption and decryption. PKCS #11 modules are analogous to drivers for cryptographic devices that can be implemented in either hardware or software. Certificate System contains a built-in PKCS #11 module and can support third-party modules.

A PKCS #11 module always has one or more slots which can be implemented as physical hardware slots in a physical reader such as smart cards or as conceptual slots in software. Each slot for a PKCS #11 module can in turn contain a token, which is a hardware or software device that actually provides cryptographic services and optionally stores certificates and keys.



Two cryptographic modules are included in the Certificate System:

- The default internal PKCS #11 module, which comes with two tokens:
  - The internal crypto services token, which performs all cryptographic operations such as encryption, decryption, and hashing.
  - The internal key storage token ("Certificate DB token"), which handles all communication with the certificate and key database files that store certificates and keys.
- The FIPS 140 module. This module complies with the FIPS 140 government standard for cryptographic module implementations. The FIPS 140 module includes a single, built-in FIPS 140 certificate database token, which handles both cryptographic operations and communication with the certificate and key database files.

Any PKCS #11 module can be used with the Certificate System. To use an external hardware token with a subsystem, load its PKCS #11 module before the subsystem is configured, and the new token is available to the subsystem.

Available PKCS #11 modules are tracked in the **`secmod.db`** database for the subsystem. The **`modutil`** utility is used to modify this file when there are changes to the system, such as installing a hardware accelerator to use for signing operations. For more information on **`modutil`**, see http://www.mozilla.org/projects/security/pki/nss/tools/.

PKCS #11 hardware devices also provide key backup and recovery features for the information stored on hardware tokens. Refer to the PKCS #11 vendor documentation for information on retrieving keys from the tokens.

3.2. SSL/TLS, ECC, AND RSA

Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols are universally accepted standards for authenticated and encrypted communication between clients and servers. Both client and server authentication occur over SSL/TLS.

NOTE

SSL3.0 is no longer supported due to security reasons. Using TLS 1.2 ciphers is strongly recommended if the environment allows it.

SSL/TLS uses a combination of public-key and symmetric-key encryption. Symmetric-key encryption is much faster than public-key encryption, but public-key encryption provides better authentication techniques. An SSL/TLS session always begins with an exchange of messages called *handshake*, initial communication between the server and client. The handshake allows the server to authenticate itself to the client using public-key techniques, optionally allows the client to authenticate to the server, then allows the client and the server to cooperate in the creation of symmetric keys used for rapid encryption, decryption, and integrity verification during the session that follows.

Both SSL and TLS support a variety of different cryptographic algorithms, or *ciphers*, for operations such as authenticating the server and client, transmitting certificates, and establishing session keys. Clients and servers may support different cipher suites, or sets of ciphers. Among other functions, the handshake determines how the server and client negotiate which cipher suite is used to authenticate each other, to transmit certificates, and to establish session keys.

Key-exchange algorithms like RSA and Elliptic Curve Diffie-Hellman (ECDH) govern the way the server and client determine the symmetric keys to use during an SSL/TLS session. The most common SSL cipher suites use RSA key exchange, while TLS supports ECC (Elliptic Curve Cryptography) cipher suites, as well as RSA. The Certificate System supports both RSA and ECC public-key cryptographic systems natively.

In more recent practise, key-exchange algorithms are being superseded by *key-agreement protocols* where each of the two or more parties can influence the outcome when establishing a common key for secure communication. Key agreement is preferrable to key exchange because it allows for Perfect Forward Secrecy (PFS) to be implemented. When PFS is used, random public keys (also called temporary cipher parameters or *ephemeral keys*) are generated for each session by a non-deterministic algorithm for the purposes of key agreement. As a result, there is no single secret value which could lead to the compromise of multiple messages, protecting past and future communication alike.

NOTE

Longer RSA keys are required to provide security as computing capabilities increase. The recommended RSA key-length is 2048 bits. Though many servers continue to use 1024-bit keys, servers should migrate to at least 2048 bits. For 64-bit machines, consider using stronger keys. All CAs should use at least 2048-bit keys, and stronger keys (such as 3072 or 4096 bits) if possible.

3.2.1. Supported Cipher Suites

Cipher and hashing algorithms are in a constant flux with regard to various vulnerabilities and security strength. As a general rule, Red Hat Certificate System 9 follows the NIST guideline and supports TLS 1.1 and TLS 1.2 cipher suites pertaining to the server keys.

3.2.2. Using ECC

Elliptic Curve Cryptography (ECC) is a cryptographic system that uses elliptic curves to create keys for encrypting data. ECC creates cryptographically-stronger keys with shorter key lengths than RSA, which makes it faster and more efficient to implement.

ECC has several advantages over RSA, since it is faster and requires shorter key lengths for stronger keys. On the other hand, ECC is not yet as widely supported as RSA. For information on cipher key strength comparison, see "Table 2.1 Comparable strengths" in NIST Recommendation for Key Management – Part 1: General (Revision 3).

Certificate System supports ECC with the ECC with CA, KRA, and OCSP subsystems, so ECC certificate requests can be submitted to CAs through any of the enrollment profiles and ECC keys can be archived and restored in the KRA. Certificate System supports ECC natively; for more information, see Chapter 9, *Installing an Instance with ECC System Certificates*.

NOTE

A CA with an ECC CA signing certificate can issue both ECC and RSA certificates. A CA with an RSA CA signing certificate can only issue RSA certificates.

Only the CA signing certificate is required; if for support purposes it is better to use RSA client certificates with the CA, simply delete the ECC subsystem certificates (except for the signing certificate) and replace them with RSA certificates.[1].

3.3. IPV4 AND IPV6 ADDRESSES

Certificate System supports both IPv4 addresses and IPv6 addresses. In a very wide variety of circumstances, Certificate System subsystems or operations reference a host name or IP address; supporting both IPv4- and IPv6-style addresses ensures forward compatibility with network protocols. The operations that support IPv6 connections include the following:

- Communications between subsystems, including between the TPS, TKS, and CA and for joining security domains
- Token operations between the TPS and the Enterprise Security Client
- Subsystem logging
- Access control instructions
- Operations performed with Certificate System tools, including the **pki** utility, the Subject Alt Name Extension tool, HttpClient, and the Bulk Issuance Tool
- Client communications, including both the **pkiconsole** utility and IPv6-enabled browsers for web services
- Certificate request names and certificate subject names, including user, server, and router certificates
- Publishing
- Connecting to LDAP databases for internal databases and authentication directories

Any time a host name or URL is referenced, an IP address can be used:

- An IPv4 address must be in the format **n.n.n.n** or **n.n.n.n,m.m.m.m**. For example, **128.21.39.40** or **128.21.39.40,255.255.255.00**.
- An IPv6 address uses a 128-bit namespace, with the IPv6 address separated by colons and the netmask separated by periods. For example, *0:0:0:0:0:0:13.1.68.3*, *FF01::43*, or *0:0:0:0:0:0:13.1.68.3,FFFF:FFFF:FFFF:FFFF:FFFF:FFFF:255.255.255.0*.

If DNS is properly configured, then an IPv4 or IPv6 address can be used to connect to the web services pages and to the subsystem Java consoles. The most common method is to use fully-qualified domain names:

```
https://ipv6host.example.com:8443/ca/services
pkiconsole https://ipv6host.example.com:8443/ca
```

To use IPv6 numeric addresses, replace the fully-qualified domain name in the URL with the IPv6 address, enclosed in brackets (**[]**). For example:

```
https://[00:00:00:00:123:456:789:00:]:8443/ca/services
pkiconsole https://[00:00:00:00:123:456:789:00:]:8443/ca
```

3.4. SUPPORTED PKIX FORMATS AND PROTOCOLS

The Certificate System supports many of the protocols and formats defined in Public-Key Infrastructure (X.509) by the IETF. In addition to the PKIX standards listed here, other PKIX-listed standards are available at the IETF Datatracker website.

Table 3.1. PKIX Standards Supported in Certificate System 9

Format or Protocol	RFC or Draft	Description
X.509 version 1 and version 3		Digital certificate formats recommended by the International Telecommunications Union (ITU).
Certificate Request Message Format (CRMF)	RFC 4211	A message format to send a certificate request to a CA.
Certificate Management Message Formats (CMMF)		Message formats to send certificate requests and revocation requests from end entities to a CA and to return information to end entities. CMMF has been subsumed by another standard, CMC.
Certificate Management Messages over CS (CMC)	RFC 5274	A general interface to public-key certification products based on CS and PKCS #10, including a certificate enrollment protocol for RSA-signed certificates with Diffie-Hellman public-keys. CMC incorporates CRMF and CMMF.
Cryptographic Message Syntax (CMS)	RFC 2630	A superset of PKCS #7 syntax used for digital signatures and encryption.
PKIX Certificate and CRL Profile	RFC 5280	A standard developed by the IETF for a public-key infrastructure for the Internet. It specifies profiles for certificates and CRLs.
Online Certificate Status Protocol (OCSP)	RFC 6960	A protocol useful in determining the current status of a digital certificate without requiring CRLs.

3.5. SUPPORTED SECURITY AND DIRECTORY PROTOCOLS

The Certificate System supports several common Internet and network protocols.

Table 3.2. Supported Security and Directory Protocols

Protocol	Description
FIPS PUBS 140	Federal Information Standards Publications (FIPS PUBS) 140 is a US government standard for implementing cryptographic modules such as hardware or software that encrypts and decrypts data, creates and verifies digital signatures, and provides other cryptographic functions. More information is available at http://csrc.nist.gov/publications/PubsFIPS.html.
Hypertext Transport Protocol (HTTP) and Hypertext Transport Protocol Secure (HTTPS)	Protocols used to communicate with web servers.
KEYGEN tag	An HTML tag that generates a key pair for use with a certificate.
Lightweight Directory Access Protocol (LDAP) v2, v3	A directory service protocol designed to run over TCP/IP and across multiple platforms. LDAP is a simplified version of Directory Access Protocol (DAP), used to access X.500 directories. LDAP is under IETF change control and has evolved to meet Internet requirements.
Public-Key Cryptography Standard (PKCS) #7	An encrypted data and message format developed by RSA Data Security to represent digital signatures, certificate chains, and encrypted data. This format is used to deliver certificates to end entities.
Public-Key Cryptography Standard (PKCS) #10	A message format developed by RSA Data Security for certificate requests. This format is supported by many server products.
Public-Key Cryptography Standard (PKCS) #11	Specifies an API used to communicate with devices such as hardware tokens that hold cryptographic information and perform cryptographic operations.
Transport Layer Security (TLS)	A set of rules governing server authentication, client authentication, and encrypted communication between servers and clients.

Protocol	Description
Security-Enhanced Linux	Security-enhanced Linux (SELinux) is a set of security protocols enforcing mandatory access control on Linux system kernels. SELinux was developed by the United States National Security Agency to keep applications from accessing confidential or protected files through lenient or flawed access controls.
Simple Certificate Enrollment Protocol (SCEP)	A protocol designed by Cisco to specify a way for a router to communicate with a CA for router certificate enrollment. Certificate System supports SCEP's CA mode of operation, where the request is encrypted with the CA signing certificate.
UTF-8	The certificate enrollment pages support all UTF-8 characters for specific fields (common name, organizational unit, requester name, and additional notes). The UTF-8 strings are searchable and correctly display in the CA, OCSP, and KRA end user and agents services pages. However, the UTF-8 support does not extend to internationalized domain names, such as those used in email addresses.
HTTPS	This protocol consists of communication over HTTP (Hypertext Transfer Protocol) within a connection encrypted by Transport Layer Security (TLS). The main purpose of HTTPS is authentication of the visited website and protection of privacy and integrity of the exchanged data.
IPv4 and IPv6	Certificate System supports both IPv4 and IPv6 address namespaces for communications and operations with all subsystems and tools, as well as for clients, subsystem creation, and token and certificate enrollment.

[1] For more information on ECC, see RFC 4492 at http://ietf.org/rfc/rfc4492.txt

CHAPTER 4. SUPPORTED PLATFORMS, HARDWARE, AND PROGRAMS

4.1. GENERAL REQUIREMENTS

For details, see the corresponding section in the Red Hat Certificate System 9 Release Notes.

4.2. SUPPORTED CHARACTER SETS

Red Hat Certificate System fully supports UTF-8 characters in the CA end users forms for specific fields. This means that end users can submit certificate requests with UTF-8 characters in those fields and can search for and retrieve certificates and CRLs in the CA and retrieve keys in the KRA when using those field values as the search parameters.

Four fields fully-support UTF-8 characters:

- Common name (used in the subject name of the certificate)
- Organizational unit (used in the subject name of the certificate)
- Requester name
- Additional notes (comments appended by the agent to the certificate)

NOTE

This support does not include supporting internationalized domain names, like in email addresses.

CHAPTER 5. PLANNING THE CERTIFICATE SYSTEM

Each Red Hat Certificate System subsystem can be installed on the same server machine, installed on separate servers, or have multiple instances installed across an organization. Before installing any subsystem, it is important to plan the deployment out: what kind of PKI services do you need? What are the network requirements? What people need to access the Certificate System, what are their roles, and what are their physical locations? What kinds of certificates do you want to issue and what constraints or rules need to be set for them?

This chapter covers some basic questions for planning a Certificate System deployment. Many of these decisions are interrelated; one choice impacts another, like deciding whether to use smart cards determines whether to install the TPS and TKS subsystems.

5.1. DECIDING ON THE REQUIRED SUBSYSTEMS

The Certificate System subsystems cover different aspects of managing certificates. Planning which subsystems to install is one way of defining what PKI operations the deployment needs to perform.

Certificates, like software or equipment, have a lifecycle with defined stages. At its most basic, there are three steps:

- It is requested and issued.
- It is valid.
- It expires.

However, this simplified scenario does not cover a lot of common issues with certificates:

- What if an employee leaves the company before the certificate expires?
- When a CA signing certificate expires, all of the certificates issued and signed using that certificate also expire. So will the CA signing certificate be renewed, allowing its issued certificates to remain valid, or will it be reissued?
- What if an employee loses a smart card or leaves it at home. Will a replacement certificate be issued using the original certificates keys? Will the other certificates be suspended or revoked? Are temporary certificates allowed?
- When a certificate expires, will a new certificate be issued or will the original certificate be renewed?

This introduces three other considerations for managing certificates: revocation, renewal, and replacements.

Other considerations are the loads on the certificate authority. Are there a lot of issuance or renewal requests? Is there a lot of traffic from clients trying to validate whether certificates are valid? How are people requesting certificates supposed to authenticate their identity, and does that process slow down the issuance process?

5.1.1. Using a Single Certificate Manager

The core of the Certificate System PKI is the Certificate Manager, a certificate authority. The CA receives certificate requests and issues all certificates.

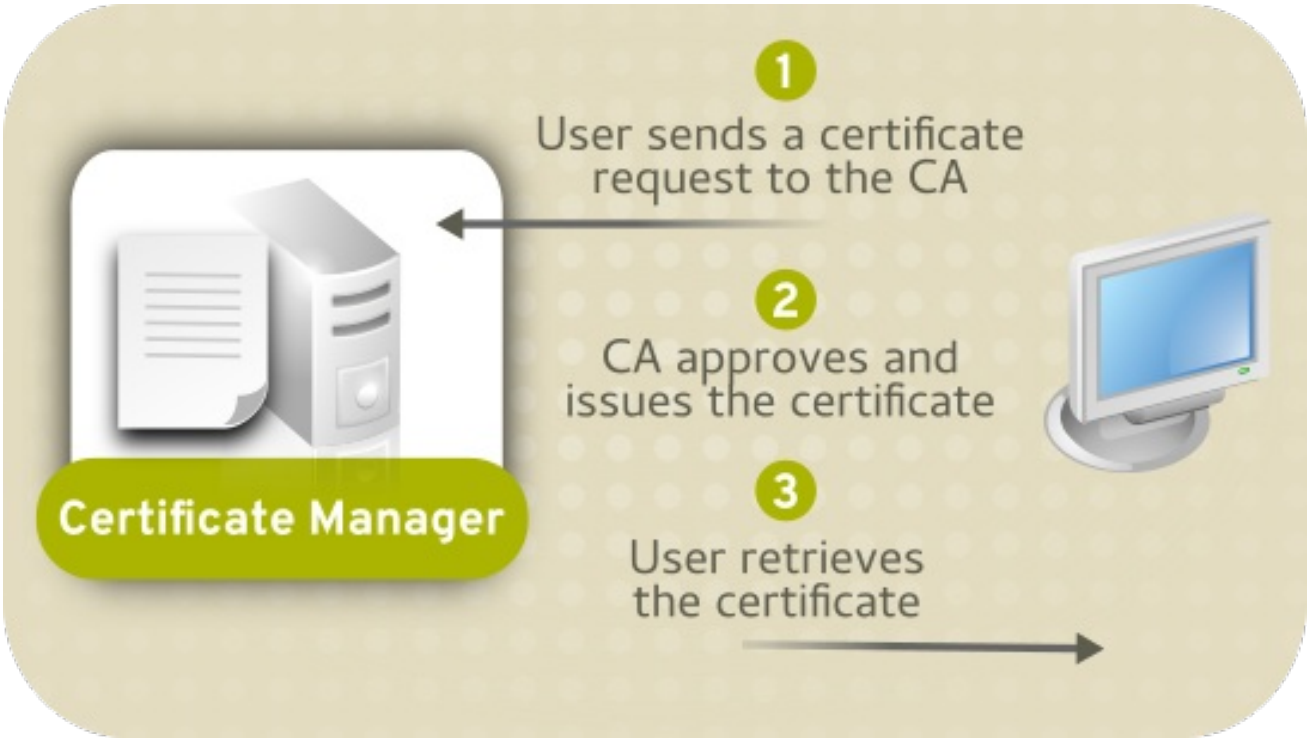
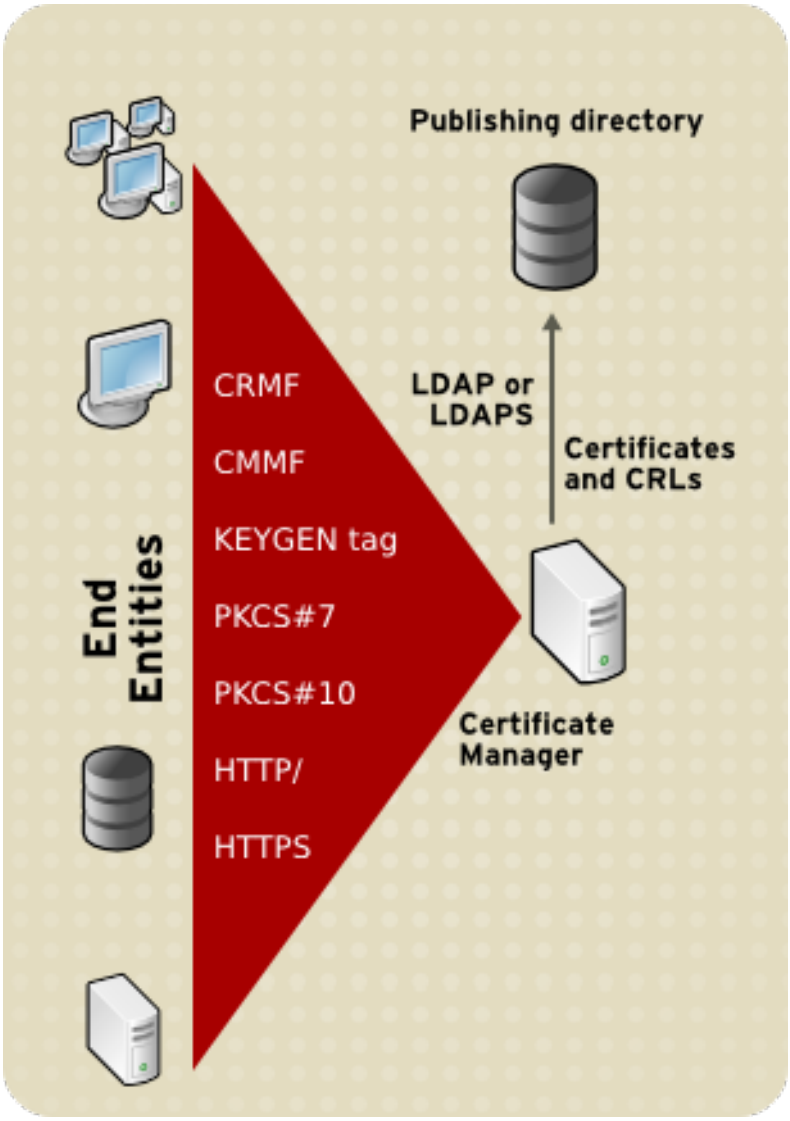


Figure 5.1. CA Only Certificate System

All of the basic processing for requests and issuing certificates can be handled by the Certificate Manager, and it is the only required subsystem. There can be a single Certificate Manager or many Certificate Managers, arranged in many different relationships, depending on the demands of the organization.

Along with issuing certificates, a Certificate Manager can also revoke certificates. One question for administrators is how to handle certificates if they are lost, compromised, or if the person or equipment for which they are issued leaves the company. Revoking a certificate invalidates it before its expiration date, and a list of revoked certificates is compiled and published by the issuing CA so that clients can verify the certificate status.



5.1.2. Planning for Lost Keys: Key Archival and Recovery

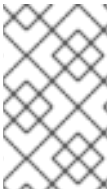
One operation the CA cannot perform, though, is key archival and recovery. A very real scenario is that a user is going to lose his private key — for instance, the keys could be deleted from a browser database or a smart card could be left at home. Many common business operations use encrypted data, like encrypted email, and losing the keys which unlock that data means the data itself is lost. Depending on the policies in the company, there probably has to be a way to recover the keys in order to regenerate or reimport a replacement certificate, and both operations require the private key.

That requires a KRA, the subsystem which specially archives and retrieves keys.

Figure 5.2. CA and KRA

The Key Recovery Authority stores encryption keys (key archival) and can retrieve those keys so that the CA can reissue a certificate (key recovery). A KRA can store keys for any certificate generated by Certificate System, whether it is done for a regular certificate or when enrolling a smart card.

The key archival and recovery process is explained in more detail in Section 2.4.5, “Archiving, Recovering, and Rotating Keys”.



NOTE

The KRA is intended for archival and recovery of private encryption keys only. Therefore, end users must use a browser that supports dual-key generation to store their public-private key pairs.

5.1.3. Balancing Certificate Request Processing

Another aspect of how the subsystems work together is load balancing. If a site has high traffic, then it is possible to simply install a lot of CAs, as clones of each other or in a flat hierarchy (where each CA is independent) or in a tree hierarchy (where some CAs are subordinate to other CAs); this is covered more in Section 5.2, “Defining the Certificate Authority Hierarchy”.

5.1.4. Balancing Client OCSP Requests

If a certificate is within its validity period but needs be invalidated, it can be revoked. A Certificate Manager can publish lists of revoked certificates, so that when a client needs to verify that a certificate is still valid, it can check the list. These requests are *online certificate status protocol requests*, meaning that they have a specific request and response format. The Certificate Manager has a built-in OCSP responder so that it can verify OCSP requests by itself.

However, as with certificate request traffic, a site may have a significant number of client

requests to verify certificate status. Example Corp. has a large web store, and each customer's browser tries to verify the validity of their SSL/TLS certificates. Again, the CA can handle issuing the number of certificates, but the high request traffic affects its performance. In this case, Example Corp. uses the external OCSP Manager subsystem to verify certificate statuses, and the Certificate Manager only has to publish updated CRLs every so often.

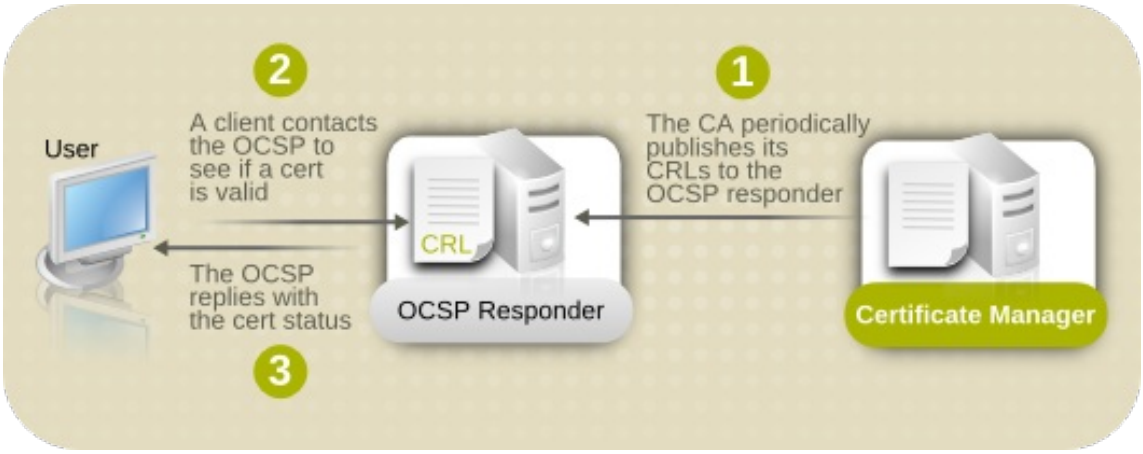
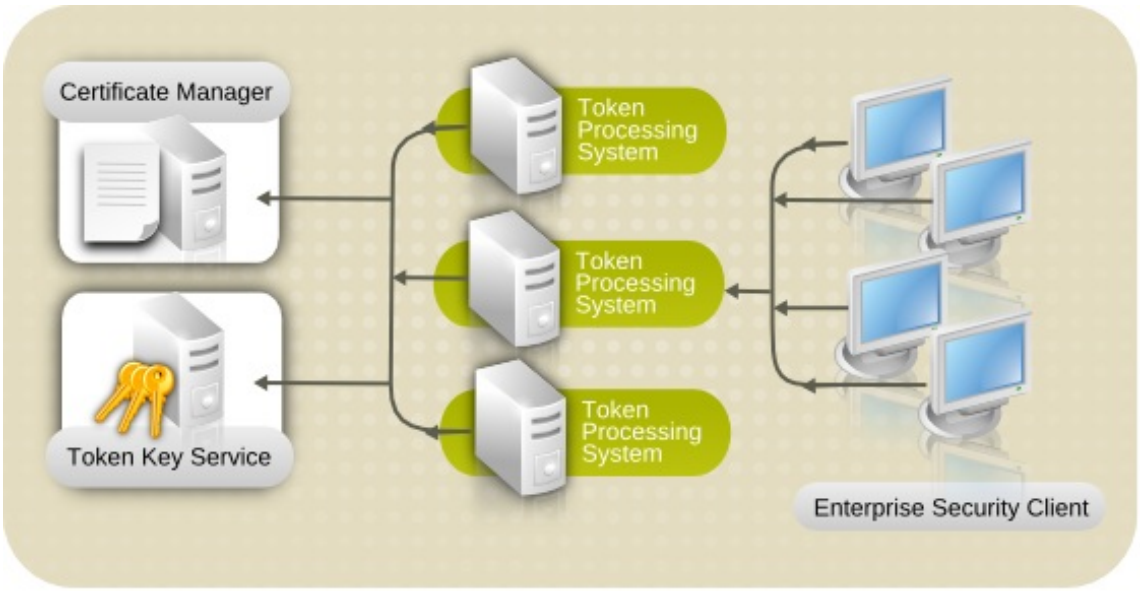


Figure 5.3. CA and OCSP

5.1.5. Using Smart Cards

Smart cards usually require in-person enrollment and approval processes, since they use a physical medium to store key and certificate data. That means that multiple agents need to have access to the TPS and, possibly, multiple TPS subsystems in different offices or geographic locations.

The token management system is very scalable. Multiple TPSs can be configured to work with a single CA, TKS, or KRA instance, while multiple Enterprise Security Clients can communicate with a single TPS. As additional clients are installed, they can point back to the TPS instance without having to reconfigure the TPS; likewise, as TPSs are added, they can point to the same CA, TKS, and KRA instances without having to reconfigure those subsystems.



After installation, the TPS configuration can be edited to use additional CA, KRA, and TKS instances for failover support, so if the primary subsystem is unavailable, the TPS can switch to the next available system without interrupting its token services.

5.2. DEFINING THE CERTIFICATE AUTHORITY HIERARCHY

The CA is the center of the PKI, so the relationship of CA systems, both to each other (CA hierarchy) and to other subsystems (security domain) is vital to planning a Certificate System PKI.

When there are multiple CAs in a PKI, the CAs are structured in a hierarchy or chain. The CA above another CA in a chain is called an *root CA*; a CA below another CA in the chain is called a *subordinate CA*. A CA can also be subordinate to a root outside of the Certificate System deployment; for example, a CA which functions as a root CA within the Certificate System deployment can be subordinate to a third-party CA.

A Certificate Manager (or CA) is subordinate to another CA because its CA signing certificate, the certificate that allows it to issue certificates, is issued by another CA. The CA that issued the subordinate CA signing certificate controls the CA through the contents of the CA signing certificate. The CA can constrain the subordinate CA through the kinds of certificates that it can issue, the extensions that it is allowed to include in certificates, the number of levels of subordinate CAs the subordinate CA can create, and the validity period of certificates it can issue, as well as the validity period of the subordinate CAs signing certificate.

NOTE

Although a subordinate CA can create certificates that violate these constraints, a client authenticating a certificate that violates those constraints will not accept that certificate.

A self-signed root CA signs its own CA signing certificate and sets its own constraints as well as setting constraints on the subordinate CA signing certificates it issues.

A Certificate Manager can be configured as either a root CA or a subordinate CA. It is easiest to make the first CA installed a self-signed root, so that it is not necessary to apply to a third party and wait for the certificate to be issued. Before deploying the full PKI, however, consider whether to have a root CA, how many to have, and where both root and subordinate CAs will be located.

5.2.1. Subordination to a Public CA

Chaining the Certificate System CA to a third-party public CA introduces the restrictions that public CAs place on the kinds of certificates the subordinate CA can issue and the nature of the certificate chain. For example, a CA that chains to a third-party CA might be restricted to issuing only Secure Multipurpose Internet Mail Extensions (S/MIME) and SSL/TLS client authentication certificates, but not SSL/TLS server certificates. There are other possible restrictions with using a public CA. This may not be acceptable for some PKI deployments.

One benefit of chaining to a public CA is that the third party is responsible for submitting the root CA certificate to a web browser or other client software. This can be a major advantage for an extranet with certificates that are accessed by different companies with browsers that cannot be controlled by the administrator. Creating a root CA in the CA

hierarchy means that the local organization must get the root certificate into all the browsers which will use the certificates issued by the Certificate System. There are tools to do this within an intranet, but it can be difficult to accomplish with an extranet.

5.2.2. Subordination to a Certificate System CA

The Certificate System CA can function as a *root CA*, meaning that the server signs its own CA signing certificate as well as other CA signing certificates, creating an organization-specific CA hierarchy. The server can alternatively be configured as a *subordinate CA*, meaning the server's CA signing key is signed by another CA in an existing CA hierarchy.

Setting up a Certificate System CA as the root CA means that the Certificate System administrator has control over all subordinate CAs by setting policies that control the contents of the CA signing certificates issued. A subordinate CA issues certificates by evaluating its own authentication and certificate profile configuration, without regard for the root CA's configuration.

5.2.3. Linked CA

The Certificate System Certificate Manager can function as a *linked CA*, chaining up to many third-party or public CAs for validation; this provides cross-company trust, so applications can verify certificate chains outside the company certificate hierarchy. A Certificate Manager is chained to a third-party CA by requesting the Certificate Manager's *CA signing certificate* from the third-party CA.

Related to this, the Certificate Manager also can issue *cross-pair* or *cross-signed certificates*. Cross-pair certificates create a trusted relationship between two separate CAs by issuing and storing cross-signed certificates between these two CAs. By using cross-signed certificate pairs, certificates issued outside the organization's PKI can be trusted within the system.

These are also called *bridge certificates*, related to the Federal Bridge Certification Authority (FBCA) definition.

5.2.4. CA Cloning

Instead of creating a hierarchy of root and subordinate CAs, it is possible to create multiple clones of a Certificate Manager and configure each clone to issue certificates within a range of serial numbers.

A *cloned* Certificate Manager uses the same CA signing key and certificate as another Certificate Manager, the *master* Certificate Manager.

NOTE

If there is a chance that a subsystem will be cloned, then it is easiest to export its key pairs during the configuration process and save them to a secure location. The key pairs for the original Certificate Manager have to be available when the clone instance is configured, so that the clone can generate its certificates from the original Certificate Manager's keys.

It is also possible to export the keys from the security databases at a later time, using the **`pk12util`** or the **`PKCS12Export`** commands.

Because clone CAs and original CAs use the same CA signing key and certificate to sign the

certificates they issue, the *issuer name* in all the certificates is the same. Clone CAs and the original Certificate Managers issue certificates as if they are a single CA. These servers can be placed on different hosts for high availability failover support.

The advantage of cloning is that it distributes the Certificate Manager's load across several processes or even several physical machines. For a CA with a high enrollment demand, the distribution gained from cloning allows more certificates to be signed and issued in a given time interval.

A cloned Certificate Manager has the same features, such as agent and end-entity gateway functions, of a regular Certificate Manager.

The serial numbers for certificates issued by clones are distributed dynamically. The databases for each clone and master are replicated, so all of the certificate requests and issued certificates, both, are also replicated. This ensures that there are no serial number conflicts while serial number ranges do not have to be manually assigned to the cloned Certificate Managers.

5.3. PLANNING SECURITY DOMAINS

A *security domain* is a registry of PKI services. PKI services, such as CAs, register information about themselves in these domains so users of PKI services can find other services by inspecting the registry. The security domain service in Certificate System manages both the registration of PKI services for Certificate System subsystems and a set of shared trust policies.

The registry provides a complete view of all PKI services provided by the subsystems within that domain. Each Certificate System subsystem must be either a host or a member of a security domain.

A CA subsystem is the only subsystem which can host a security domain. The security domain shares the CA internal database for privileged user and group information to determine which users can update the security domain, register new PKI services, and issue certificates.

A security domain is created during CA configuration, which automatically creates an entry in the security domain CA's LDAP directory. Each entry contains all the important information about the domain. Every subsystem within the domain, including the CA registering the security domain, is recorded under the security domain container entry.

The URL to the CA uniquely identifies the security domain. The security domain is also given a friendly name, such as **`Example Corp Intranet PKI`**. All other subsystems — KRA, TPS, TKS, OCSP, and other CAs — must become members of the security domain by supplying the security domain URL when configuring the subsystem.

Each subsystem within the security domain shares the same trust policies and trusted roots which can be retrieved from different servers and browsers. The information available in the security domain is used during configuration of a new subsystem, which makes the configuration process streamlined and automated. For example, when a TPS needs to connect to a CA, it can consult the security domain to get a list of available CAs.

Each CA has its own LDAP entry. The security domain is an organizational group underneath that CA entry:

```
ou=Security Domain,dc=server.example.com-pki-ca
```

Then there is a list of each subsystem type beneath the security domain organizational group, with a special object class (**pkiSecurityGroup**) to identify the group type:

```
cn=KRAList,ou=Security Domain,o=pki-tomcat-CA
objectClass: top
objectClass: pkiSecurityGroup
cn: KRAList
```

Each subsystem instance is then stored as a member of that group, with a special **pkiSubsystem** object class to identify the entry type:

```
dn: cn=kra.example.com:8443,cn=KRAList,ou=Security Domain,o=pki-tomcat-CA
objectClass: top
objectClass: pkiSubsystem
cn: kra.example.com:8443
host: server.example.com
UnSecurePort: 8080
SecurePort: 8443
SecureAdminPort: 8443
SecureAgentPort: 8443
SecureEEClientAuthPort: 8443
DomainManager: false
Clone: false
SubsystemName: KRA kra.example.com 8443
```

If a subsystem needs to contact another subsystem to perform an operation, it contacts the CA which hosts the security domain (by invoking a servlet which connects over the administrative port of the CA). The security domain CA then retrieves the information about the subsystem from its LDAP database, and returns that information to the requesting subsystem.

The subsystem authenticates to the security domain using a subsystem certificate.

Consider the following when planning the security domain:

- The CA hosting the security domain can be signed by an external authority.
- Multiple security domains can be set up within an organization. However, each subsystem can belong to only one security domain.
- Subsystems within a domain can be cloned. Cloning subsystem instances distributes the system load and provides failover points.
- The security domain streamlines configuration between the CA and KRA; the KRA can push its KRA connector information and transport certificates automatically to the CA instead of administrators having to manually copy the certificates over to the CA.
- The Certificate System security domain allows an offline CA to be set up. In this scenario, the offline root has its own security domain. All online subordinate CAs belong to a different security domain.
- The security domain streamlines configuration between the CA and OCSP. The OCSP can push its information to the CA for the CA to set up OCSP publishing and also retrieve the CA certificate chain from the CA and store it in the internal database.

5.4. DETERMINING THE REQUIREMENTS FOR SUBSYSTEM CERTIFICATES

The CA configuration determines many of the characteristics of the certificates which it issues, regardless of the actual type of certificate being issued. Constraints on the CA's own validity period, distinguished name, and allowed encryption algorithms impact the same characteristics in their issued certificates. Additionally, the Certificate Managers have predefined profiles that set rules for different kinds of certificates that they issue, and additional profiles can be added or modified. These profile configurations also impact issued certificates.

5.4.1. Determining Which Certificates to Install

When a Certificate System subsystem is first installed and configured, the certificates necessary to access and administer it are automatically created. These include an agent's certificate, server certificate, and subsystem-specific certificates. These initial certificates are shown in Table 5.1, “Initial Subsystem Certificates”.

Table 5.1. Initial Subsystem Certificates

Subsystem	Certificates
Certificate Manager	• CA signing certificate • OCSP signing certificate • SSL/TLS server certificate • Subsystem certificate • User's (agent/administrator) certificate • Audit log signing certificate
OCSP	• OCSP signing certificate • SSL/TLS server certificate • Subsystem certificate • User's (agent/administrator) certificate • Audit log signing certificate

Subsystem	Certificates
KRA	• Transport certificate • Storage certificate • SSL/TLS server certificate • Subsystem certificate • User's (agent/administrator) certificate • Audit log signing certificate
TKS	• SSL/TLS server certificate • User's (agent/administrator) certificate • Audit log signing certificate
TPS	• SSL/TLS server certificate • User's (agent/administrator) certificate • Audit log signing certificate

There are some cautionary considerations about replacing existing subsystem certificates.

- Generating new key pairs when creating a new self-signed CA certificate for a root CA will invalidate all certificates issued under the previous CA certificate.

  This means none of the certificates issued or signed by the CA using its old key will work; subordinate Certificate Managers, KRAs, OCSPs, TKSs, and TPSs will no longer function, and agents can no longer to access agent interfaces.

  This same situation occurs if a subordinate CA's CA certificate is replaced by one with a new key pair; all certificates issued by that CA are invalidated and will no longer work.

  Instead of creating new certificates from new key pairs, consider renewing the existing CA signing certificate.

- If the CA is configured to publish to the OCSP and it has a new CA signing certificate or a new CRL signing certificate, the CA must be identified again to the OCSP.

- If a new transport certificate is created for the KRA, the KRA information must be updated in the CA's configuration file, **`CS.cfg`**. The existing transport certificate must be replaced with the new one in the **`ca.connector.KRA.transportCert`** parameter.

- If a CA is cloned, then when creating a new SSL/TLS server certificate for the master Certificate Manager, the clone CAs' certificate databases all need updated with the new SSL/TLS server certificate.

- If the Certificate Manager is configured to publish certificates and CRLs to an LDAP directory and uses the SSL/TLS server certificate for SSL/TLS client authentication, then the new SSL/TLS server certificate must be requested with the appropriate extensions. After installing the certificate, the publishing directory must be configured to use the new server certificate.

- Any number of SSL/TLS server certificates can be issued for a subsystem instance, but it really only needs one SSL/TLS certificate. This certificate can be renewed or replaced as many times as necessary.

5.4.2. Planning the CA Distinguished Name

The core elements of a CA are a signing unit and the Certificate Manager identity. The signing unit digitally signs certificates requested by end entities. A Certificate Manager must have its own distinguished name (DN), which is listed in every certificate it issues.

Like any other certificate, a CA certificate binds a DN to a public key. A DN is a series of name-value pairs that in combination uniquely identify an entity. For example, the following DN identifies a Certificate Manager for the Engineering department of a corporation named Example Corporation:

```
cn=demoCA, o=Example Corporation, ou=Engineering, c=US
```

Many combinations of name-value pairs are possible for the Certificate Manager's DN. The DN must be unique and readily identifiable, since any end entity can examine it.

5.4.3. Setting the CA Signing Certificate Validity Period

Every certificate, including a Certificate Manager signing certificate, must have a validity period. The Certificate System does not restrict the validity period that can be specified. Set as long a validity period as possible, depending on the requirements for certificate renewal, the place of the CA in the certificate hierarchy, and the requirements of any public CAs that are included in the PKI.

A Certificate Manager cannot issue a certificate that has a validity period longer than the validity period of its CA signing certificate. If a request is made for a period longer than the CA certificate's validity period, the requested validity date is ignored and the CA signing certificate validity period is used.

5.4.4. Choosing the Signing Key Type and Length

A signing key is used by a subsystem to verify and "seal" something. CAs use a CA signing certificate to sign certificates or CRLs that it issues; OCSPs use signing certificates to verify their responses to certificate status requests; all subsystems use log file signing certificates to sign their audit logs.

The signing key must be cryptographically strong to provide protection and security for its signing operations. The following signing algorithms are considered secure:

- SHA256withRSA

- SHA512withRSA

- SHA256withEC

- SHA512withEC

NOTE

Certificate System includes native ECC support. It is also possible to load and use a third-party PKCS #11 module with ECC-enabled. This is covered in Chapter 9, *Installing an Instance with ECC System Certificates*.

Along with a key *type*, each key has a specific *bit length*. Longer keys are considered cryptographically stronger than shorter keys. However, longer keys require more time for signing operations.

The default RSA key length in the configuration wizard is 2048 bits; for certificates that provide access to highly sensitive data or services, consider increasing the length to 4096 bits. ECC keys are much stronger than RSA keys, so the recommended length for ECC keys is 256 bits, which is equivalent in strength to a 2048-bit RSA key.

5.4.5. Using Certificate Extensions

An X.509 v3 certificate contains an extension field that permits any number of additional fields to be added to the certificate. Certificate extensions provide a way of adding information such as alternative subject names and usage restrictions to certificates. Older Netscape servers, such as Red Hat Directory Server and Red Hat Certificate System, require Netscape-specific extensions because they were developed before PKIX part 1 standards were defined.

The X.509 v1 certificate specification was originally designed to bind public keys to names in an X.500 directory. As certificates began to be used on the Internet and extranets and directory lookups could not always be performed, problem areas emerged that were not covered by the original specification.

- *Trust*. The X.500 specification establishes trust by means of a strict directory hierarchy. By contrast, Internet and extranet deployments frequently involve distributed trust models that do not conform to the hierarchical X.500 approach.
- *Certificate use*. Some organizations restrict how certificates are used. For example, some certificates may be restricted to client authentication only.
- *Multiple certificates*. It is not uncommon for certificate users to possess multiple certificates with identical subject names but different key material. In this case, it is necessary to identify which key and certificate should be used for what purpose.
- *Alternate names*. For some purposes, it is useful to have alternative subject names that are also bound to the public key in the certificate.
- *Additional attributes*. Some organizations store additional information in certificates, such as when it is not possible to look up information in a directory.
- *Relationship with CA*. When certificate chaining involves intermediate CAs, it is useful to have information about the relationships among CAs embedded in their certificates.
- *CRL checking*. Since it is not always possible to check a certificate's revocation status against a directory or with the original certificate authority, it is useful for certificates to include information about where to check CRLs.

The X.509 v3 specification addressed these issues by altering the certificate format to include additional information within a certificate by defining a general format for certificate extensions and specifying extensions that can be included in the certificate. The extensions defined for X.509 v3 certificates enable additional attributes to be associated with users or public keys and manage the certification hierarchy. The *Internet X.509 Public Key Infrastructure Certificate and CRL Profile* recommends a set of extensions to use for Internet certificates and standard locations for certificate or CA information. These extensions are called *standard extensions*.

NOTE

For more information on standard extensions, see RFC 2459, RFC 3280, and RFC 3279.

The X.509 v3 standard for certificates allows organizations to define custom extensions and include them in certificates. These extensions are called *private*, *proprietary*, or *custom* extensions, and they carry information unique to an organization or business. Applications may not able to validate certificates that contain private critical extensions, so it not recommended that these be used in wide-spread situations.

The X.500 and X.509 specifications are controlled by the International Telecommunication Union (ITU), an international organization that primarily serves large telecommunication companies, government organizations, and other entities concerned with the international telecommunications network. The Internet Engineering Task Force (IETF), which controls many of the standards that underlie the Internet, is currently developing public-key infrastructure X.509 (PKIX) standards. These proposed standards further refine the X.509 v3 approach to extensions for use on the Internet. The recommendations for certificates and CRLs have reached proposed standard status and are in a document referred to as *PKIX Part 1*.

Two other standards, Abstract Syntax Notation One (ASN.1) and Distinguished Encoding Rules (DER), are used with Certificate System and certificates in general. These are specified in the CCITT Recommendations X.208 and X.209. For a quick summary of ASN.1 and DER, see *A Layman's Guide to a Subset of ASN.1, BER, and DER*, which is available at RSA Laboratories' web site, http://www.rsa.com.

5.4.5.1. Structure of Certificate Extensions

In RFC 3280, an X.509 certificate extension is defined as follows:

```
Extension  ::=  SEQUENCE  {

extnID OBJECT IDENTIFIER,
critical BOOLEAN DEFAULT FALSE,
extnValue OCTET STRING  }
```

The means a certificate extension consists of the following:

- The object identifier (OID) for the extension. This identifier uniquely identifies the extension. It also determines the ASN.1 type of value in the value field and how the value is interpreted. When an extension appears in a certificate, the OID appears as the extension ID field (**extnID**) and the corresponding ASN.1 encoded structure appears as the value of the octet string (**extnValue**).
- A flag or Boolean field called **critical**.

The value, which can be either **true** or **false**, assigned to this field indicates whether the extension is critical or noncritical to the certificate.

- If the extension is critical and the certificate is sent to an application that does not understand the extension based on the extension's ID, the application must reject the certificate.
- If the extension is not critical and the certificate is sent to an application that does not understand the extension based on the extension's ID, the application can ignore the extension and accept the certificate.

- An octet string containing the DER encoding of the value of the extension.

Typically, the application receiving the certificate checks the extension ID to determine if it can recognize the ID. If it can, it uses the extension ID to determine the type of value used.

Some of the standard extensions defined in the X.509 v3 standard include the following:

- Authority Key Identifier extension, which identifies the CA's public key, the key used to sign the certificate.
- Subject Key Identifier extension, which identifies the subject's public key, the key being certified.

NOTE

Not all applications support certificates with version 3 extensions. Applications that do support these extensions may not be able to interpret some or all of these specific extensions.

5.4.6. Using and Customizing Certificate Profiles

Certificates have different types and different applications. They can be used to establish a single sign-on environment for a corporate network, to set up VPNs, to encrypt email, or to authenticate to a website. The requirements for all of these certificates can be different, just as there may also be different requirements for the same type of certificate for different kinds of users. These certificate characteristics are set in *certificate profiles*. The Certificate Manager defines a set of certificate profiles that it uses as enrollment forms when users or machines request certificates.

Certificate Profiles

A certificate profile defines everything associated with issuing a particular type of certificate, including the authentication method, the certificate content (defaults), constraints for the values of the content, and the contents of the input and output for the certificate profile. Enrollment requests are submitted to a certificate profile and are then subject to the defaults and constraints set in that certificate profile. These constraints are in place whether the request is submitted through the input form associated with the certificate profile or through other means. The certificate that is issued from a certificate profile request contains the content required by the defaults with the information required by the default parameters. The constraints provide rules for what content is allowed in the certificate.

For example, a certificate profile for user certificates defines all aspects of that certificate, including the validity period of the certificate. The default validity period can be set to two years, and a constraint can be set on the profile that the validity period for certificates requested through this certificate profile cannot exceed two years. When a user requests a

certificate using the input form associated with this certificate profile, the issued certificate contains the information specified in the defaults and will be valid for two years. If the user submits a pre-formatted request for a certificate with a validity period of four years, the request is rejected since the constraints allow a maximum of two years validity period for this type of certificate.

A set of certificate profiles have been predefined for the most common certificates issued. These certificate profiles define defaults and constraints, associate the authentication method, and define the needed inputs and outputs for the certificate profile.

Modifying the Certificate Profile Parameters

The parameters of the default certificate profiles can be modified; this includes the authentication method, the defaults, the constraints used in each profile, the values assigned to any of the parameters in a profile, the input, and the output. It is also possible to create new certificate profiles for other types of certificates or for creating more than one certificate profile for a certificate type. There can be multiple certificate profiles for a particular type of certificate to issue the same type of certificate with a different authentication method or different definitions for the defaults and constraints. For example, there can be two certificate profiles for enrollment of SSL/TLS server certificates where one certificate profile issues certificates with a validity period of six months and another certificate profile issues certificates with a validity period of two years.

An input sets a text field in the enrollment form and what kind of information needs gathered from the end entity; this includes setting the text area for a certificate request to be pasted, which allows a request to be created outside the input form with any of the request information required. The input values are set as values in the certificate. The default inputs are not configurable in the Certificate System.

An output specifies how the response page to a successful enrollment is presented. It usually displays the certificate in a user-readable format. The default output shows a printable version of the resultant certificate; other outputs set the type of information generated at the end of the enrollment, such as PKCS #7.

Policy sets are sets of constraints and default extensions attached to every certificate processed through the profile. The extensions define certificate content such as validity periods and subject name requirements. A profile handles one certificate request, but a single request can contain information for multiple certificates. A PKCS#10 request contains a single public key. One CRMF request can contain multiple public keys, meaning multiple certificate requests. A profile may contain multiple sets of policies, with each set specifying how to handle one certificate request within a CRMF request.

Certificate Profile Administration

An administrator sets up a certificate profile by associating an existing authentication plug-in, or method, with the certificate profile; enabling and configuring defaults and constraints; and defining inputs and outputs. The administrator can use the existing certificate profiles, modify the existing certificate profiles, create new certificate profiles, and delete any certificate profile that will not be used in this PKI.

Once a certificate profile is set up, it appears on the **Manage Certificate Profiles** page of the agent services page where an agent can approve, and thus enable, a certificate profile. Once the certificate profile is enabled, it appears on the **Certificate Profile** tab of the end-entities page where end entities can enroll for a certificate using the certificate profile.

The certificate profile enrollment page in the end-entities interface contains links to each certificate profile that has been enabled by the agents. When an end entity selects one of those links, an enrollment page appears containing an enrollment form specific to that

certificate profile. The enrollment page is dynamically generated from the inputs defined for the profile. If an authentication plug-in is configured, additional fields may be added to authenticate the user.

When an end entity submits a certificate profile request that is associated with an agent-approved (manual) enrollment, an enrollment where no authentication plug-in is configured, the certificate request is queued in the agent services interface. The agent can change some aspects of the enrollment, request, validate it, cancel it, reject it, update it, or approve it. The agent is able to update the request without submitting it or validate that the request adheres to the profile's defaults and constraints. This validation procedure is only for verification and does not result in the request being submitted. The agent is bound by the constraints set; they cannot change the request in such a way that a constraint is violated. The signed approval is immediately processed, and a certificate is issued.

When a certificate profile is associated with an authentication method, the request is approved immediately and generates a certificate automatically if the user successfully authenticates, all the information required is provided, and the request does not violate any of the constraints set up for the certificate profile. There are profile policies which allow user-supplied settings like subject names or validity periods. The certificate profile framework can also preserve user-defined content set in the original certificate request in the issued certificate.

The issued certificate contains the content defined in the defaults for this certificate profile, such as the extensions and validity period for the certificate. The content of the certificate is constrained by the constraints set for each default. Multiple policies (defaults and constraints) can be set for one profile, distinguishing each set by using the same value in the policy set ID. This is particularly useful for dealing with dual keys enrollment where encryption keys and signing keys are submitted to the same profile. The server evaluates each set with each request it receives. When a single certificate is issued, one set is evaluated, and any other sets are ignored. When dual-key pairs are issued, the first set is evaluated with the first certificate request, and the second set is evaluated with the second certificate request. There is no need for more than one set for issuing a single certificate or more than two sets for issuing dual-key pairs.

Guidelines for Customizing Certificate Profiles

Tailor the profiles for the organization to the real needs and anticipated certificate types used by the organization:

- Decide which certificate profiles are needed in the PKI. There should be at least one profile for each type of certificate issued. There can be more than one certificate profile for each type of certificate to set different authentication methods or different defaults and constraints for a particular type of certificate type. Any certificate profile available in the administrative interface can be approved by an agent and then used by an end entity to enroll.
- Delete any certificate profiles that will not be used.
- Modify the existing certificate profiles for specific characteristics for the company's certificates.
    - Change the defaults set up in the certificate profile, the values of the parameters set in the defaults, or the constraints that control the certificate content.
    - Change the constraints set up by changing the value of the parameters.
    - Change the authentication method.

- Change the inputs by adding or deleting inputs in the certificate profile, which control the fields on the input page.
- Add or delete the output.

5.4.6.1. Adding SAN Extensions to the SSL Server Certificate

Certificate System enables adding Subject Alternative Name (SAN) extensions to the SSL server certificate during the installation of non-root CA or other Certificate System instances. To do so, follow the instructions in the **/usr/share/pki/ca/profiles/ca/caInternalAuthServerCert.cfg** file and add the following parameters to the configuration file supplied to the **pkispawn** utility:

pki_san_inject
: Set the value of this parameter to **True**.

pki_san_for_server_cert
: Provide a list of the required SAN extensions separated by commas (,).

For example:

```
pki_san_inject=True
pki_san_for_server_cert=intca01.example.com,intca02.example.com,intca.exam
ple.com
```

5.4.7. Planning Authentication Methods

As implied in Section 5.4.6, “Using and Customizing Certificate Profiles”, *authentication* for the certificate process means the way that a user or entity requesting a certificate proves that they are who they say they are. There are three ways that the Certificate System can authenticate an entity:

- In *agent-approved* enrollment, end-entity requests are sent to an agent for approval. The agent approves the certificate request.
- In *automatic* enrollment, end-entity requests are authenticated using a plug-in, and then the certificate request is processed; an agent is not involved in the enrollment process.
- In *CMC enrollment*, a third party application can create a request that is signed by an agent and then automatically processed.

A Certificate Manager is initially configured for agent-approved enrollment and for CMC authentication. Automated enrollment is enabled by configuring one of the authentication plug-in modules. More than one authentication method can be configured in a single instance of a subsystem. The HTML registration pages contain hidden values specifying the method used. With certificate profiles, the end-entity enrollment pages are dynamically-generated for each enabled profile. The authentication method associated with this certificate profile is specified in the dynamically-generated enrollment page.

The authentication process is simple.

1. An end entity submits a request for enrollment. The form used to submit the request identifies the method of authentication and enrollment. All HTML forms are

dynamically-generated by the profiles, which automatically associate the appropriate authentication method with the form.

2. If the authentication method is an agent-approved enrollment, the request is sent to the request queue of the CA agent. If the automated notification for a request in queue is set, an email is sent to the appropriate agent that a new request has been received. The agent can modify the request as allowed for that form and the profile constraints. Once approved, the request must pass the certificate profiles set for the Certificate Manager, and then the certificate is issued. When the certificate is issued, it is stored in the internal database and can be retrieved by the end entity from the end-entities page by serial number or by request ID.

3. If the authentication method is automated, the end entity submits the request along with required information to authenticate the user, such as an LDAP user name and password. When the user is successfully authenticated, the request is processed without being sent to an agent's queue. If the request passes the certificate profile configuration of the Certificate Manager, the certificate is issued and stored in the internal database. It is delivered to the end entity immediately through the HTML forms.

The requirements for how a certificate request is authenticated can have a direct impact on the necessary subsystems and profile settings. For example, if an agent-approved enrollment requires that an agent meet the requester in person and verify their identity through supported documentation, the authentication process can be time-intensive, as well as constrained by the physical availability of both the agent and the requester.

5.4.8. Publishing Certificates and CRLs

A CA can publish both certificates and CRLs. Certificates can be published to a plain file or to an LDAP directory; CRLs can be published to file or an LDAP directory, as well, and can also be published to an OCSP responder to handle certificate verification.

Configuring publishing is fairly straightforward and is easily adjusted. For continuity and accessibility, though, it is good to plan out where certificates and CRLs need to be published and what clients need to be able to access them.

Publishing to an LDAP directory requires special configuration in the directory for publishing to work:

- If certificates are published to the directory, than every user or server to which a certificate is issued must have a corresponding entry in the LDAP directory.

- If CRLs are published to the directory, than they must be published to an entry for the CA which issued them.

- For SSL/TLS, the directory service has to be configured in SSL/TLS and, optionally, be configured to allow the Certificate Manager to use certificate-based authentication.

- The directory administrator should configure appropriate access control rules to control DN (entry name) and password based authentication to the LDAP directory.

5.4.9. Renewing or Reissuing CA Signing Certificates

When a CA signing certificate expires, all certificates signed with the CA's corresponding signing key become invalid. End entities use information in the CA certificate to verify the

certificate's authenticity. If the CA certificate itself has expired, applications cannot chain the certificate to a trusted CA.

There are two ways of resolving CA certificate expiration:

- *Renewing* a CA certificate involves issuing a new CA certificate with the same subject name and public and private key material as the old CA certificate, but with an extended validity period. As long as the new CA certificate is distributed to all users before the old CA certificate expires, renewing the certificate allows certificates issued under the old CA certificate to continue working for the full duration of their validity periods.

- *Reissuing* a CA certificate involves issuing a new CA certificate with a new name, public and private key material, and validity period. This avoids some problems associated with renewing a CA certificate, but it requires more work for both administrators and users to implement. All certificates issued by the old CA, including those that have not yet expired, must be renewed by the new CA.

There are problems and advantages with either renewing or reissuing a CA certificate. Begin planning the CA certificate renewal or re-issuance before installing any Certificate Managers, and consider the ramifications the planned procedures may have for extensions, policies, and other aspects of the PKI deployment.

NOTE

Correct use of extensions, for example the **`authorityKeyIdentifier`** extension, can affect the transition from an old CA certificate to a new one.

5.5. PLANNING FOR NETWORK AND PHYSICAL SECURITY

When deploying any Certificate System subsystem, the physical and network security of the subsystem instance has to be considered because of the sensitivity of the data generated and stored by the subsystems.

5.5.1. Considering Firewalls

There are two considerations about using firewalls with Certificate System subsystems:

- Protecting sensitive subsystems from unauthorized access

- Allowing appropriate access to other subsystems and clients outside of the firewall

The CA, KRA, and TKS are always placed inside a firewall because they contain critical information that can cause devastating security consequences if they are compromised.

The TPS and OCSP can be placed outside the firewall. Likewise, other services and clients used by the Certificate System can be on a different machine outside the firewall. In that case, the local networks have to be configured to allow access between the subsystems behind the firewall and the services outside it.

The LDAP database can be on a different server, even on a different network, than the subsystem which uses it. In this case, all LDAP ports (**389** for LDAP and **636** for LDAPS, by default) need to be open in the firewall to allow traffic to the directory service. Without access to the LDAP database, all subsystem operations can fail.

As part of configuring the firewalls, if iptables is enabled, then it must have configured

policies to allow communication over the appropriate Certificate System ports. Configuring iptables is described in the *Red Hat Security Guide*.

5.5.2. Considering Physical Security and Location

Because of the sensitive data they hold, consider keeping the CA, KRA, and TKS in a secure facility with adequate physical access restrictions. Just as network access to the systems needs to be restricted, the physical access also needs to be restricted.

Along with finding a secure location, consider the proximity to the subsystem agents and administrators. Key recovery, for example, can require multiple agents to give approval; if these agents are spread out over a large geographical area, then the time differences may negatively impact the ability to retrieve keys. Plan the physical locations of the subsystems, then according to the locations of the agents and administrators who will manage the subsystem.

5.5.3. Planning Ports

Each Certificate System server uses up to four ports:

- A non-secure HTTP port for end entity services that do not require authentication
- A secure HTTP port for end entity services, agent services, administrative console, and admin services that require authentication
- A Tomcat Server Management port
- A Tomcat AJP Connector Port

All of the service pages and interfaces described in Section 2.7, “Red Hat Certificate System User Interfaces” are connected to using the instance's URL and the corresponding port number. For example:

```
https://server.example.com:8443/ca/ee/ca
```

To access the admin console, the URL specifies the admin port:

```
pkiconsole https://server.example.com:8443/ca
```

All agent and admin functions require SSL/TLS client authentication. For requests from end entities, the Certificate System listens on both the SSL/TLS (encrypted) port and non-SSL/TLS ports.

The ports are defined in the **server.xml** file. If a port is not used, it can be disabled in that file. For example:

```
<Service name="Catalina">
    <!--Connector port="8080" ... /-->  unused standard port
    <Connector port="8443" ... />
```

Whenever a new instance in installed, make sure that the new ports are unique on the host system.

To verify that a port is available for use, check the appropriate file for the operating system. Port numbers for network-accessible services are usually maintained in a file

named **services**. On Red Hat Enterprise Linux, it is also helpful to confirm that a port is not assigned by SELinux, by running the command **semanage port -l** to list all ports which currently have an SELinux context.

When a new subsystem instance is created, any number between 1 and 65535 can be specified as the secure port number.

5.6. TOKENS FOR STORING CERTIFICATE SYSTEM SUBSYSTEM KEYS AND CERTIFICATES

A *token* is a hardware or software device that performs cryptographic functions and stores public-key certificates, cryptographic keys, and other data. The Certificate System defines two types of tokens, *internal* and *external*, for storing key pairs and certificates that belong to the Certificate System subsystems.

An internal (software) token is a pair of files, usually called the *certificate database* (**cert8.db**) and *key database* (**key3.db**), that the Certificate System uses to generate and store its key pairs and certificates. The Certificate System automatically generates these files in the filesystem of its host machine when first using the internal token. These files are created during the Certificate System subsystem configuration if the internal token was selected for key-pair generation.

These security databases are located in the **/var/lib/pki/*instance_name*/alias** directory.

An external token refers to an external hardware device, such as a smart card or hardware security module (HSM), that the Certificate System uses to generate and store its key pairs and certificates. The Certificate System supports any hardware tokens that are compliant with PKCS #11.

PKCS #11 is a standard set of APIs and shared libraries which isolate an application from the details of the cryptographic device. This enables the application to provide a unified interface for PKCS #11-compliant cryptographic devices.

The PKCS #11 module implemented in the Certificate System supports cryptographic devices supplied by many different manufacturers. This module allows the Certificate System to plug in shared libraries supplied by manufacturers of external encryption devices and use them for generating and storing keys and certificates for the Certificate System managers.

Consider using external tokens for generating and storing the key pairs and certificates used by Certificate System. These devices are another security measure to safeguard private keys because hardware tokens are sometimes considered more secure than software tokens.

Before using external tokens, plan how the external token is going to be used with the subsystem:

- All system keys for a subsystem must be generated on the same token.
- The subsystem must be installed in an empty HSM slot. If the HSM slot has previously been used to store other keys, then use the HSM vendor's utilities to delete the contents of the slot. The Certificate System has to be able to create certificates and keys on the slot with default nicknames. If not properly cleaned up, the names of these objects may collide with previous instances.

The Certificate System can also use *hardware cryptographic accelerators* with external tokens. Many of the accelerators provide the following security features:

- Fast SSL/TLS connections. Speed is important to accommodate a high number of simultaneous enrollment or service requests.
- Hardware protection of private keys. These devices behave like smart cards by not allowing private keys to be copied or removed from the hardware token. This is important as a precaution against key theft from an active attack of an online Certificate Manager.

The Certificate System supports the nCipher nShield hardware security module (HSM), by default. Certificate System-supported HSMs are automatically added to the **secmod.db** database with **modutil** during the pre-configuration stage of the installation, if the PKCS #11 library modules are in the default installation paths.

During configuration, the **Security Modules** panel displays the supported modules, along with the NSS internal software PKCS #11 module. All supported modules that are detected show a status of **Found** and is individually marked as either **Logged in** or **Not logged in**. If a token is found but not logged in, it is possible to log in using the **Login** under **Operations**. If the administrator can log into a token successfully, the password is stored in a configuration file. At the next start or restart of the Certificate System instance, the passwords in the password store are used to attempt a login for each corresponding token.

Administrators are allowed to select any of the tokens that are logged in as the default token, which is used to generate system keys.

5.7. A CHECKLIST FOR PLANNING THE PKI

Q: How many security domains will be created, and what subsystem instances will be placed in each domain?

A: A subsystem can only communicate with another subsystem if they have a trusted relationship. Because the subsystems within a security domain have automatic trusted relationships with each other, it is important what domain a subsystem joins. Security domains can have different certificate issuing policies, different kinds of subsystems within them, or a different Directory Server database. Map out where (both on the physical machine and in relation to each other) each subsystem belongs, and assign it to the security domain accordingly.

Q: What ports should be assigned for each subsystem? Is it necessary to have a single SSL/TLS port, or is it better to have port separation for extra security?

A: The most secure solution is to use separate ports for each SSL/TLS interface. However, the feasibility of implementing multiple ports may depends on your network setup, firewall rules, and other network conditions.

Q: What subsystems should be placed behind firewalls? What clients or other subsystems need to access those firewall-protected subsystems and how will access be granted? Is firewall access allowed for the LDAP database?

A: The location to install a subsystem depends on the network design. The OCSP subsystem is specifically designed to operate outside a firewall for user convenience, while the CA, KRA, and TPS should all be secured behind a firewall for increased

security.

When deciding the location of the subsystems, it is critical to plan what *other* subsystems or services that server needs to access (including the internal database, a CA, or external users) and look at firewall, subnet, and VPN configuration.

Q: What subsystems need to be physically secured? How will access be granted, and who will be granted access?

A: The CA, TKS, and KRA all store extremely sensitive key and certificate information. For some deployments, it may be necessary to limit physical access to the machines these subsystems run on. In that case, both the subsystems and their host machines must be included in the larger infrastructure inventory and security design.

Q: What is the physical location of all agents and administrators? What is the physical location of the subsystems? How will administrators or agents access the subsystem services in a timely-manner? Is it necessary to have subsystems in each geographical location or time zone?

A: The physical location of Certificate System users may impact things like request approval and token enrollment, as well as system performance because of network lag time. The importance of response time for processing certificate operations should be taken into account when deciding where and how many subsystems to install.

Q: How many subsystems do you need to install?

A: The primary consideration is the expected load for each subsystem and then, secondarily, geographical or departmental divisions. Subsystems with fairly low loads (like the KRA or TKS) may only require a single instance for the entire PKI. Systems with high load (the CA) or which may benefit from local instances for local agents (like the TPS) may require multiple instances.

Subsystems can be cloned, meaning they essentially are clustered, operating as a single unit, which is good for load balancing and high availability. Cloning is especially good for CAs and KRAs, where the same key and certificate data may need to be accessed by larger numbers of users or to have reliable failover.

When planning for multiple subsystem instances, keep in mind how the subsystems fit within the established security domains. Security domains create trusted relationships between subsystems, allowing them to work together to find available subsystems to respond to immediate needs. Multiple security domains can be used in a single PKI, with multiple instances of any kind of subsystem, or a single security domain can be used for all subsystems.

Q: Will any subsystems need to be cloned and, if so, what are the methods for securely storing their key materials?

A: Cloned subsystems work together, essentially as a single instance. This can be good for high demand systems, failover, or load balancing, but it can become difficult to maintain. For example, cloned CAs have serial number ranges for the certificates they issue, and a clone could hit the end of its range.

Q: Will the subsystem certificates and keys be stored on the internal software token in Certificate System or on an external hardware token?

A: Certificate System supports two hardware security modules (HSM): nCipher nShield and Safenet LunaSA. Using a hardware token can require additional setup and configuration before installing the subsystems, but it also adds another layer of security.

Q: What are the requirements for the CA signing certificate? Does the Certificate System need control over attributes like the validity period? How will the CA certificates be distributed?

A: The real question here is, will the CA bee a root CA which sets its own rules on issuing certificates or will it be a subordinate CA where another CA (another CA in your PKI or even an external CA) sets restrictions on what kind of certificates it can issue.

A Certificate Manager can be configured as either a root CA or a subordinate CA. The difference between a root CA and a subordinate CA is who signs the CA signing certificate. A root CA signs its own certificate. A subordinate CA has another CA (either internal or external) sign its certificate.

A self-signing root CA issues and signs its own CA signing certificate. This allows the CA to set its own configuration rules, like validity periods and the number of allowed subordinate CAs.

A subordinate CA has its certificates issued by a public CA or another Certificate System root CA. This CA is *subordinate* to the other CA's rules about its certificate settings and how the certificate can be used, such as the kinds of certificates that it can issue, the extensions that it is allowed to include in certificates, and the levels of subordinate CAs the subordinate CA can create.

One option is to have the Certificate manager subordinate to a public CA. This can be very restrictive, since it introduces the restrictions that public CAs place on the kinds of certificates the subordinate CA can issue and the nature of the certificate chain. On the other hand, one benefit of chaining to a public CA is that the third party is responsible for submitting the root CA certificate to a web browser or other client software, which is a major advantage for certificates that are accessed by different companies with browsers that cannot be controlled by the administrator.

The other option is make the CA subordinate to a Certificate System CA. Setting up a Certificate System CA as the root CA means that the Certificate System administrator has control over all subordinate CAs by setting policies that control the contents of the CA signing certificates issued.

It is easiest to make the first CA installed a self-signed root, so that it is not necessary to apply to a third party and wait for the certificate to be issued. Make sure that you determine how many root CAs to have and where both root and subordinate CAs will be located.

Q: What kinds of certificates will be issued? What characteristics do they need to have, and what profile settings are available for those characteristics? What restrictions need to be placed on the certificates?

A: As touched on in Section 5.4.6, “Using and Customizing Certificate Profiles”, the profiles (the forms which configure each type of certificate issued by a CA) can be customized. This means that the subject DN, the validity period, the type of SSL/TLS client, and other elements can be defined by an administrator for a specific purpose. For security, profiles should provide only the functionality that is required by the PKI. Any unnecessary profiles should be disabled.

Q: What are the requirements for approving a certificate request? How does the requester authenticate himself, and what kind of process is required to approve the request?

A: The request approval process directly impacts the security of the certificate. Automated processes are much faster but are less secure since the identity of the requester is only superficially verified. Likewise, agent-approved enrollments are more secure (since, in the most secure scenario they can require an in-person verification and supporting identification) but they can also be the most time-consuming.

First determine how secure the identity verification process needs to be, then determine what authentication (approval) mechanism is required to validate that identity.

Q: Will many external clients need to validate certificate status? Can the internal OCSP in the Certificate Manager handle the load?

A: Publishing CRLs and validating certificates is critical. Determine what kinds of clients will be checking the certificate status (will it mainly be internal clients? external clients? will they be validating user certificates or server certificates?) and also try to determine how many OCSP checks will be run. The CA has an internal OCSP which can be used for internal checks or infrequent checks, but a large number of OCSP checks could slow down the CA performance. For a larger number of OCSP checks and especially for large CRLs, consider using a separate OCSP Manager to take the load off the CA.

Q: Will the PKI allow replacement keys? Will it require key archival and recovery?

A: If lost certificates or tokens are simply revoked and replaced, then there does not need to be a mechanism to recover them. However, when certificates are used to sign or encrypt data such as emails, files, or harddrives, then the key must always be available so that the data can be recovered. In that case, use a KRA to archive the keys so the data can always be accessed.

Q: Will the organization use smart cards? If so, will temporary smart cards be allowed if smart cards are mislaid, requiring key archival and recovery?

A: If no smart cards are used, then it is never necessary to configure the TPS or TKS, since those subsystems are only used for token management. However, if smart cards are used, then the TPS and TKS are required. If tokens can be lost and replaced, then it is also necessary to have a KRA to archive the keys so that the token's certificates can be regenerated.

Q: Where will certificates and CRLs be published? What configuration needs to be done on the receiving end for publishing to work? What kinds of certificates or CRLs need to be published and how frequently?

A: The important thing to determine is what clients need to be able to access the certificate or CRL information and how that access is allowed. From there, you cna define the publishing policy.

5.8. OPTIONAL THIRD-PARTY SERVICES

5.8.1. Load Balancers

5.8.2. Backup Hardware and Software

PART II. INSTALLING RED HAT CERTIFICATE SYSTEM

This section describes the requirements and procedures for installing Red Hat Certificate System.

CHAPTER 6. PREREQUISITES AND PREPARATION FOR INSTALLATION

The Red Hat Certificate System installation process requires some preparation of the environment. This chapter describes the requirements, dependencies, and other prerequisites when installing Certificate System subsystems.

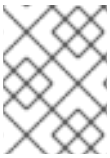
6.1. INSTALLING RED HAT ENTERPRISE LINUX

For details about installing Red Hat Enterprise Linux, see the *Red Hat Enterprise Linux Installation Guide*.

If your Certificate System instance needs to meet the Federal Information Processing Standard (FIPS), enable the FIPS mode on Red Hat Enterprise Linux. For details, see the *Red Hat Security Guide*.

6.2. SECURING THE SYSTEM USING SELINUX

Security-Enhanced Linux (SELinux) is an implementation of a mandatory access control mechanism in the Linux kernel, checking for allowed operations after standard discretionary access controls are checked. SELinux can enforce rules on files and processes in a Linux system, and on their actions, based on defined policies.



NOTE

For increased security requirements, Red Hat recommends running Certificate System with SELinux in **`enforcing`** mode.

In most cases, no actions are required to run Certificate System with SELinux in **`enforcing`** mode. If a procedure in the Certificate System documentation requires to manually set SELinux-related settings, such as when using a Hardware Security Module (HSM), it is mentioned in the corresponding section.

For further details about SELinux, see the *SELinux User's and Administrator's Guide*.

6.2.1. Verifying if SELinux is Running in Enforcing Mode

By default, after installing Red Hat Enterprise Linux, SELinux is enabled and running in **`enforcing`** mode and no further actions are required.

To display the current SELinux mode, enter:

```
# getenforce
```

6.2.2. Enabling the SELinux Enforcing Mode

If SELinux is **`disabled`** or running in **`permissive`** mode, update the settings on Red Hat Enterprise Linux to enable **`enforcing`** mode.

For details, see the *Enabling SELinux* section in the *SELinux User's and Administrator's Guide*.

6.3. FIREWALL CONFIGURATION

The following table lists the default ports used by Certificate System subsystems:

Table 6.1. Certificate System Default Ports

Service	Port	Protocol
HTTP	8080	TCP
HTTPS	8443	TCP
Tomcat Apache JServ Protocol (AJP)	8009	TCP
Tomcat Management	8005	TCP

If you use different ports, open them correspondingly in the firewall. For further details about ports, see Section 5.5.3, “Planning Ports”.

For ports required to access Directory Server, see corresponding section in the *Directory Server Installation Guide*.

6.3.1. Opening the Required Ports in the Firewall

To enable communication between the clients and Certificate System, open the required ports in your firewall:

1. Make sure the **firewalld** service is running.

```
# systemctl status firewalld
```

2. To start **firewalld** and configure it to start automatically when the system boots:

```
# systemctl start firewalld
# systemctl enable firewalld
```

3. Open the required ports using the **firewall-cmd** utility. For example, to open the Certificate System default ports in the default firewall zone:

```
# firewall-cmd --permanent --add-port=
{8080/tcp,8443/tcp,8009/tcp,8005/tcp}
```

   For details on using **firewall-cmd** to open ports on a system, see the *Red Hat Enterprise Linux Security Guide* or the firewall-cmd(1) man page.

4. Reload the firewall configuration to ensure that the change takes place immediately:

```
# firewall-cmd --reload
```

6.4. INSTALLING RED HAT DIRECTORY SERVER

Certificate System uses Red Hat Directory Server to store system certificates and user data. You can install both Directory Server and Certificate System on the same or any other host in the network.

Perform the following steps to install Red Hat Directory Server:

1. Attach a Directory Server subscription to the host.
2. Install the Directory Server packages.
3. Run the **setup-ds.pl** Perl script to set up a Directory Server instance.

For a detailed procedure, see the *Red Hat Directory Server Installation Guide*.

6.4.1. Enabling TLS Support in Directory Server

For details about enabling TLS support in Directory Server, see the *Enabling TLS in Directory Server* section in the *Directory Server Administration Guide*.

NOTE

For stronger security, Red Hat recommends that you install Certificate System with TLS enabled for the connection to Directory Server.

As described in the Directory Server documentation, you can configure TLS either using a certificate issued by an external Certificate Authority (CA) or a temporary self-signed server certificate. However, after setting up the Certificate System CA, you can use this CA to issue a certificate and replace it with the one used when you set up Directory Server. For details on how to request and issue a TLS server certificate for Directory Server, see the corresponding section in the *Red Hat Certificate System Administration Guide*.

6.5. ENABLING THE CERTIFICATE SYSTEM REPOSITORY

Before you can install and update Certificate System using the **yum** utility, you must enable the corresponding repository:

1. Attach the Red Hat subscriptions to the system:

   Skip this step, if your system is already registered or has a Certificate Server subscription attached.

   a. Register the system to the Red Hat subscription management service:

   ```
   # subscription-manager register --auto-attach
   Username: admin@example.com
   Password:
   The system has been registered with id: 566629db-a4ec-43e1-aa02-
   9cbaa6177c3f

   Installed Product Current Status:
   Product Name:           Red Hat Enterprise Linux Server
   Status:                 Subscribed
   ```

   Use the **--auto-attach** option to automatically apply a subscription for the operating system.

b. List the available subscriptions and note the pool ID providing the Red Hat Certificate System. For example:

```
# subscription-manager list --available --all
...
Subscription Name:   Red Hat Enterprise Linux Developer Suite
Provides:            ...
                     Red Hat Certificate System
                     ...
Pool ID:             7aba89677a6a38fc0bba7dac673f7993
Available:           1
...
```

In case you have a lot of subscriptions, the output of the command can be very long. You can optionally redirect the output to a file:

```
# subscription-manager list --available --all >
/root/subscriptions.txt
```

c. Attach the Certificate System subscription to the system using the pool ID from the previous step:

```
# subscription-manager attach --
pool=7aba89677a6a38fc0bba7dac673f7993
Successfully attached a subscription for: Red Hat Enterprise
Linux Developer Suite
```

2. Enable the Certificate Server repository:

```
# subscription-manager repos --enable=rhel-7-server-rhcmsys-9-rpms
Repository 'rhel-7-server-rhcmsys-9-rpms' is enabled for this
system.
```

Installing the required packages is described in the Chapter 7, *Installing and Configuring Certificate System* chapter.

6.6. SETTING UP OPERATING SYSTEM USERS AND GROUPS

When installing Certificate System, the **pkiuser** account and the corresponding **pkiuser** group are automatically created. Certificate System uses this account and group to start services. Additionally, Red Hat recommends creating additional groups to let users maintain tasks and to read the signed audit logs.

6.6.1. Creating Groups for Certificate System

Certificate Systems uses the following groups:

pkiuser

The **pkiuser** group is automatically created when installing the Certificate Systems packages and uses GID ***17***. Only the auto-created **pkiuser** account is a member of this group. Certificate Systems uses this account and group to start services. Do not add other accounts to this group.

pkiadmin

Members of this system group have full access to tasks in the agent service interface.

To create the recommended **pkiadmin** group, enter:

```
# groupadd -r pkiadmin
```

Optionally, add **sudo** rules to Red Hat Enterprise Linux to enable members of this group to read and modify Certificate System configuration files, such as **CS.cfg**, **server.xml**, and profiles. For details about configuring **sudo**, see the corresponding documentation in the *Red Hat System Administrator's Guide*.

pkiaudit

Members of this system group can read the signed audit logs.

To create the recommended **pkiaudit** group, enter:

```
# groupadd -r pkiaudit
```

Optional: A hardware token group

If the subsystem uses a hardware token, the **pkiuser** account must be a member of the hardware token group. For example, when you use the nCipher token, the **nfast** group is used to access the module.

6.6.2. Creating Users and Assigning Them to the Certificate System Groups

By adding users to the recommended **pkiadmin** and **pkiaudit** groups, you assign permissions to these accounts. For example, members of **pkiadmin** can manage tasks in the agent interface, and members of **pkiaudit** can read signed audit logs.

For example, to create a new user and assign the account to the **pkiadmin** group:

1. Create the user account:

   ```
   # useradd -m user_name
   ```

   For further details about creating user accounts, see the corresponding section in the *System Administrator's Guide*.

2. Set a password to the account:

   ```
   # passwd user_name
   ```

3. Add the account to the **pkiadmin** group:

   ```
   # usermod -a -G pkiadmin user_name
   ```

CHAPTER 7. INSTALLING AND CONFIGURING CERTIFICATE SYSTEM

Red Hat Certificate System provides different subsystems that can be installed individually. For example, you can install multiple subsystem instances on a single server or you can run them independently on different hosts. This enables you to adapt the installation to your environment to provide a higher availability, scalability, and fail-over support. This chapter describes the package installation and how to set up the individual subsystems.

The Certificate System includes the following subsystems:

- Certificate Authority (CA)
- Key Recovery Authority (KRA)
- Online Certificate Status Protocol (OCSP) Responder
- Token Key Service (TKS)
- Token Processing System (TPS)

Each subsystem is installed and configured individually as a standalone Tomcat web server instance. However, Red Hat Certificate System additionally supports running a single shared Tomcat web server instance that can contain up to one of each subsystem.

7.1. SUBSYSTEM CONFIGURATION ORDER

The order in which the individual subsystems are set up is important because of relationships between the different subsystems:

1. At least one CA running as a security domain is required before any of the other public key infrastructure (PKI) subsystems can be installed.
2. Install the OCSP after the CA has been configured.
3. The KRA, and TKS subsystems can be installed in any order, after the CA and OCSP have been configured.
4. The TPS subsystem depends on the CA and TKS, and optionally on the KRA and OCSP subsystem.

NOTE

In certain situations, administrators want to install a standalone KRA or OCSP which do not require a CA running as a security domain. For details, see Section 7.8, “Setting up a Standalone KRA or OCSP”.

7.2. CERTIFICATE SYSTEM PACKAGES

When installing the Certificate System packages you can either install them for each subsystem individually or all at once.

IMPORTANT

To install and update Certificate Server packages, you must enable the corresponding repository. For details, see Section 6.5, “Enabling the Certificate System Repository”.

The following subsystem packages and components are available:

- pki-ca: Provides the Certificate Authority (CA) subsystem.
- pki-kra: Provides the Key Recovery Authority (KRA) subsystem.
- pki-ocsp: Provides the Online Certificate Status Protocol (OCSP) responder.
- pki-tks: Provides the Token Key Service (TKS).
- pki-tps: Provides the Token Processing Service (TPS).
- pki-console and redhat-pki-console-theme: Provides the Java-based Red Hat PKI console. Both packages must be installed.
- pki-server and redhat-pki-server-theme: Provides the web-based Certificate System interface. Both packages must be installed.

  This package is installed as a dependency if you install one of the following packages: pki-ca, pki-kra, pki-ocsp, pki-tks, pki-tps

Example 7.1. Installing Certificate System Packages

- To install the CA subsystem and the optional web interface, enter:

  ```
  # yum install pki-ca redhat-pki-server-theme
  ```

  For other subsystems, replace the pki-ca package name with the one of the subsystem you want to install.

- If you require the optional PKI console:

  ```
  # yum install pki-console redhat-pki-console-theme
  ```

- Alternatively, install all Certificate System subsystem packages and components automatically:

  ```
  # yum install redhat-pki
  ```

7.3. UNDERSTANDING THE `PKISPAWN` UTILITY

In Red Hat Certificate System, you set up the individual public key infrastructure (PKI) subsystems using the **pkispawn** utility. During the setup, **pkispawn**:

1. Reads the default values from the **/etc/pki/default.cfg** file. For further details, see the pki_default.cfg(5) man page.

IMPORTANT

Do not edit the **/etc/pki/default.cfg** file. Instead, create a configuration file and that overrides the defaults, and pass it to the **pkispawn** utility. For details about using a configuration file, see Section 7.6, “Two-step Installation”.

2. Uses the passwords and other deployment-specific information provided depending on the setup mode:
   - Interactive mode: The user is asked for the individual settings during the setup. Use this mode for simple deployments.
   - Batch mode: The values are read from a configuration file the user provides. Parameters not set in the configuration file use the defaults.
3. Performs the installation of the requested PKI subsystem.
4. Passes the settings to a Java servlet that performs the configuration based on the settings.

Use the **pkispawn** utility to install:

- A root CA. For details, see Section 7.4, “Setting Up a Root Certificate Authority”.
- A subordinate CA or any other subsystem. For details, see Section 7.5, “Setting up Additional Subsystems”.

NOTE

See Section 7.4, “Setting Up a Root Certificate Authority” how to set up a root CA using the **pkispawn** utility. For a setup of a subordinate CA or non-CA subsystems, see Section 7.7, “Setting up Subsystems with an External CA”.

For further information about **pkispawn** and examples, see the pkispawn(8) man page.

7.4. SETTING UP A ROOT CERTIFICATE AUTHORITY

The Certificate Authority (CA) subsystem is the prerequisite for all other Certificate System subsystems. Therefore, set up the CA before configuring other subsystems.

To set up a root CA in Certificate System, you have the following options:

- Configuration file-based installation:

  Use this method for high-level customization. This installation method uses a configuration file that overrides the default installation parameters.

  You can install Certificate System using a configuration file in a single step or in two steps. For details and examples, see:
  - The pkispawn(8) man page for the single-step installation.
  - Section 7.6, “Two-step Installation” for the two-step installation.

- Interactive installation:

  ```
  # pkispawn -s CA
  ```

  Use the interactive installer if you only want to set the minimum of required configuration options.

7.5. SETTING UP ADDITIONAL SUBSYSTEMS

After you have installed the root Certificate Authority (CA) as described in Section 7.4, “Setting Up a Root Certificate Authority”, you can install additional Certificate System subsystems.

Prerequisites
All additional subsystems require a root Certificate Authority (CA). If you have not installed a root Certificate System CA, see Section 7.4, “Setting Up a Root Certificate Authority”.

Installing the Subsystem
To set up an additional subsystem, you have the following options:

- Configuration file-based installation:

  Use this method for high-level customization. This installation method uses a configuration file that overrides the default installation parameters.

  You can install Certificate System using a configuration file in a single step or in two steps. For details and examples, see:

  - The pkispawn(8) man page for the single-step installation.
  - Section 7.6, “Two-step Installation” for the two-step installation.

- Interactive installation:

  Use the interactive installer if you only want to set the minimum of required configuration options.

  For example:

  ```
  # pkispawn -s subsystem
  ```

  Replace *subsystem* with one of the following subsystems:**KRA**, **OCSP**, **TKS**, or **TPS**.

  The interactive installer does not support installing a subordinate CA. To install a subordinate CA, use the two-step installation. See Section 7.6, “Two-step Installation”.

7.6. TWO-STEP INSTALLATION

To customize certain configuration parameters during installation, the installation process needs be done in two steps, with the configuration between them. For this, the **pkispawn** utility enables you run the installation of a subsystem in two steps.

7.6.1. When to Use the Two-Step Installation

Use the two-step installation in scenarios such as:

- Increasing security.
- Customizing subsystem certificates.
- Customizing the cipher list in the ***sslRangeCiphers*** parameter in the **/etc/pki/*instance_name*/server.xml** file when installing a new Certificate System instance to be connected to an existing Certificate System.
- Installing CA clones, KRA, OCSP, TKS and TPS in FIPS mode.
- Installing Certificate System with a Hardware Security Module (HSM) in FIPS mode.

7.6.2. The Two Major Parts of the Two-step Installation

The two-step installation consists of the following two major parts:

1. **Installation**

   During this step, **pkispawn** copies configuration files from the**/usr/share/pki/** directory to the instance-specific **/etc/pki/*instance_name*/** directory. Additionally, **pkispwan** sets the settings based on values defined in the deployment configuration file.

   This part of the installation contains the following substeps:

   1. Section 7.6.3, “Creating the Configuration File for the First Step of the Installation”
   2. Section 7.6.4, “Starting the Installation Step”

2. **Configuration**

   During this step, **pkispawn** continues the installation based on the configuration files in the instance-specific **/etc/pki/*instance_name*/** directory.

   This part of the installation contains the following substeps:

   1. Section 7.6.5, “Customizing the Configuration Between the Installation Steps”
   2. Section 7.6.6, “Starting the Configuration Step”

7.6.3. Creating the Configuration File for the First Step of the Installation

Create a text file for the configuration settings, such as ***/root/config.txt***, and fill it with the settings described below.

IMPORTANT

This section describes a minimum configuration with Directory Server running on the same host as Certificate System. Depending on your environment, additional parameters may be necessary. For additional examples, see the *EXAMPLES* section in the pkispawn(8) man page.

For descriptions about the parameters covered in this section, see the pki_default.cfg(5) man page.

Subsystem-independent Settings

Independently of the subsystem you install, the following settings are required in the configuration file:

1. Set the passwords of the Certificate System **admin** user, the PKCS #12 file, and Directory Server:

```
[DEFAULT]
pki_admin_password=password
pki_client_pkcs12_password=password
pki_ds_password=password
```

2. To use an LDAPS connection to Directory Server running on the same host, add the following parameters to the **[DEFAULT]** section in the configuration file:

```
pki_ds_secure_connection=True
pki_ds_secure_connection_ca_pem_file=path_to_CA_or_self-
signed_certificate
```

NOTE

For security reasons, Red Hat recommends using an encrypted connection to Directory Server.

If you use a self-signed certificate in Directory Server use the following command to export it from the Directory Server's Network Security Services (NSS) database:

```
# certutil -L -d /etc/dirsrv/slapd-instance_name/ \
     -n "server-cert" -a -o /root/ds.crt
```

IMPORTANT

By default, Certificate System removes the **~/.dogtag/*instance_name*/*subsystem*/alias** client database after the installation. For security reasons, Red Hat recommends not enabling the ***pki_client_database_purge*** parameter in the configuration file. If you manually set this parameter to **True**, Certificate System does not remove the client database after the installation.

After you created the initial configuration file, add the subsystem-specific settings to it. See:

- the section called “CA Settings”
- the section called “Settings for Other Subsystems”

CA Settings

In addition to the section called “Subsystem-independent Settings”, you need the following settings to install a CA:

1. To increase security, enable random serial numbers by adding the **[CA]** section with the following setting to the configuration file:

```
[CA]
pki_random_serial_numbers_enable=true
```

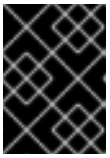
2. Optionally, set the following parameters in the **[CA]** section to specify the data of the **admin** user, which will be automatically created during the installation:

```
pki_admin_nickname=caadmin
pki_admin_name=CA administrator account
pki_admin_password=password
pki_admin_uid=caadmin
pki_admin_email=caadmin@example.com
```

   Certificate System assigns administrator privileges to this account. Use this account after the installation to manage Certificate System and to create further user accounts.

3. To enable Certificate System to generate unique nicknames, set the following parameters in the **[DEFAULT]** section:

```
pki_instance_name=instance_name
pki_security_domain_name=example.com Security Domain
pki_host=server.example.com
```



IMPORTANT

If you install Certificate System with a network-shared Hardware Security Module (HSM), you must use unique certificate nicknames.

4. Optionally, to use Elliptic Curve Cryptography (ECC) instead of RSA when generating certificates:

   a. Add the following parameters to the **[DEFAULT]** section:

```
pki_admin_key_algorithm=SHA256withEC
pki_admin_key_size=nistp256
pki_admin_key_type=ecc
pki_sslserver_key_algorithm=SHA256withEC
pki_sslserver_key_size=nistp256
pki_sslserver_key_type=ecc
pki_subsystem_key_algorithm=SHA256withEC
pki_subsystem_key_size=nistp256
pki_subsystem_key_type=ecc
```

   b. Add the following parameters to the **[CA]** section:

```
pki_ca_signing_key_algorithm=SHA256withEC
pki_ca_signing_key_size=nistp256
pki_ca_signing_key_type=ecc
pki_ca_signing_signing_algorithm=SHA256withEC
pki_ocsp_signing_key_algorithm=SHA256withEC
```

```
pki_ocsp_signing_key_size=nistp256
pki_ocsp_signing_key_type=ecc
pki_ocsp_signing_signing_algorithm=SHA256withEC
```

c. Add the following parameters to the **[CA]** section to override the RSA profiles with ECC profiles:

```
pki_source_admincert_profile=/usr/share/pki/ca/conf/ECadminCert.p
rofile
pki_source_servercert_profile=/usr/share/pki/ca/conf/ECserverCert
.profile
pki_source_subsystemcert_profile=/usr/share/pki/ca/conf/ECsubsyst
emCert.profile
```

Settings for Other Subsystems

In addition to the section called "Subsystem-independent Settings", you need the following settings to install a subordinate CA, KRA, OCSP, TKS, or TPS:

1. Add the following entry to **[DEFAULT]** section of your configuration file:

```
pki_client_database_password=password
```

2. If you are installing a TPS:

   a. Add the following section with the following section:

```
[TPS]
pki_authdb_basedn=basedn_of_the_TPS_authentication_database
```

   b. Optionally, to configure that the TPS use server-side key generation utilizing a KRA that has already been installed in the shared CA instance, add the following entry to the **[TPS]** section:

```
pki_enable_server_side_keygen=True
```

7.6.4. Starting the Installation Step

After you prepared the configuration file as described in Section 7.6.3, "Creating the Configuration File for the First Step of the Installation", start the first step of the installation:

```
# pkispawn -f /root/config.txt -s subsystem --skip-configuration
```

Replace *subsystem* with one of the following subsystems: **CA**, **KRA**, **OCSP**, **TKS**, or **TPS**.

7.6.5. Customizing the Configuration Between the Installation Steps

After the installation step described in Section 7.6.4, "Starting the Installation Step" has finished successfully, you can manually update the instance-specific configuration files before the actual configuration begins. This section provides certain examples of what you can customize between the first and second step of the installation.

7.6.5.1. Disabling Certificate Profiles

In certain situations, administrators want to disable unused enrollment profiles on the Certificate Authority (CA). To disable a certificate profile, edit the corresponding ***.cfg** file in the **/var/lib/pki/*instance_name*/ca/profiles/ca/** directory and set the ***visible*** and ***enable*** parameters to **false**.

To disable all non-CMC profiles:

1. List all non-CMC profiles:

```
# ls -l /var/lib/pki/instance_name/ca/profiles/ca/ | grep -v "CMC"
```

2. In each of the displayed files, set the following parameters to **false**:

```
visible=false
enable=false
```

Optionally, set **visible=false** in all CMC profiles to make them invisible on the end entity page:

1. List all CMC profiles:

```
# ls -l /var/lib/pki/instance_name/ca/profiles/ca/*CMC*
```

2. In each of the displayed files, set:

```
visible=false
```

7.6.5.2. Changing the Default Validity Time of Certificates

In each profile on a Certificate Authority (CA), you can set how long certificates issued using a profile are valid. You can change this value for security reasons.

For example, to set the validity of the generated Certificate Authority (CA) signing certificate to **825** days (approximately 27 months), open the **/var/lib/pki/*instance_name*/ca/profiles/ca/caCACert.cfg** file in an editor and set:

```
policyset.caCertSet.2.default.params.range=825
```

7.6.5.3. Enabling Signed Audit Logging

The signed audit logging feature enables the prevention of unauthorized log manipulation.

NOTE

Enable signed audit logging to improve security.

To enable signed audit logging, open the **/var/lib/pki/*instance_name*/*subsystem*/conf/CS.cfg** file in an editor and set:

```
log.instance.SignedAudit.logSigning=true
log.instance.SignedAudit.signedAuditCertNickname=audit_signing_certificate
```

Optionally, configure an audit filter. For details, see the *Filtering Audit Events* section in the *Red Hat Certificate System Administration Guide*. After you configured audit filtering, do not restart Certificate System at this point of the installation.

7.6.5.4. Updating the Ciphers List

In certain situations, administrators want to update the ciphers list. For example:

- To secure the Certificate System instance
- To install a Certificate System instance and to add it to an existing site that supports only specific ciphers

For details on updating the cipher list, see the corresponding section in the *Red Hat Certificate System Administration Guide*.

Default FIPS Mode Enabled Ciphers for RSA Encryption

By default, Certificate System contains the following FIPS mode enabled ciphers for RSA encryption:

- TLS_DHE_RSA_WITH_AES_128_CBC_SHA
- TLS_DHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_DHE_RSA_WITH_AES_256_CBC_SHA
- TLS_DHE_RSA_WITH_AES_256_CBC_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
- TLS_RSA_WITH_AES_128_CBC_SHA
- TLS_RSA_WITH_AES_128_CBC_SHA256
- TLS_RSA_WITH_AES_256_CBC_SHA
- TLS_RSA_WITH_AES_256_CBC_SHA256

Default FIPS Mode Enabled Ciphers for ECC Encryption

By default, Certificate System contains the following FIPS mode enabled ciphers for Elliptic Curve Cryptography (ECC) encryption:

- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256

- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
- TLS_RSA_WITH_AES_256_CBC_SHA
- TLS_RSA_WITH_AES_256_CBC_SHA256

Required RSA Ciphers When Running an HSM on System with FIPS Mode Enabled
If you install Certificate System with either LunaSA or Thales Hardware Security Module (HSM) on systems with FIPS mode enabled for RSA, disable the following ciphers, as they are unsupported on HSMs:

- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA

7.6.5.5. Configuring the PKI Console Timeout

For details, see the corresponding section in the Red Hat Certificate System Administration Guide.

Follow the procedure, but make sure to skip the steps that stop and start Certificate System.

7.6.5.6. Setting the KRA into Encryption Mode

If you are using a Hardware Security Module (HSM), it is necessary in certain situations to set the Key Recovery Authority (KRA) into encryption mode. For details, see Red Hat Certificate System Administration Guide.

7.6.5.7. Enabling OCSP

For details about enabling the Online Certificate Status Protocol (OCSP) see *Enabling Revocation Checking* in the *Red Hat Certificate System Administration Guide*.

7.6.5.8. Configuring Ranges for Requests and Serial Numbers

Specify the ranges Certificate System will use for requests and serial numbers in the **/etc/pki/*instance_name*/*subsystem*/CS.cfg** file:

```
dbs.beginRequestNumber=1001001007001
dbs.endRequestNumber=11001001007000
dbs.requestIncrement=10000000000000
dbs.requestLowWaterMark=2000000000000
dbs.requestCloneTransferNumber=10000
dbs.requestDN=ou=ca, ou=requests
dbs.requestRangeDN=ou=requests, ou=ranges
dbs.beginSerialNumber=1001001007001
dbs.endSerialNumber=11001001007000
```

```
dbs.serialIncrement=10000000000000
dbs.serialLowWaterMark=2000000000000
dbs.serialCloneTransferNumber=10000
dbs.serialDN=ou=certificateRepository, ou=ca
dbs.serialRangeDN=ou=certificateRepository, ou=ranges
dbs.beginReplicaNumber=1
dbs.endReplicaNumber=100
dbs.replicaIncrement=100
dbs.replicaLowWaterMark=20
dbs.replicaCloneTransferNumber=5
dbs.replicaDN=ou=replica
dbs.replicaRangeDN=ou=replica, ou=ranges
dbs.ldap=internaldb
dbs.newSchemaEntryAdded=true
```

NOTE

Certificate System supports **BigInteger** values for the ranges.

7.6.6. Starting the Configuration Step

After you have customized the configuration files according to Section 7.6.5, “Customizing the Configuration Between the Installation Steps”, start the second step of the installation:

```
# pkispawn -f /root/config.txt -s subsystem --skip-installation
```

Replace *subsystem* with one of the following subsystems:**CA**, **KRA**, **OCSP**, **TKS**, or **TPS**.

If the configuration step succeeds, the **pkispawn** utility displays an installation summary. For example:

```
==========================================================================
                      INSTALLATION SUMMARY
==========================================================================

  Administrator's username:             caadmin
  Administrator's PKCS #12 file:
        /root/.dogtag/instance_name/ca_admin_cert.p12

  To check the status of the subsystem:
        systemctl status pki-tomcatd@instance_name.service

  To restart the subsystem:
        systemctl restart pki-tomcatd@instance_name.service

  The URL for the subsystem is:
        https://server.example.com:8443/ca/

  PKI instances will be enabled upon system boot

==========================================================================
```

7.7. SETTING UP SUBSYSTEMS WITH AN EXTERNAL CA

7.7.1. The Difference Between an Internal and External CA

In Red Hat Certificate System, when the **pkispawn** utility sends subsystem Certificate Signing Requests (CSR) to a previously installed Certificate System, and the resulting issued certificates are received and used by **pkispawn**, the CA the CSRs were sent to is called an **Internal CA**.

An **External CA**, by contrast, can be one of the following:

- A non-Red Hat Certificate System CA that issues the signing certificate for a Certificate System subordinate CA.
- A previously installed Red Hat Certificate System CA that does not allow direct submission of CSRs. For example, this is the case if your environment requires CSRs from a subordinate CA, KRA, OCSP, TKS, or TPS to be in other formats than PKCS #10.

7.7.2. Installing a Subsystem with an External CA

This section describes how to set up a subordinate CA or other subsystems whose certificate will be signed by an external CA.

Preparing the Configuration File for the External CA Installation

Prepare the configuration file depending on whether you want your subsystem to be integrated into Certificate System or standalone:

- If you install a subsystem which is integrated into an existing Certificate System installation but which uses a certificate signed by an external CA:

  1. Create the configuration file for the subsystem. See Section 7.6.3, “Creating the Configuration File for the First Step of the Installation”.
  2. Add the following settings to your configuration file:

     - For a CA installation:

```
pki_external=True
pki_external_step_two=False

pki_storage_csr_path=/home/user_name/kra_storage.csr
pki_transport_csr_path=/home/user_name/kra_transport.csr
pki_subsystem_csr_path=/home/user_name/subsystem.csr
pki_sslserver_csr_path=/home/user_name/sslserver.csr
pki_audit_signing_csr_path=/home/user_name/kra_audit_signing.csr
pki_admin_csr_path=/home/user_name/kra_admin.csr
```

     - For a KRA installation:

```
pki_external=True
pki_external_step_two=False

pki_storage_csr_path=/home/user_name/kra_storage.csr
pki_transport_csr_path=/home/user_name/kra_transport.csr
pki_subsystem_csr_path=/home/user_name/subsystem.csr
pki_sslserver_csr_path=/home/user_name/sslserver.csr
```

```
pki_audit_signing_csr_path=/home/user_name/kra_audit_signing.csr
pki_admin_csr_path=/home/user_name/kra_admin.csr
```

- For an OCSP installation:

```
pki_external=True
pki_external_step_two=False

pki_ocsp_signing_csr_path=/home/user_name/ocsp_signing.csr
pki_subsystem_csr_path=/home/user_name/subsystem.csr
pki_sslserver_csr_path=/home/user_name/sslserver.csr
pki_audit_signing_csr_path=/home/user_name/ocsp_audit_signing.csr
pki_admin_csr_path=/home/user_name/ocsp_admin.csr
```

- If you install a standalone KRA or OCSP, which is not integrated into an existing Certificate System installation, execute the steps described in Section 7.8, “Setting up a Standalone KRA or OCSP”.

Starting the Installation of a Subsystem with an External CA

To start the installation with the configuration file:

1. Use the **pkispawn** utility to start the installation:

   ```
   # pkispawn -f /root/config.txt -s subsystem
   ```

   Replace *subsystem* with the subsystem you want to install: **CA**, **KRA**, or **OCSP**.

   During this step, the setup stores the CSRs in the files specified in the configuration.

2. Submit the CSRs to the external CA. Proceed after the CA has issued the corresponding certificates.

   In certain environments, if the external CA is also a Certificate System instance, the CSR in PKCS#10 format needs to be converted into CMC format before being submitted to the CA. See the *Issuing Certificates Using CMC* section in the *Red Hat Certificate System Administration Guide* for details about issuing the certificates.

3. Optionally, customize the installation. For details, see Section 7.6.5, “Customizing the Configuration Between the Installation Steps”.

4. After the external CA has issued the certificates, edit the deployment configuration file:

   a. Set the ***pki_external_step_two*** to **True**:

   ```
   pki_external_step_two=True
   ```

   b. Add the following parameters, based on the subsystem you are installing:

   - For a CA, set the path to the certificate file. For example:

   ```
   pki_ca_signing_cert_path=/home/user_name/ca_signing.crt
   ```

If the specified file does not contain the certificate including the certificate chain, additionally specify the path to the certificate chain file and its nickname. For example:

```
pki_cert_chain_path=/home/user_name/cert_chain.p7b
pki_cert_chain_nickname=CA Signing Certificate
```

- For a KRA, set the paths to the certificate files. For example:

```
pki_storage_cert_path=/home/user_name/kra_storage.crt
pki_transport_cert_path=/home/user_name/kra_transport.crt
pki_subsystem_cert_path=/home/user_name/subsystem.crt
pki_sslserver_cert_path=/home/user_namesslserver.crt
pki_audit_signing_cert_path=/home/user_name/kra_audit_signing.
crt
pki_admin_cert_path=/home/user_name/kra_admin.crt
```

If the specified files do not contain the certificate including the certificate chain, additionally specify the paths to the signing certificate file and the certificate chain file together with their nicknames. For example:

```
pki_ca_signing_nickname=CA Signing Certificate
pki_ca_signing_cert_path=/home/user_name/ca_signing.crt
pki_cert_chain_nickname=External Certificate Chain
pki_cert_chain_path=/home/user_name/cert_chain.p7b
```

- For an OCSP, set the paths to the certificate files. For example:

```
pki_ocsp_signing_cert_path=/home/user_name/ocsp_signing.crt
pki_subsystem_cert_path=/home/user_name/subsystem.crt
pki_sslserver_cert_path=/home/user_name/sslserver.crt
pki_audit_signing_cert_path=/home/user_name/ocsp_audit_signing
.crt
pki_admin_cert_path=/home/user_name/ocsp_admin.crt
```

If the specified files do not contain the certificate including the certificate chain, additionally specify the paths to the signing certificate file and the certificate chain file together with their nicknames. For example:

```
pki_ca_signing_nickname=CA Signing Certificate
pki_ca_signing_cert_path=/home/user_name/ca_signing.crt
pki_cert_chain_nickname=External Certificate Chain
pki_cert_chain_path=/home/user_name/cert_chain.p7b
```

5. Optionally, customize the configuration files. For examples, see Section 7.6.5, “Customizing the Configuration Between the Installation Steps”.

6. Start the configuration step:

```
# pkispawn -f /root/config.txt -s subsystem
```

Replace *subsystem* with the subsystem you want to install: **CA**, **KRA**, or **OCSP**.

7.8. SETTING UP A STANDALONE KRA OR OCSP

This section describes how to install a standalone KRA and OCSP. A standalone installation provides the flexibility to use a non-Certificate System CA to issue the certificates, because the CSRs generated during the installation are not automatically submitted to the CA and imported into the subsystem. Additionally, a KRA or an OCSP installed in standalone mode is not part of the CA's security domain, and the connector in the CA for key archival will not be configured.

To install a standalone KRA or OCSP:

1. Create a configuration file, such as **/root/config.txt**, with the following content:

```
[DEFAULT]
pki_admin_password=password
pki_client_database_password=password
pki_client_pkcs12_password=password
pki_ds_password=password
pki_token_password=password
pki_client_database_purge=False
pki_security_domain_name=EXAMPLE

pki_standalone=True
pki_external_step_two=False
```

2. For a standalone KRA, add the following section to the configuration file:

```
[KRA]
pki_admin_email=kraadmin@example.com
pki_ds_base_dn=dc=kra,dc=example,dc=com
pki_ds_database=kra

pki_admin_nickname=kraadmin
pki_audit_signing_nickname=kra_audit_signing
pki_sslserver_nickname=sslserver
pki_storage_nickname=kra_storage
pki_subsystem_nickname=subsystem
pki_transport_nickname=kra_transport

pki_standalone=True
```

3. For a standalone OCSP, add the following section to the configuration file:

```
[OCSP]
pki_admin_email=ocspadmin@example.com
pki_ds_base_dn=dc=ocsp,dc=example,dc=com
pki_ds_database=ocsp

pki_admin_nickname=ocspadmin
pki_audit_signing_nickname=ocsp_audit_signing
pki_ocsp_signing_nickname=ocsp_signing
pki_sslserver_nickname=sslserver
pki_subsystem_nickname=subsystem

pki_standalone=True
```

–

4. To use an LDAPS connection to Directory Server running on the same host, add the following parameters to the **DEFAULT** section in the configuration file:

```
pki_ds_secure_connection=True
pki_ds_secure_connection_ca_pem_file=path_to_CA_or_self-
signed_certificate
```

NOTE

For security reasons, Red Hat recommends using an encrypted connection to Directory Server.

If you use a self-signed certificate in Directory Server, use the following command to export it from the Directory Server's Network Security Services (NSS) database:

```
# certutil -L -d /etc/dirsrv/slapd-instance_name/ \
     -n "server-cert" -a -o /root/ds.crt
```

5. Proceed with the steps described in the section called “Starting the Installation of a Subsystem with an External CA”.

CHAPTER 8. USING HARDWARE SECURITY MODULES FOR SUBSYSTEM SECURITY DATABASES

A subsystem instance generates and stores its key information in a key store, called a *security module* or a *token*. A subsystem instance can be configured for the keys to be generated and stored using the internal NSS token or on a separate cryptographic device, a hardware token.

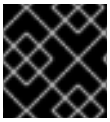
8.1. INSTALLING CERTIFICATE SYSTEM WITH AN HSM

Use the following parameters in the configuration file you pass to the **pkispawn** utility when installing Certificate System with an HSM:

```
[DEFAULT]
#########################
# Provide HSM parameters #
#########################
pki_hsm_enable=True
pki_hsm_libfile=hsm_libfile
pki_hsm_modulename=hsm_modulename
pki_token_name=hsm_token_name
pki_token_password=pki_token_password

######################################
# Provide PKI-specific HSM token names #
######################################
pki_audit_signing_token=hsm_token_name
pki_ssl_server_token=hsm_token_name
pki_subsystem_token=hsm_token_name
```

8.2. USING HARDWARE SECURITY MODULES WITH SUBSYSTEMS

The Certificate System supports the nCipher nShield hardware security module (HSM) and Gemalto Safenet LunaSA HSM by default. Certificate System-supported HSMs are automatically added to the **secmod.db** database with the **modutil** command during the pre-configuration stage of the installation, if the PKCS #11 library modules are in the specified installation paths.



IMPORTANT

Certain deployments require to setup their HSM to use FIPS mode.

8.2.1. Enabling the FIPS Mode on an nCipher HSM

To enable FIPS mode on an nCipher HSM:

1. Open the security UI:

   ```
   # /opt/nfast/bin/ksafe
   ```

2. In the **Security World** tab, select **Strict FIPS 140-2 Level III**.

3. Click **yes** to confirm.

4. Optionally, verify that FIPS mode is enabled. For details, see Section 8.2.2, “Verifying if FIPS Mode is Enabled on an nCipher HSM”.

For further details on configuring FIPS mode, see the hardware vendor documentation.

8.2.2. Verifying if FIPS Mode is Enabled on an nCipher HSM

To verify if the FIPS mode is enabled on an nCipher HSM, enter:

```
# /opt/nfast/bin/nfkminfo
```

If the **StrictFIPS140** is listed in the state flag, the FIPS mode is enabled.

8.2.3. Adding or Managing the HSM Entry for a Subsystem

During installation, when an HSM is selected as the default token where appropriate HSM-specific parameters are set in the configuration file passed to the **pkispawn** command, the following parameter is added to the **/var/lib/pki/*instance_name*/conf/password.conf** file for the HSM password:

```
hardware-HSM_token_name=HSM_token_password
```

8.2.4. Setting up SELinux for an HSM

If you want to install Certificate System with an Hardware Security Modules (HSM) and SELinux is running in **enforcing** mode, certain HSMs require that you manually update SELinux settings before you can install Certificate System.

The following section describes the required actions for supported HSMs:

nCipher nShield

After you installed the HSM and before you start installing Certificate System:

1. Reset the context of files in the **/opt/nfast/** directory:

   ```
   # restorecon -R /opt/nfast/
   ```

2. Restart the **nfast** software.

   ```
   # /opt/nfast/sbin/init.d-ncipher restart
   ```

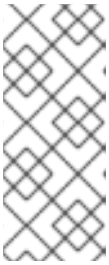
Gemalto Safenet LunaSA HSM

No SELinux-related actions are required before you start installing Certificate System.

8.2.5. Installing a Subsystem Using nCipher nShield HSM

To install a subsystem instance that use nCipher nShield HSM, follow this procedure:

1. Prepare an override file, which corresponds to your particular deployment. The following **default_hms.txt** file is an example of such an override file:



NOTE

This file serves only as an example of an nCipher HSM override configuration file -- numerous other values can be overridden including default hashing algorithms. Also, only one of the [CA], [KRA], [OCSP], [TKS], or [TPS] sections will be utilized depending upon the subsystem invocation specified on the **pkispawn** command-line.

Example 8.1. An Override File Sample to Use with nCipher HSM

```
##############################################################################
#############
##############################################################################
#############
##############################################################################
#############
##
##
## EXAMPLE: Configuration File used to override
'/etc/pki/default.cfg'       ##
##          when using an nCipher Hardware Security Module (HSM):
##
##
##
##
##
##     # modutil -dbdir . -list
##
##
##
##     Listing of PKCS #11 Modules
##
##     -----------------------------------------------------------
----           ##
##       1. NSS Internal PKCS #11 Module
##
##              slots: 2 slots attached
##
##             status: loaded
##
##
##
##              slot: NSS Internal Cryptographic Services
##
##             token: NSS Generic Crypto Services
##
##
##
##              slot: NSS User Private Key and Certificate
Services          ##
##             token: NSS Certificate DB
```

```
##
##
##
##      2. nfast
##
##             library name:
/opt/nfast/toolkits/pkcs11/libcknfast.so         ##
##              slots: 2 slots attached
##
##             status: loaded
##
##
##
##              slot: <serial_number> Rt1
##
##             token: accelerator
##
##
##
##              slot: <serial_number> Rt1 slot 0
##
##             token: <HSM_token_name>
##
##     ------------------------------------------------------
----            ##
##
##
##
##
##     Based on the example above, substitute all password values,
##
##     as well as the following values:
##
##
##
##         <hsm_libfile>=/opt/nfast/toolkits/pkcs11/libcknfast.so
##
##         <hsm_modulename>=nfast
##
##         <hsm_token_name>=NHSM6000
##
##
##
#####################################################################
#############
#####################################################################
#############
#####################################################################
#############

[DEFAULT]
##########################
# Provide HSM parameters #
##########################
pki_hsm_enable=True
pki_hsm_libfile=<hsm_libfile>
```

```
pki_hsm_modulename=<hsm_modulename>
pki_token_name=<hsm_token_name>
pki_token_password=<pki_token_password>

#######################################
# Provide PKI-specific HSM token names #
#######################################
pki_audit_signing_token=<hsm_token_name>
pki_ssl_server_token=<hsm_token_name>
pki_subsystem_token=<hsm_token_name>

#################################
# Provide PKI-specific passwords #
#################################
pki_admin_password=<pki_admin_password>
pki_client_pkcs12_password=<pki_client_pkcs12_password>
pki_ds_password=<pki_ds_password>

#####################################
# Provide non-CA-specific passwords #
#####################################
pki_client_database_password=<pki_client_database_password>

#####################################################################
# ONLY required if specifying a non-default PKI instance name #
#####################################################################
#pki_instance_name=<pki_instance_name>

#############################################################
# ONLY required if specifying non-default PKI instance ports #
#############################################################
#pki_http_port=<pki_http_port>
#pki_https_port=<pki_https_port>

#####################################################################
####
# ONLY required if specifying non-default 389 Directory Server
ports #
#####################################################################
####
#pki_ds_ldap_port=<pki_ds_ldap_port>
#pki_ds_ldaps_port=<pki_ds_ldaps_port>

#####################################################################
####
# ONLY required if PKI is using a Security Domain on a remote
system #
#####################################################################
####
#pki_ca_hostname=<pki_ca_hostname>
#pki_issuing_ca_hostname=<pki_issuing_ca_hostname>
#pki_issuing_ca_https_port=<pki_issuing_ca_https_port>
#pki_security_domain_hostname=<pki_security_domain_hostname>
#pki_security_domain_https_port=<pki_security_domain_https_port>

#############################################################
```

```
# ONLY required for PKI using an existing Security Domain #
##########################################################
# NOTE:  pki_security_domain_password == pki_admin_password
#        of CA Security Domain Instance
#pki_security_domain_password=<pki_admin_password>


[Tomcat]
##################################################################
# ONLY required if specifying non-default PKI instance ports #
##################################################################
#pki_ajp_port=<pki_ajp_port>
#pki_tomcat_server_port=<pki_tomcat_server_port>


[CA]
#######################################
# Provide CA-specific HSM token names #
#######################################
pki_ca_signing_token=<hsm_token_name>
pki_ocsp_signing_token=<hsm_token_name>

#####################################################################
#########
# ONLY required if 389 Directory Server for CA resides on a remote
system #
#####################################################################
#########
#pki_ds_hostname=<389 hostname>


[KRA]
#######################################
# Provide KRA-specific HSM token names #
#######################################
pki_storage_token=<hsm_token_name>
pki_transport_token=<hsm_token_name>

#####################################################################
##########
# ONLY required if 389 Directory Server for KRA resides on a
remote system #
#####################################################################
##########
#pki_ds_hostname=<389 hostname>


[OCSP]
#######################################
# Provide OCSP-specific HSM token names #
#######################################
pki_ocsp_signing_token=<hsm_token_name>

#####################################################################
###########
# ONLY required if 389 Directory Server for OCSP resides on a
```

```
remote system #
##########################################################################
###########
#pki_ds_hostname=<389 hostname>


[TKS]
#######################################
# Provide TKS-specific HSM token names #
#######################################

##########################################################################
##########
# ONLY required if 389 Directory Server for TKS resides on a
remote system #
##########################################################################
##########
#pki_ds_hostname=<389 hostname>


[TPS]
#################################
# Provide TPS-specific parameters #
#################################
pki_authdb_basedn=<dnsdomainname where hostname.b.c.d is
dc=b,dc=c,dc=d>

#######################################
# Provide TPS-specific HSM token names #
#######################################

##########################################################################
##########
# ONLY required if 389 Directory Server for TPS resides on a
remote system #
##########################################################################
##########
#pki_ds_hostname=<389 hostname>

#########################################################
# ONLY required if TPS requires a CA on a remote machine #
#########################################################
#pki_ca_uri=https://<pki_ca_hostname>:<pki_ca_https_port>

#######################################
# ONLY required if TPS requires a KRA #
#######################################
#pki_enable_server_side_keygen=True

#########################################################
# ONLY required if TPS requires a KRA on a remote machine #
#########################################################
#pki_kra_uri=https://<pki_kra_hostname>:<pki_kra_https_port>

#########################################################
```

```
# ONLY required if TPS requires a TKS on a remote machine #
############################################################
#pki_tks_uri=https://<pki_tks_hostname>:<pki_tks_https_port>
```

2. Use the configuration file as described in Section 7.6, “Two-step Installation”.
   - `# pkispawn -s CA -f ./default_hsm.txt -vvv`
   - `# pkispawn -s KRA -f ./default_hsm.txt -vvv`
   - `# pkispawn -s OCSP -f ./default_hsm.txt -vvv`
   - `# pkispawn -s TKS -f ./default_hsm.txt -vvv`
   - `# pkispawn -s TPS -f ./default_hsm.txt -vvv`

8.2.6. Installing a Subsystem Using Gemalto Safenet LunaSA HSM

To install a subsystem instance that use Gemalto Safenet LunaSA HSM, follow the procedure detailed in Section 8.2.5, “Installing a Subsystem Using nCipher nShield HSM”. The override file should be similar to the sample provided in Example 8.1, “An Override File Sample to Use with nCipher HSM”, differing in values related to the particular deployment. The following example provides a sample LunaSA header that is to be substituted for the header of the nCipher override file and to be used with the [DEFAULT], [Tomcat], [CA], [KRA], [OCSP], [TKS], and [TPS] sections provided in the aforementioned nCipher example.

Example 8.2. A Sample of the LunaSA Override File Header

```
##############################################################################
#######
##############################################################################
#######
##############################################################################
#######
##
##
## EXAMPLE: Configuration File used to override '/etc/pki/default.cfg'
##
##          when using a LunaSA Hardware Security Module (HSM):
##
##
##
##
##
##     # modutil -dbdir . -list
##
##
##
##     Listing of PKCS #11 Modules
##
```

```
##     ----------------------------------------------------------
##
##       1. NSS Internal PKCS #11 Module
##
##              slots: 2 slots attached
##
##             status: loaded
##
##
##
##              slot: NSS Internal Cryptographic Services
##
##             token: NSS Generic Crypto Services
##
##
##
##              slot: NSS User Private Key and Certificate Services
##
##             token: NSS Certificate DB
##
##
##
##      2. lunasa
##
##            library name:
/usr/safenet/lunaclient/lib/libCryptoki2_64.so   ##
##             slots: 4 slots attached
##
##            status: loaded
##
##
##
##             slot: LunaNet Slot
##
##            token: dev-intca
##
##
##
##             slot: Luna UHD Slot
##
##            token:
##
##
##
##             slot: Luna UHD Slot
##
##            token:
##
##
##
##             slot: Luna UHD Slot
##
##            token:
##
##     ----------------------------------------------------------
##
```

```
##
##
##
##
##     Based on the example above, substitute all password values,
##
##     as well as the following values:
##
##
##
##         <hsm_libfile>=/usr/safenet/lunaclient/lib/libCryptoki2_64.so
##
##         <hsm_modulename>=lunasa
##
##         <hsm_token_name>=dev-intca
##
##
##
#########################################################################
#######
#########################################################################
#######
#########################################################################
#######
```

8.3. BACKING UP KEYS ON HARDWARE SECURITY MODULES

It is not possible to export keys and certificates stored on an HSM to a **.p12** file. If such an instance is to be backed-up, contact the manufacturer of your HSM for support.

8.4. INSTALLING A CLONE SUBSYSTEM USING AN HSM

Normally, clone subsystems are created using a PKCS #12 file containing the system keys of the master subsystem. This file is generated during the installation of the master subsystem by setting the ***pki_backup_keys*** to **True** in the configuration file you use for the installation along with the defined value for the ***pki_backup_password*** option, or by exporting the keys using the **PKCS12Export** tool.

You cannot generate a PKCS #12 file using this tool when using an HSM because the keys cannot be retrieved from the HSM. Instead, the clone subsystem should either be pointed to the HSM used by the master subsystem, so that it can access the same keys. Alternatively, if using a separate HSM, clone the keys to this HSM using the utilities provided by the HSM vendor. You must provide access to the master subsystem's key before running the **pkispawn** utility.

As Certificate System uses no PKCS #12 when installing with an HSM, you do not need to set the parameters in the configuration file for the location of a PKCS #12 file and its password. The following example lists options from the **[Tomcat]** section of the respective PKI configuration file that are required for generating a CA clone.

For generating a non-HSM CA clone:

- **pki_clone**=True

- **pki_clone_pkcs12_password**=Secret123
- **pki_clone_pkcs12_path**=<path_to_pkcs12_file>
- **pki_clone_replicate_schema**=True (default value)
- **pki_clone_uri**=https://<master_ca_host_name>:<master_ca_https_port>

For generating a CA clone using an HSM:

- **pki_clone**=True
- **pki_clone_replicate_schema**=True (default value)
- **pki_clone_uri**=https://<master_ca_host_name>:<master_ca_https_port>

NOTE

In order for the master subsystem to share the same certificates and keys with its clones, the HSM must be on a network that is in shared mode and accessible by all subsystems.

8.5. VIEWING TOKENS

To view a list of the tokens currently installed for a Certificate System instance, use the **modutil** utility.

1. Change to the instance **alias** directory. For example:

```
# cd /var/lib/pki/pki-tomcat/alias
```

2. Show the information about the installed PKCS #11 modules installed as well as information on the corresponding tokens using the **modutil** tool.

```
# modutil -dbdir . -nocertdb -list
```

8.6. DETECTING TOKENS

To see if a token can be detected by Certificate System, use the **TokenInfo** utility, pointing to the **alias** directory for the subsystem instance. This is a Certificate System tool which is available after the Certificate System packages are installed.

For example:

```
# TokenInfo /var/lib/pki/pki-tomcat/alias
```

This utility returns all tokens which can be detected by the Certificate System, not only tokens which are installed in the Certificate System.

8.7. FAILOVER AND RESILIENCE

Failover means setting up multiple units, and configuring them so that if one unit fails, another one will take over and continue the service without interruption.

Resilience ensures that when the network connection to a unit is interrupted and then reconnected, the service is not interrupted within a reasonable timeframe.

Some Hardware Security Module (HSM) models offer failover or resilience of varying degrees. For detail on the exact make and models and the features that they offer, consult your HSM manual, or contact the manufacturer. The HSMs described in the following sections have been tested with Red Hat Certificate System.

8.7.1. nCipher nShield HSM

8.7.1.1. Failover

With Thales nShield Connect 6000, failover has been tested in the scenario where there are two HSM modules, nShield1, and nShield2, both running and configured for failover.

If one of nShield units goes down, the other exhibits ability to continue the provision of cryptographic services to Certificate System with no known issues, without restarting of the RHCS instance.

When the above situation happens (one HSM unit goes down), the administrator is expected to schedule a downtime for all the connected Certificate System instances and fix the down hsm unit and bring it back up and restart the instances. This means that if one unit goes down, Certificate System is expected to continue functioning; however, if the down hsm is brought back up without restarting the instances, the newly brought up HSM unit is not expected to be part of the failover scheme as originally planned.

8.7.1.2. Resilience

With Thales nShield Connect 6000, testing has shown that when the network cable is pulled off the HSM unit, and replugged in within up to 90 minutes, the service continues. There is no data for any time period longer than 90 minutes.

8.7.2. Gemalto Safenet LunaSA HSM

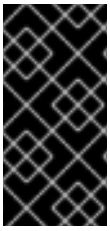
8.7.2.1. Failover

The Gemalto Safenet LunaSA Cloning model offers Failover. However, we have no data on this model.

CHAPTER 9. INSTALLING AN INSTANCE WITH ECC SYSTEM CERTIFICATES

Elliptic curve cryptography (ECC) may be preferred over RSA-style encryption in some cases, as it allows it to use much shorter key lengths and makes it faster to generate certificates. CAs which are ECC-enabled can issue both RSA and ECC certificates, using their ECC signing certificate.

Certificate System includes native support for ECC features; the support is enabled by default starting from NSS 3.16. It is also possible to load and use a third-party PKCS #11 module, such as a hardware security module (HSM). To use the ECC module, it must be loaded before the subsystem instance is configured.



IMPORTANT

Third-party ECC modules must have an SELinux policy configured for them, or SELinux needs to be changed from **enforcing** mode to **permissive** mode to allow the module to function. Otherwise, any subsystem operations which require the ECC module will fail.

9.1. LOADING A THIRD-PARTY ECC MODULE

Loading a third-party ECC module follows the same principles as loading HSMs supported by Certificate System, which is described in Chapter 8, *Using Hardware Security Modules for Subsystem Security Databases*. See this chapter for more information.

9.2. USING ECC WITH AN HSM

The HSMs supported by Certificate System support their own native ECC modules. To create an instance with ECC system certificates:

1. Set up the HSM per manufacturer's instructions. If multiple hosts are sharing the HSM, make sure they can all access the same partition if needed and if the site policies allow it.

2. Define the required parameters in the **pkispawn** utility configuration file and run **pkispawn**. For example, to configure Certificate System to create an ECC CA, assuming the configuration file is **ecc.inf**:

    1. Edit **ecc.inf** to specify the appropriate settings. For an example of the configuration file, see the pkispawn(8) man page.

    2. Run **pkispawn** against **ecc.inf**:

    ```
    $ script -c 'pkispawn -s CA -f /root/pki/ecc.inf -vvv'
    ```

CHAPTER 10. CLONING SUBSYSTEMS

When a new subsystem instance is first configured, the Red Hat Certificate System allows subsystems to be cloned, or duplicated, for high availability of the Certificate System. The cloned instances run on different machines to avoid a single point of failure and their databases are synchronized through replication.

The master CA and its clones are functionally identical, they only differ in serial number assignments and CRL generation. Therefore, this chapter refers to master or any of its clones as *replicated CAs*.

10.1. BACKING UP SUBSYSTEM KEYS FROM A SOFTWARE DATABASE

Ideally, the keys for the master instance are backed up when the instance is first created. If the keys were not backed up then or if the backup file is lost, then it is possible to extract the keys from the internal software database for the subsystem instance using the **PKCS12Export** utility. For example:

```
PKCS12Export -debug -d /var/lib/pki/instance_name/alias -w p12pwd.txt -p
internal.txt -o master.p12
```

Then copy the PKCS #12 file to the clone machine to be used in the clone instance configuration. For more details, see Section 2.8.4, “Cloning and Key Stores”.

NOTE

Keys cannot be exported from an HSM. However, in a typical deployment, HSMs support networked access, as long as the clone instance is installed using the same HSM as the master. If both instances use the same key store, then the keys are naturally available to the clone.

If backing up keys from the HSM is required, contact the HSM manufacturer for assistance.

10.2. CLONING A CA

1. Configure the master CA, and back up the keys.
2. In the **CS.cfg** file for the master CA, enable the master CA to monitor replication database changes by adding the **ca.listenToCloneModifications** parameter:

   ```
   ca.listenToCloneModifications=true
   ```

3. Create the clone subsystem instance.

   For examples of the configuration file required by **pkispawn** when cloning CA subsystems, see the **Installing a CA clone** and **Installing a CA clone on the same host** sections of the pkispawn(8) man page.

4. Restart the Directory Server instance used by the clone.

   ```
   systemctl restart pki-tomcatd@kra-clone-ds-instance.service
   ```

NOTE

Restarting the Directory Server reloads the updated schema, which is required for proper performance.

5. Restart the clone instance.

```
systemctl restart pki-tomcatd@instance_name.service
```

After configuring the clone, test to make sure that the master-clone relationship is functioning:

1. Request a certificate from the cloned CA.
2. Approve the request.
3. Download the certificate to the browser.
4. Revoke the certificate.
5. Check master CA's CRL for the revoked certificate. In the master Certificate Manager's agent services page, click **Update Certificate Revocation List**. Find the CRL in the list.

   The CRL should show the certificate revoked by the cloned Certificate Manager. If that certificate is not listed, check logs to resolve the problem.

10.3. UPDATING CA-KRA CONNECTOR INFORMATION AFTER CLONING

As covered in Section 2.8.7, “Custom Configuration and Clones”, configuration information is not updated in clone instances if it is made *after* the clone is created. Likewise, changes made to a clone are not copied back to the master instance.

If a new KRA is installed or cloned after a clone CA is created, then the clone CA does not have the new KRA connector information in its configuration. This means that the clone CA is not able to send any archival requests to the KRA.

Whenever a new KRA is created or cloned, copy its connector information into all of the *cloned* CAs in the deployment. To do this, use the **pki ca-kraconnector-add** command.

If it is required to do this manually, follow these steps:

1. On the master clone machine, open the master CA's **CS.cfg** file, and copy all of the ***ca.connector.KRA.**** lines for the new KRA connector.

```
[root@master ~]# vim /var/lib/pki/pki-tomcat/ca/conf/CS.cfg
```

2. Stop the clone CA instance. For example:

```
[root@clone-ca ~] systemctl stop pki-tomcatd@instance_name.service
```

3. Open the clone CA's **CS.cfg** file.

```
[root@clone-ca ~]# vim /var/lib/pki/pki-tomcat/ca/conf/CS.cfg
```

4. Copy in the connector information for the new KRA instance or clone.

```
ca.connector.KRA.enable=true ca.connector.KRA.host=server-
kra.example.com
ca.connector.KRA.local=false ca.connector.KRA.nickName=subsystemCert
cert-pki-ca
ca.connector.KRA.port=10444 ca.connector.KRA.timeout=30
ca.connector.KRA.transportCert=MIIDbD...ZR0Y2zA==
ca.connector.KRA.uri=/kra/agent/kra/connector
```

5. Start the clone CA.

```
[root@clone-ca ~] systemctl start pki-tomcatd@instance_name.service
```

10.4. CLONING OCSP SUBSYSTEMS

1. Configure the master OCSP, and back up the keys.

2. In the **CS.cfg** file for the master OCSP, set the **OCSP.Responder.store.defStore.refreshInSec** parameter to any non-zero number other than 21600; 21600 is the setting for a clone.

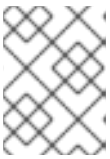
```
vim /etc/instance_name/CS.cfg

OCSP.Responder.store.defStore.refreshInSec=15000
```

3. Create the clone subsystem instance using the **pkispawn** utility.

   For examples of the configuration file required by **pkispawn** when cloning OCSP subsystems, see the pkispawn(8) man page.

4. Restart the Directory Server instance used by the clone.

```
systemctl dirsrv@instance_name.service
```



NOTE

Restarting the Directory Server reloads the updated schema, which is required for proper performance.

5. Restart the clone instance.

```
systemctl restart pki-tomcatd@instance_name.service
```

After configuring the clone, test to make sure that the master-clone relationship is functioning:

1. Set up OCSP publishing in the master CA so that the CRL is published to the master OCSP.

2. Once the CRL is successfully published, check both the master and cloned OCSP's **List Certificate Authorities** link in the agent pages. The list should be identical.

3. Use the **OCSPClient** tool to submit OCSP requests to the master and the cloned Online Certificate Status Manager. The tool should receive identical OCSP responses from both managers.

10.5. CLONING KRA SUBSYSTEMS

1. Configure the master subsystem, and back up the keys.

2. Create the clone subsystem instance using the **pkispawn** utility.

   For examples of the configuration file required by **pkispawn** when cloning KRA subsystems, see the **Installing a KRA or TPS clone** section of the pkispawn(8) man page.

3. Restart the Directory Server instance used by the clone.

   ```
   systemctl dirsrv@instance_name.service
   ```

   NOTE

   Restarting the Directory Server reloads the updated schema, which is required for proper performance.

4. Restart the clone instance.

   ```
   systemctl restart pki-tomcatd@instance_name.service
   ```

For the KRA clone, test to make sure that the master-clone relationship is functioning:

1. Go to the KRA agent's page.

2. Click **List Requests**.

3. Select **Show all requests** for the request type and status.

4. Click **Submit**.

5. Compare the results from the cloned KRA and the master KRA. The results ought to be identical.

10.6. CLONING TKS SUBSYSTEMS

1. Configure the master subsystem, and back up the keys.

2. Create the clone subsystem instance using the **pkispawn** utility.

   For examples of the configuration file required by **pkispawn** when cloning TKS subsystems, see the **Installing a KRA or TKS clone** section of the pkispawn(8) man page.

3. Restart the clone instance.

```
systemctl restart pki-tomcatd@instance_name.service
```

For the TKS, enroll a smart card and then run an **ldapsearch** to make sure that the same key information is contained in both databases.

10.7. CONVERTING MASTERS AND CLONES

Only one single active CA generating CRLs can exist within the same topology. Similarly, only one OCSP receiving CRLs can exist within the same topology. As such, there can be any number of clones, but there can only be a single configured master for CA and OCSP.

For KRAs and TKSs, there is no configuration difference between masters and clones, but CAs and OCSPs do have some configuration differences. This means that when a master is taken offline — because of a failure or for maintenance or to change the function of the subsystem in the PKI — then the existing master must be reconfigured to be a clone, and one of the clones promoted to be the master.

10.7.1. Converting CA Clones and Masters

1. Stop the master CA if it is still running.
2. Open the existing master CA configuration directory:

```
cd /var/lib/pki/pki-tomcat/ca/conf
```

3. Edit the **CS.cfg** file for the master, and change the CRL and maintenance thread settings so that it is set as a clone:
   - Disable control of the database maintenance thread:

```
ca.certStatusUpdateInterval=0
```

   - Disable monitoring database replication changes:

```
ca.listenToCloneModifications=false
```

   - Disable maintenance of the CRL cache:

```
ca.crl.IssuingPointId.enableCRLCache=false
```

   - Disable CRL generation:

```
ca.crl.IssuingPointId.enableCRLUpdates=false
```

   - Set the CA to redirect CRL requests to the new master:

```
master.ca.agent.host=new_master_hostname
master.ca.agent.port=new_master_port
```

4. Stop the cloned CA server.

```
systemctl stop pki-tomcatd@instance_name.service
```

5. Open the cloned CA's configuration directory.

```
cd /etc/instance_name
```

6. Edit the **CS.cfg** file to configure the clone as the new master.

    1. Delete each line which begins with the **ca.crl.** prefix.

    2. Copy each line beginning with the **ca.crl.** prefix from the former master CA **CS.cfg** file into the cloned CA's **CS.cfg** file.

    3. Enable control of the database maintenance thread; the default value for a master CA is **600**.

    ```
    ca.certStatusUpdateInterval=600
    ```

    4. Enable monitoring database replication:

    ```
    ca.listenToCloneModifications=true
    ```

    5. Enable maintenance of the CRL cache:

    ```
    ca.crl.IssuingPointId.enableCRLCache=true
    ```

    6. Enable CRL generation:

    ```
    ca.crl.IssuingPointId.enableCRLUpdates=true
    ```

    7. Disable the redirect settings for CRL generation requests:

    ```
    master.ca.agent.host=hostname
    master.ca.agent.port=port number
    ```

7. Start the new master CA server.

```
systemctl start pki-tomcatd@instance_name.service
```

10.7.2. Converting OCSP Clones

1. Stop the OCSP master, if it is still running.

2. Open the existing master OCSP configuration directory.

```
cd /etc/instance_name
```

3. Edit the **CS.cfg**, and reset the **OCSP.Responder.store.defStore.refreshInSec** parameter to **21600**:

```
OCSP.Responder.store.defStore.refreshInSec=21600
```

4. Stop the online cloned OCSP server.

```
systemctl stop pki-tomcatd@instance_name.service
```

5. Open the cloned OCSP responder's configuration directory.

```
cd /etc/instance_name
```

6. Open the **CS.cfg** file, and delete the **OCSP.Responder.store.defStore.refreshInSec** parameter or change its value to any non-zero number:

```
OCSP.Responder.store.defStore.refreshInSec=15000
```

7. Start the new master OCSP responder server.

```
systemctl start pki-tomcatd@instance_name.service
```

10.8. CLONING A CA THAT HAS BEEN RE-KEYED

When a certificate expires, it has to be replaced. This can either be done by renewing the certificate, which re-uses the original keypair to generate a new certificate, or it can be done by generating a new keypair and certificate. The second method is called *re-keying*.

When a CA is re-keyed, new keypairs are stored in its certificate database, and these are the keys references for normal operations. However, for cloning a subsystem, the cloning process checks for the CA private key IDs as stored in its **CS.cfg** configuration file — and those key IDs are not updated when the certificate database keys change.

If a CA has been re-keyed and then an administrator attempts to clone it, the cloned CA fails to generate any certificates for the certificates which were re-keyed, and it shows up in the error logs with this error:

```
CertUtil::createSelfSignedCert() - CA private key is null!
```

To clone a CA that has been re-keyed:

1. Find all of the private key IDs in the **CS.cfg** file.

```
# grep privkey.id /var/lib/pki/pki-tomcat/ca/conf/CS.cfg
cloning.signing.privkey.id     =-
4d798441aa7230910d4e1c39fa132ea228d5d1bc
cloning.ocsp_signing.privkey.id =-
3e23e743e0ddd88f2a7c6f69fa9f9bcebef1a60
cloning.subsystem.privkey.id     =-
c3c1b3b4e8f5dd6d2bdefd07581c0b15529536
cloning.sslserver.privkey.id
=3023d30245804a4fab42be209ebb0dc683423a8f
cloning.audit_signing.privkey.id=2fe35d9d46b373efabe9ef01b8436667a70
df096
```

2. Print all of the current private key IDs stored in the NSS database and compare them to the private key IDs stored in the **CS.cfg** file:

```
# certutil -K -d alias
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      a7b0944b7b8397729a4c8c9af3a9c2b96f49c6f3
caSigningCert cert-ca4-test-master
< 1> rsa      6006094af3e5d02aaa91426594ca66cb53e73ac0
ocspSigningCert cert-ca4-test-master
< 2> rsa      d684da39bf4f2789a3fc9d42204596f4578ad2d9
subsystemCert cert-ca4-test-master
< 3> rsa      a8edd7c2b5c94f13144cacd99624578ae30b7e43
sslserverCert cert-ca4-test1
< 4> rsa      2fe35d9d46b373efabe9ef01b8436667a70df096
auditSigningCert cert-ca4-test1
```

   In this example, only the audit signing key is the same; the others have been changed.

3. Take the keys returned in step 2 and convert them from unsigned values (which is what **certutil** returns) to signed Java BigIntegers (which is how the keys are stored in the Certificate System database).

   This can be done with a calculator or by using the script in Example 10.1, “Certutil to BigInteger Conversion Program”.

4. Copy the new key values into the **CS.cfg** file.

```
# vim /var/lib/pki/pki-tomcat/ca/conf/CS.cfg

cloning.signing.privkey.id     =-
584f6bb4847c688d65b373650c563d4690b6390d
cloning.ocsp_signing.privkey.id
=6006094af3e5d02aaa91426594ca66cb53e73ac0
cloning.subsystem.privkey.id   =-
297b25c640b0d8765c0362bddfba690ba8752d27
cloning.sslserver.privkey.id   =-
5712283d4a36b0ecebb3532669dba8751cf481bd
cloning.audit_signing.privkey.id=2fe35d9d46b373efabe9ef01b8436667a70
df096
```

5. Clone the CA as described in Section 10.2, “Cloning a CA”.

Example 10.1. Certutil to BigInteger Conversion Program

This Java program can convert the key output from **certutil** to the required BigInteger format.

Save this as a **.java** file, such as **Test.java**.

```
import java.math.BigInteger;

public class Test
```

```
{

  public static byte[] hexStringToByteArray(String s) {
      int len = s.length();
      byte[] data = new byte[len / 2];
      for (int i = 0; i < len; i += 2) {
          data[i / 2] = (byte) ((Character.digit(s.charAt(i), 16) << 4)
                               + Character.digit(s.charAt(i+1), 16));
      }
      return data;
  }

  public static void main(String[] args)
  {
      byte[] bytes = hexStringToByteArray(args[0]);
      BigInteger big = new BigInteger (bytes);
      System.out.println("Result is  ==> " + big.toString(16));
  }
}
```

Then, compile the file:

```
# javac Test.java
```

CHAPTER 11. ADDITIONAL INSTALLATION OPTIONS

All Red Hat Certificate System instances created with **pkispawn** make certain assumptions about the instances being installed, such as the default signing algorithm to use for CA signing certificates and whether to allow IPv6 addresses for hosts.

This chapter describes additional configuration options that impact the installation and configuration for new instances, so many of these procedures occur before the instance is created.

11.1. REQUESTING SUBSYSTEM CERTIFICATES FROM AN EXTERNAL CA

Most of the time, it is easier and simpler to have a CA within your PKI be the root CA, since this affords a lot of flexibility for defining the rules and settings of the PKI deployment. However, for public-facing networks, this can be a difficult thing to implement, because web administrators have to find some way to propagate and update CA certificate chains to their clients so that any site is trusted. For this reason, people often use public CAs, hosted by companies like VeriSign or Thawte, to issue CA signing certificates and make all of their CAs subordinate to the public CA. This is one of the planning considerations covered in this guide.

All subsystem certificates can be submitted to an external CA when the subsystem is configured. When the certificates are generated from a CA outside the Certificate System deployment (or from a Certificate System CA in a different security domain), then the configuration process does not occur in one sitting. The configuration process is on hold until the certificates can be retrieved. Aside from that delay, the process is more or less the same as in Chapter 7, *Installing and Configuring Certificate System*.

To see an example of installation and configuration of a Certificate System instance using an External CA, run the **man pkispawn** command and read the *Installing an externally signed CA* part under the *EXAMPLES* section.

11.2. LIGHTWEIGHT SUB-CAS

Using the default settings, you are able to create lightweight sub-CAs. They enable you to configure services, like virtual private network (VPN) gateways, to accept only certificates issued by one sub-CA. At the same time, you can configure other services to accept only certificates issued by a different sub-CA or the root CA.

If you revoke the intermediate certificate of a sub-CA, all certificates issued by this sub-CA are automatically invalid.

If you set up the CA subsystem in Certificate System, it is automatically the root CA. All sub-CAs you create, are subordinated to this root CA.

11.2.1. Setting up a Lightweight Sub-CA

Depending on your environment, the installation of a sub-CA is different:

- If the parent CA is a Red Hat Certificate System instance and the sub-CA will join the parent's security domain, see http://www.dogtagpki.org/wiki/Installing_Subordinate_CA.
- If the parent CA is not a Red Hat Certificate System instance, or if the sub-CA will

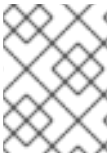
not join the parent's security domain, see http://www.dogtagpki.org/wiki/Installing_CA_with_Externaly-Signed_CA_Certificate.

11.2.2. Disabling the Creation of Lightweight Sub-CAs

In certain situations, administrators want to disable lightweight sub-CAs. To prevent adding, modifying, or removing sub-CAs, enter the following command on the Directory Server instance used by Certificate System:

```
# ldapmodify -D "cn=Directory Manager" -W -x -h server.example.com

dn: cn=aclResources,o=instance_name
changetype: modify
delete: resourceACLS
resourceACLS: certServer.ca.authorities:create,modify:allow
(create,modify) gr
 oup="Administrators":Administrators may create and modify lightweight
authori
 ties
-
delete: resourceACLS
resourceACLS: certServer.ca.authorities:delete:allow (delete)
group="Administr
 ators":Administrators may delete lightweight authorities
```

This command removes the default Access Control List (ACL) entries, which grant the permissions to manage sub-CAs.



NOTE

If any ACLs related to lightweight sub-CA creation have been modified or added, remove the relevant values.

11.2.3. Re-enabling the Creation of Lightweight Sub-CAs

If you previously disabled the creation of lightweight sub-CAs, you can re-enable the feature by entering the following command on the Directory Server instance used by Certificate System:

```
# ldapmodify -D "cn=Directory Manager" -W -x -h server.example.com

dn: cn=aclResources,o=instance_name
changetype: modify
add: resourceACLS
resourceACLS: certServer.ca.authorities:create,modify:allow
(create,modify) gr
 oup="Administrators":Administrators may create and modify lightweight
authori
 ties
resourceACLS: certServer.ca.authorities:delete:allow (delete)
group="Administr
 ators":Administrators may delete lightweight authorities
```

This command adds the Access Control List (ACL) entries, which grant the permissions to manage sub-CAs.

11.3. ENABLING IPV6 FOR A SUBSYSTEM

Certificate System automatically configures and manages connections between subsystems. Every subsystem must interact with a CA as members of a security domain and to perform their PKI operations.

For these connections, Certificate System subsystems can be recognized by their host's fully-qualified domain name or an IP address. By default, Certificate System resolves IPv4 addresses and host names automatically, but Certificate System can also use IPv6 for their connections. IPv6 is supported for all server connections: to other subsystems, to the administrative console (**pkiconsole**), or through command-line scripts such as**tpsclient**:

```
op=var_set name=ca_host value=IPv6 address
```

1. Install the Red Hat Certificate System packages.
2. Set the IPv4 and IPv6 addresses in the **/etc/hosts** file. For example:

```
 vim /etc/hosts

 192.0.0.0    server.example.com IPv4 address
 3ffe:1234:2222:2000:202:55ff:fe67:f527         server6.example.com
IPv6 address
```

3. Then, export the environment variable to use the IPv6 address for the server. For example:

```
export PKI_HOSTNAME=server6.example.com
```

4. Run **pkispawn** to create the new instance. The values for the server host name in the **CS.cfg** file will be set to the IPv6 address.

11.4. ENABLING LDAP-BASED ENROLLMENT PROFILES

To install with LDAP-based profiles set the **pki_profile_in_ldap=True** option in the **[CA]** section of the **pkispawn** configuration file.

NOTE

In this case, profile files will still appear in **/var/lib/pki/*instance_name*/ca/profiles/ca/**, but will be ignored.

To enable LDAP-based profiles on an existing instance, change the following in the instance's **CS.cfg**:

```
subsystem.1.class=com.netscape.cmscore.profile.LDAPProfileSubsystem
```

Then, import profiles manually into the database using either the **pki** command line utility or a custom script.

11.5. CUSTOMIZING TLS CIPHERS

It is possible to enforce TLS ciphers during the installation. See the *Configuring Ciphers* section in the *Red Hat Certificate System Administration Guide*.

CHAPTER 12. TROUBLESHOOTING INSTALLATION AND CLONING

This chapter covers some of the more common installation and migration issues that are encountered when installing Certificate System.

12.1. Installation

Q: I cannot see any Certificate System packages or updates.

A: Verify that your system is correctly registered to the Red Hat subscription management service, a valid subscription is assigned, and the Certificate System repository is enabled. For details, see Section 6.5, “Enabling the Certificate System Repository”.

Q: The init script returned an OK status, but my CA instance does not respond. Why?

A: This should not happen. Usually (but not always), this indicates a listener problem with the CA, but it can have many different causes. To see what errors have occurred, examine the **journal** log by running the following command:

```
journalctl -u pki-tomcatd@instance_name.service
```

Alternatively, examine the debug log files at **/var/log/pki/*instance_name*/*subsystem_type*/debug**.

One situation is when there is a PID for the CA, indicating the process is running, but that no listeners have been opened for the server. This would return Java invocation class errors in the **catalina.out** file:

```
Oct 29, 2010 4:15:44 PM org.apache.coyote.http11.Http11Protocol init
INFO: Initializing Coyote HTTP/1.1 on http-9080
java.lang.reflect.InvocationTargetException
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native
Method)
        at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.j
ava:64)
        at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccess
orImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:615)
        at
org.apache.catalina.startup.Bootstrap.load(Bootstrap.java:243)
        at
org.apache.catalina.startup.Bootstrap.main(Bootstrap.java:408)
Caused by: java.lang.UnsatisfiedLinkError: jss4
```

This could mean that you have the wrong version of JSS or NSS. The process requires **libnss3.so** in the path. Check this with this command:

```
ldd /usr/lib64/libjss4.so
```

If **libnss3.so** is not found, set the correct classpath in the **/etc/sysconfig/*instance_name*** configuration file. Then restart the CA using the **systemctl restart pki-tomcatd@*instance_name*.service** command.

Q: I want to customize the subject name for the CA signing certificate, but do not see a way to do this using the `pkispawn` interactive install mode.

A: To do this, a configuration file representing delta links to the **/etc/pki/default.cfg** file is required. See the pkispawn(8) and pki_default.cfg(5) man pages.

Q: I want to set different certificate validity periods and extensions for my root certificate authority — but I do not see a way to set it using `pkispawn`.

A: You cannot currently do this using **pkispawn**. However, there *is* a way to edit the certificate profiles used by **pkispawn** to generate the root CA certificates.

IMPORTANT

You must do this *before* running **pkispawn** to create a new CA instance.

1. Back up the original CA certificate profile used by **pkispawn**.

```
cp -p /usr/share/pki/ca/conf/caCert.profile
/usr/share/pki/ca/conf/caCert.profile.orig
```

2. Open the CA certificate profile used by the configuration wizard.

```
vim /usr/share/pki/ca/conf/caCert.profile
```

3. Reset the validity period in the Validity Default to whatever you want. For example, to change the period to two years:

```
2.default.class=com.netscape.cms.profile.def.ValidityDefault
2.default.name=Validity Default
2.default.params.range=7200
```

4. Add any extensions by creating a new default entry in the profile and adding it to the list. For example, to add the Basic Constraint Extension, add the default (which, in this example, is default #9):

```
9.default.class=com.netscape.cms.profile.def.BasicConstraintsExt
Default
9.default.name=Basic Constraint Extension Constraint
9.default.params.basicConstraintsCritical=true
9.default.params.basicConstraintsIsCA=true
9.default.params.basicConstraintsPathLen=2
```

Then, add the default number to the list of defaults to use the new default:

```
list=2,4,5,6,7,8,9
```

5. Once the new profile is set up, then run **pkispawn** to create the new CA instance and go through the configuration wizard.

Q: I am seeing an HTTP 500 error code when I try to connect to the web services pages after configuring my subsystem instance.

A: This is an unexpected generic error which can have many different causes. Check in the **journal**, **system**, and **debug** log files for the instance to see what errors have occurred. This lists a couple of common errors, but there are many other possibilities.

Error #1: The LDAP database is not running.

If the Red Hat Directory Server instance use for the internal database is not running, then you cannot connect to the instance. This will be apparent in exceptions in the **journal** file that the instance is not ready:

```
java.io.IOException: CS server is not ready to serve.

com.netscape.cms.servlet.base.CMSServlet.service(CMSServlet.java:409)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:688)
```

The Tomcat logs will specifically identify the problem with the LDAP connection:

```
5558.main - [29/Oct/2010:11:13:40 PDT] [8] [3] In Ldap (bound)
connection pool
to host ca1 port 389, Cannot connect to LDAP server. Error:
netscape.ldap.LDAPException: failed to connect to server
ldap://ca1.example.com:389 (91)
```

As will the instance's **debug** log:

```
[29/Oct/2010:11:39:10][main]: CMS:Caught EBaseException
Internal Database Error encountered: Could not connect to LDAP server
host
ca1 port 389 Error netscape.ldap.LDAPException: failed to connect to
server ldap://ca1:389 (91)
        at
com.netscape.cmscore.dbs.DBSubsystem.init(DBSubsystem.java:262)
```

Error #2: A VPN is blocking access.

Another possibility is that you are connecting to the subsystem over a VPN. The VPN **must** have a configuration option like **Use this connection only for resources on its network** enabled. If that option is not enabled, then the **journal** log file for the instance's Tomcat service shows a series of connection errors that result in the HTTP 500 error:

```
May 26, 2010 7:09:48 PM org.apache.catalina.core.StandardWrapperValve
invoke
SEVERE: Servlet.service() for servlet services threw exception
java.io.IOException: CS server is not ready to serve.
        at
com.netscape.cms.servlet.base.CMSServlet.service(CMSServlet.java:441)
        at
javax.servlet.http.HttpServlet.service(HttpServlet.java:803)
        at
```

```
org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(Appli
cationFilterChain.java:269)
        at
org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFi
lterChain.java:188)
        at
org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperVa
lve.java:210)
        at
org.apache.catalina.core.StandardContextValve.invoke(StandardContextVa
lve.java:172)
        at
org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.ja
va:127)
        at
org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.ja
va:117)
        at
org.apache.catalina.valves.AccessLogValve.invoke(AccessLogValve.java:5
42)
        at
org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValv
e.java:108)
        at
org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java
:151)
        at
org.apache.coyote.http11.Http11Processor.process(Http11Processor.java:
870)
        at
org.apache.coyote.http11.Http11BaseProtocol$Http11ConnectionHandler.pr
ocessConnection(Http11BaseProtocol.java:665)
        at
org.apache.tomcat.util.net.PoolTcpEndpoint.processSocket(PoolTcpEndpoi
nt.java:528)
        at
org.apache.tomcat.util.net.LeaderFollowerWorkerThread.runIt(LeaderFoll
owerWorkerThread.java:81)
        at
org.apache.tomcat.util.threads.ThreadPool$ControlRunnable.run(ThreadPo
ol.java:685)
 at java.lang.Thread.run(Thread.java:636)
```

12.2. Java Console

Q: I cannot open the `pkiconsole` and I am seeing Java exceptions in stdout.

A: This probably means that you have the wrong JRE installed or the wrong JRE set as the default. Run **`alternatives --config java`** to see what JRE is selected. Red Hat Certificate System requires OpenJDK 1.7.

Q: I tried to run `pkiconsole`, and I got Socket exceptions in stdout. Why?

A: This means that there is a port problem. Either there are incorrect SSL/TLS settings for the administrative port (meaning there is bad configuration in the **server.xml**) or the wrong port was given to access the admin interface.

Port errors will look like the following:

```
NSS Cipher Supported '0xff04'
java.io.IOException: SocketException cannot read on socket
        at org.mozilla.jss.ssl.SSLSocket.read(SSLSocket.java:1006)
        at
org.mozilla.jss.ssl.SSLInputStream.read(SSLInputStream.java:70)
        at
com.netscape.admin.certsrv.misc.HttpInputStream.fill(HttpInputStream.j
ava:303)
        at
com.netscape.admin.certsrv.misc.HttpInputStream.readLine(HttpInputStre
am.java:224)
        at
com.netscape.admin.certsrv.connection.JSSConnection.readHeader(JSSConn
ection.java:439)
        at
com.netscape.admin.certsrv.connection.JSSConnection.initReadResponse(J
SSConnection.java:430)
        at
com.netscape.admin.certsrv.connection.JSSConnection.sendRequest(JSSCon
nection.java:344)
        at
com.netscape.admin.certsrv.connection.AdminConnection.processRequest(A
dminConnection.java:714)
        at
com.netscape.admin.certsrv.connection.AdminConnection.sendRequest(Admi
nConnection.java:623)
        at
com.netscape.admin.certsrv.connection.AdminConnection.sendRequest(Admi
nConnection.java:590)
        at
com.netscape.admin.certsrv.connection.AdminConnection.authType(AdminCo
nnection.java:323)
        at
com.netscape.admin.certsrv.CMSServerInfo.getAuthType(CMSServerInfo.jav
a:113)
        at com.netscape.admin.certsrv.CMSAdmin.run(CMSAdmin.java:499)
        at com.netscape.admin.certsrv.CMSAdmin.run(CMSAdmin.java:548)
        at com.netscape.admin.certsrv.Console.main(Console.java:1655)
```

Q: I attempt to start the console, and the system prompts me for my user name and password. After I enter these credentials, the console fails to appear.

A: Make sure the user name and password you entered are valid. If so, enable the debug output and examine it.

To enable the debug output, open the **/usr/bin/pkiconsole** file, and add the following lines:

```
==========================================
```

```
${JAVA} ${JAVA_OPTIONS} -cp ${CP} -
Djava.util.prefs.systemRoot=/tmp/.java -
Djava.util.prefs.userRoot=/tmp/java com.netscape.admin.certsrv.Console
-s instanceID -D 9:all -a $1
----------
note: "-D 9:all" is for verbose output on the console.
============================================
```

PART III. UPGRADING CERTIFICATE SYSTEM FROM 9.X TO THE LATEST VERSION

To upgrade Red Hat Certificate System from 9.0 to the latest version:

1. Upgrade the packages and configuration files. See Upgrading the Packages and Configuration Files.
2. Upgrade the databases. See:
   - Upgrading the Database from 9.0 to 9.1
   - Upgrading the Database from 9.1 to 9.2
   - Upgrading the Database from 9.2 to 9.3
3. Restart the Certificate System instance:

```
# systemctl restart pki-tomcatd@instance_name.service
```

NOTE

To migrate from a lower major version, such as 8.2, to Certificate System 9, see Chapter 18, *Migrating From Certificate System 8 to 9*.

CHAPTER 13. UPGRADING THE PACKAGES AND CONFIGURATION FILES

Before you upgrade the subsystem databases, install the latest updates:

```
# yum update
```

This command updates all packages on the server. If Directory Server is installed on a different host, additionally update all packages on that host as well.

During the update, Certificate System configuration files, such as **/etc/pki/*instance_name*/*subsystem*/CS.cfg** and **/etc/pki/*instance_name*/server.xml**, are automatically modified to the new version.

Log files generated during the Certificate System upgrade are:

- **/var/log/pki/pki-server-upgrade-*version*.log**
- **/var/log/pki/pki-upgrade-*version*.log**

Note that the version number in the log file's name represent the pki* package version and not the Certificate System version.

For details about updating Directory Server, see:

- the corresponding chapter in the *Red Hat Directory Server Installation Guide*.
- Red Hat Directory Server Release Notes for notable changes, bug fixes, and known issues in Directory Server.

CHAPTER 14. UPGRADING THE DATABASE FROM 9.0 TO 9.1

After you upgraded the packages and configuration files, you must manually upgrade the database schema and subsystem databases for every Certificate System instance.

14.1. UPGRADING THE DATABASE SCHEMA

To upgrade the Certificate System database schema in Directory Server:

```
# ldapmodify -D "cn=Directory Manager" -W -h server.example.com -p 389 -x
dn: cn=schema
changetype: modify
add: attributeTypes
attributeTypes: ( realm-oid NAME 'realm' DESC 'CMS defined attribute'
 SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 X-ORIGIN 'user defined' )

dn: cn=schema
changetype: modify
delete: objectClasses
objectClasses: ( request-oid NAME 'request' DESC 'CMS defined class'
 SUP top STRUCTURAL MUST cn MAY ( requestId $ dateOfCreate $ dateOfModify
 $ requestState $ requestResult $ requestOwner $ requestAgentGroup
 $ requestSourceId $ requestType $ requestFlag $ requestError
 $ userMessages $ adminMessages ) X-ORIGIN 'user defined' )

add: objectClasses
objectClasses: ( request-oid NAME 'request' DESC 'CMS defined class'
 SUP top STRUCTURAL MUST cn MAY ( requestId $ dateOfCreate $ dateOfModify
 $ requestState $ requestResult $ requestOwner $ requestAgentGroup
 $ requestSourceId $ requestType $ requestFlag $ requestError
 $ userMessages $ adminMessages $ realm ) X-ORIGIN 'user defined' )

dn: cn=schema
changetype: modify
add: attributeTypes
attributeTypes: ( authorityID-oid NAME 'authorityID' DESC 'Authority ID'
 SYNTAX 1.3.6.1.4.1.1466.115.121.1.40 SINGLE-VALUE X-ORIGIN
 'user defined' )
attributeTypes: ( authorityKeyNickname-oid NAME 'authorityKeyNickname'
 DESC 'Authority key nickname' SYNTAX 1.3.6.1.4.1.1466.115.121.1.44
 SINGLE-VALUE X-ORIGIN 'user-defined' )
attributeTypes: ( authorityParentID-oid NAME 'authorityParentID' DESC
 'Authority Parent ID' SYNTAX 1.3.6.1.4.1.1466.115.121.1.40 SINGLE-VALUE
 X-ORIGIN 'user defined' )
attributeTypes: ( authorityEnabled-oid NAME 'authorityEnabled' DESC
 'Authority Enabled' SYNTAX 1.3.6.1.4.1.1466.115.121.1.7 SINGLE-VALUE
 X-ORIGIN 'user defined' )
attributeTypes: ( authorityDN-oid NAME 'authorityDN' DESC 'Authority DN'
 SYNTAX 1.3.6.1.4.1.1466.115.121.1.12 SINGLE-VALUE X-ORIGIN
 'user defined' )
attributeTypes: ( authoritySerial-oid NAME 'authoritySerial' DESC
 'Authority certificate serial number' SYNTAX
 1.3.6.1.4.1.1466.115.121.1.27 SINGLE-VALUE X-ORIGIN 'user defined' )
attributeTypes: ( authorityParentDN-oid
```

```
 NAME 'authorityParentDN' DESC 'Authority Parent DN' SYNTAX
 1.3.6.1.4.1.1466.115.121.1.12 SINGLE-VALUE X-ORIGIN 'user defined' )
attributeTypes: ( authorityKeyHost-oid NAME 'authorityKeyHost' DESC
 'Authority Key Hosts' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 X-ORIGIN
 'user defined' )

dn: cn=schema
changetype: modify
add: objectClasses
objectClasses: ( authority-oid NAME 'authority' DESC
 'Certificate Authority' SUP top STRUCTURAL MUST ( cn $ authorityID
 $ authorityKeyNickname $ authorityEnabled $ authorityDN ) MAY
 ( authoritySerial $ authorityParentID $ authorityParentDN
 $ authorityKeyHost $ description ) X-ORIGIN 'user defined' )
```

14.2. UPGRADING THE CA DATABASE

To upgrade the certificate authority (CA) database:

1. Upgrade the container entries:

```
# ldapmodify -D "cn=Directory Manager" -W -h server.example.com -p
389 -x
dn: ou=authorities,ou=ca,CA_base_DN
changetype: add
objectClass: top
objectClass: organizationalUnit
ou: authorities
```

2. Upgrade the access control list (ACL) entries:

```
# ldapmodify -D "cn=Directory Manager" -W -h server.example.com -p
389 -x
dn: cn=aclResources,CA_base_DN
changetype: modify
add: resourceACLS
resourceACLS: certServer.ca.authorities:list,read:allow (list,read)
 user="anybody":Anybody may list and read lightweight authorities
resourceACLS: certServer.ca.authorities:create,modify:allow
 (create,modify) group="Administrators":Administrators may create
 and modify lightweight authorities resourceACLS:
 certServer.ca.authorities:delete:allow (delete)
 group="Administrators":Administrators may delete lightweight
 authorities
```

3. Upgrade the database indexes:

```
# ldapmodify -D "cn=Directory Manager" -W -h server.example.com -p
389 -x
dn: cn=issuername,cn=index,cn=CA_database_name,cn=ldbm database,
 cn=plugins, cn=config
changetype: add
objectClass: top
objectClass: nsIndex
```

```
nsindexType: eq
nsindexType: pres
nsindexType: sub
nsSystemindex: false
cn: issuername
```

4. Add the ***realm*** attribute:

```
# ldapmodify -D "cn=Directory Manager" -W -h server.example.com -p
389 -x
dn: cn=schema
changetype: modify
add: attributeTypes
attributeTypes: ( realm-oid NAME 'realm' DESC 'CMS defined
attribute'
 SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 X-ORIGIN 'user defined' )

delete: objectClasses
objectClasses: ( request-oid NAME 'request' DESC 'CMS defined class'
 SUP top STRUCTURAL MUST cn MAY ( requestId $ dateOfCreate $
 dateOfModify $ requestState $ requestResult $ requestOwner $
 requestAgentGroup $ requestSourceId $ requestType $ requestFlag $
 requestError $ userMessages $ adminMessages ) X-ORIGIN 'user
defined' )

add: objectClasses
objectClasses: ( request-oid NAME 'request' DESC 'CMS defined class'
 SUP top STRUCTURAL MUST cn MAY ( requestId $ dateOfCreate $
 dateOfModify $ requestState $ requestResult $ requestOwner $
 requestAgentGroup $ requestSourceId $ requestType $ requestFlag $
 requestError $ userMessages $ adminMessages $ realm ) X-ORIGIN
'user
 defined' )
```

5. Remove the certificate validity delay:

   a. In the **/var/lib/pki/*instance_name*/ca/profiles/ca/caDualCert.cfg** file, set:

   ```
   policyset.signingCertSet.2.default.params.startTime=0
   ```

   b. In the **/var/lib/pki/*instance_name*/ca/profiles/ca/caECDualCert.cfg** file, set:

   ```
   policyset.signingCertSet.2.default.params.startTime=0
   ```

   c. In the **/var/lib/pki/*instance_name*/ca/profiles/ca/caDualCert.cfg** file, set:

   ```
   policyset.signingCertSet.2.default.params.startTime=0
   ```

   d. In the **/var/lib/pki/*instance_name*/ca/profiles/ca/caJarSigningCert.cfg** file, set:

   ```
   
   ```

```
policyset.caJarSigningSet.2.default.params.startTime=0
```

e. In the **/var/lib/pki/*instance_name*/ca/profiles/ca/caSignedLogCert.cfg** file, set:

```
policyset.caLogSigningSet.2.default.params.startTime=0
```

6. Add the ***issuerName*** attribute to certificate records:

```
# pki-server db-upgrade
```

7. Update the attribute syntax to allow underscores in instance names:

```
# ldapmodify -D "cn=Directory Manager" -W -h server.example.com -p
389 -x
dn: cn=schema
changetype: modify
delete: objectClasses
objectClasses: ( authority-oid NAME 'authority' DESC 'Certificate
 Authority' SUP top STRUCTURAL MUST ( cn $ authorityID
 $ authorityKeyNickname $ authorityEnabled $ authorityDN ) MAY
 ( authoritySerial $ authorityParentID $ authorityParentDN $
 authorityKeyHost $ description ) X-ORIGIN 'user defined' )

delete: attributeTypes
attributeTypes: ( authorityKeyNickname-oid NAME
 'authorityKeyNickname' DESC 'Authority key nickname' SYNTAX
 1.3.6.1.4.1.1466.115.121.1.44 SINGLE-VALUE X-ORIGIN
 'user-defined' )

add: attributeTypes
attributeTypes: ( authorityKeyNickname-oid NAME
 'authorityKeyNickname' DESC 'Authority key nickname'
 SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 SINGLE-VALUE
 X-ORIGIN 'user-defined' )

add: objectClasses
objectClasses: ( authority-oid NAME 'authority' DESC
 'Certificate Authority' SUP top STRUCTURAL MUST ( cn
 $ authorityID $ authorityKeyNickname $ authorityEnabled
 $ authorityDN ) MAY ( authoritySerial $ authorityParentID
 $ authorityParentDN $ authorityKeyHost $ description )
 X-ORIGIN 'user defined' )
```

14.3. UPGRADING THE KRA DATABASE

To update the key recovery authority (KRA) database:

1. Upgrade the database indexes:

```
# ldapmodify -D "cn=Directory Manager" -W -h server.example.com -p
389 -x
dn: cn=realm,cn=index,cn=KRA_database_name,cn=ldbm database,
```

```
 cn=plugins,cn=config
changetype: add
objectClass: top
objectClass: nsIndex
nsindexType: eq
nsindexType: pres
nsSystemindex: false
cn: realm
```

2. Add the ***realm*** attribute:

```
# ldapmodify -D "cn=Directory Manager" -W -h server.example.com -p
389 -x
dn: cn=schema
changetype: modify
add: attributeTypes
attributeTypes: ( realm-oid NAME 'realm' DESC 'CMS defined
attribute'
 SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 X-ORIGIN 'user defined' )

delete: objectClasses
objectClasses: ( request-oid NAME 'request' DESC 'CMS defined class'
 SUP top STRUCTURAL MUST cn MAY ( requestId $ dateOfCreate $
 dateOfModify $ requestState $ requestResult $ requestOwner $
 requestAgentGroup $ requestSourceId $ requestType $ requestFlag $
 requestError $ userMessages $ adminMessages ) X-ORIGIN 'user
defined' )

add: objectClasses
 objectClasses: ( request-oid NAME 'request' DESC 'CMS defined
class'
 SUP top STRUCTURAL MUST cn MAY ( requestId $ dateOfCreate $
 dateOfModify $ requestState $ requestResult $ requestOwner $
 requestAgentGroup $ requestSourceId $ requestType $ requestFlag $
 requestError $ userMessages $ adminMessages $ realm ) X-ORIGIN
'user
 defined' )

delete: objectClasses
objectClasses: ( keyRecord-oid NAME 'keyRecord' DESC 'CMS defined
 class' SUP top STRUCTURAL MUST cn MAY ( serialno $ dateOfCreate $
 dateOfModify $ keyState $ privateKeyData $ ownerName $ keySize $
 metaInfo $ dateOfArchival $ dateOfRecovery $ algorithm $
 publicKeyFormat $ publicKeyData $ archivedBy $ clientId $ dataType
$
 status ) X-ORIGIN 'user defined' )

add: objectClasses
objectClasses: ( keyRecord-oid NAME 'keyRecord' DESC 'CMS defined
 class' SUP top STRUCTURAL MUST cn MAY ( serialno $ dateOfCreate $
 dateOfModify $ keyState $ privateKeyData $ ownerName $ keySize $
 metaInfo $ dateOfArchival $ dateOfRecovery $ algorithm $
 publicKeyFormat $ publicKeyData $ archivedBy $ clientId $ dataType
$
 status $ realm ) X-ORIGIN 'user defined' )
```

3. Update and re-index the virtual list views (VLV):

   a. Delete the existing indexes:

   ```
   # pki-server kra-db-vlv-del -i CS_instance_name -D DS_bind_DN \
        -w DS_bind_password
   ```

   b. Add the new indexes:

   ```
   # pki-server kra-db-vlv-add -i CS_instance_name -D DS_bind_DN \
        -w DS_bind_password
   ```

   c. Restart the Directory Server instance:

   ```
   # systemctl restart dirsrv@DS_instance_name
   ```

   d. Re-index the database:

   ```
   # pki-server kra-db-vlv-reindex -i CS_instance_name -D DS_bind_DN
   \
        -w DS_bind_password
   ```

14.4. UPGRADING THE TPS DATABASE

The token processing system (TPS) was a technology preview in Certificate System 9.0. Therefore, upgrading the TPS from this version is not supported.

CHAPTER 15. UPGRADING THE DATABASE FROM 9.1 TO 9.2

No database updates required.

CHAPTER 16. UPGRADING THE DATABASE FROM 9.2 TO 9.3

No database updates required.

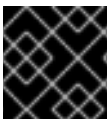
CHAPTER 17. UPGRADING THE DATABASE FROM 9.3 TO 9.4

No database updates required.

PART IV. MIGRATING TO CERTIFICATE SYSTEM 9

CHAPTER 18. MIGRATING FROM CERTIFICATE SYSTEM 8 TO 9

Red Hat Certificate System does not support in-place upgrades from an older major version, such as from 8.2 to 9. For major version upgrades, you must migrate the old instance. Additionally, you can use the same procedure to create a copy of your production environment for testing and debugging purposes.



IMPORTANT

Before migrating, read the *Red Hat Certificate System Release Notes*.

A Certificate System migration requires the following steps:

1. Section 18.1, “Exporting Data from the Previous System”
2. Section 18.2, “Setting up the CA on the New Host”
3. Section 18.3, “Importing the Data into the New CA”
4. Section 18.4, “Reassigning Users to Default Groups”

18.1. EXPORTING DATA FROM THE PREVIOUS SYSTEM

Before you set up the new Certificate System instance, export the data of the current certificate authority (CA).

On the host that runs the Certificate System instance:

1. Create a directory for the files to export. For example:

```
# mkdir -m 770 /tmp/cs_bak/
```

2. Export the signing certificate and key:
   - When using a hardware security module (HSM):
     1. List the CA signing certificate nickname. For example:

```
# certutil -L -d /var/lib/instance_name/alias/ | grep
"caSigningCert"
caSigningCert ca-pki-ca          CTu,Cu,Cu
```

     2. Export the CA certificate:

```
# certutil -L -d /var/lib/instance_name/alias/ \
     -n "caSigningCert ca-pki-ca" \
     -a > /tmp/cs_bak/ca_signing.crt
```

        The key is stored in the HSM and must be available to the new instance.
   - When not using an HSM:

1. In the configuration file, find the password that protects the CA Network Security Service (NSS) database, and write it to a file:

   - If the password is stored in the **/var/lib/*instance_name*/conf/password.conf** file:

     ```
     # grep "internal="
     /var/lib/instance_name/conf/password.conf | \
          awk -F= '{print $2;}' > /tmp/cs_bak/nss_password.txt
     ```

   - If the password is stored in the **/etc/*instance_name*/password.conf** file:

     ```
     # grep "internal=" /etc/instance_name/password.conf | \
          awk -F= '{print $2;}' > /tmp/cs_bak/nss_password.txt
     ```

2. Create a file with a password that will be used in the next step. For example:

   ```
   # echo Secret123 > /tmp/cs_bak/pkcs12_password.txt
   ```

3. Export the signing certificate and key:

   ```
   # PKCS12Export -d /var/lib/instance_name/alias/ \
        -p /tmp/cs_bak/nss_password.txt \
        -w /tmp/cs_bak/pkcs12_password.txt \
        -o /tmp/cs_bak/ca.p12
   ```

3. Export the certificate signing request (CSR):

   - If the CSR is stored in the **/etc/*instance_name*/CS.cfg** file:

     ```
     # echo "-----BEGIN NEW CERTIFICATE REQUEST-----" >
     /tmp/cs_bak/ca_signing.csr

     # sed -n "/^ca.signing.certreq=/ s/^[^=]*=// p"
     /etc/instance_name/ca/CS.cfg \
          >> /tmp/cs_bak/ca_signing.csr

     # echo "-----END NEW CERTIFICATE REQUEST-----" >>
     /tmp/cs_bak/ca_signing.csr
     ```

   - If the CSR is stored in the **/var/lib/*instance_name*/conf/CS.cfg** file:

     ```
     # echo "-----BEGIN NEW CERTIFICATE REQUEST-----" >
     /tmp/cs_bak/ca_signing.csr

     # sed -n "/^ca.signing.certreq=/ s/^[^=]*=// p"
     /var/lib/instance_name/conf/CS.cfg \
          >> /tmp/cs_bak/ca_signing.csr

     # echo "-----END NEW CERTIFICATE REQUEST-----" >>
     /tmp/cs_bak/ca_signing.csr
     ```

4. If the CA is an intermediate CA, extract the root CA or certificate chain from the NSS database:

```
# certutil -L -d /var/lib/instance_name/alias/ -n "root_CA_nickname"
\
     -a > /tmp/cs_bak/ca_rootca_signing.crt
```

5. Copy the directory that contains the exported files to the new server. For example:

```
# scp -r /tmp/cs_bak/ new_server:/tmp/
```

On the host that runs the Directory Server instance:

1. Create a directory for the files to export, and grant write permissions to the Directory Server user. For example:

```
# mkdir -m 770 /tmp/ds_bak/
# chown root:dirsrv /tmp/ds_bak/
```

2. Export the Directory Server database:

```
# db2ldif -Z DS_instance_name -n "CS_database_name" -a
/tmp/ds_bak/old_ca.ldif
```

   In the example:

   - ***DS_instance_name*** sets the Directory Server instance name. For example: ***slapd-host_name***.
   - ***CS_database_name*** sets the Certificate System CA database name used in Directory Server. For example: ***host_name-CS_instance_name***.

NOTE

The **db2ldif** command runs under the Directory Server user. Therefore the destination directory must be writable by this user.

3. Copy the directory that contains the exported files to the new server. For example:

```
# scp -r /tmp/ds_bak/ new_server:/tmp/
```

18.2. SETTING UP THE CA ON THE NEW HOST

After you have exported the data from the existing Directory Server and Certificate System instances in Section 18.1, “Exporting Data from the Previous System”, set up the certificate authority (CA) on the new host:

1. Set up Directory Server. See Section 6.4, “Installing Red Hat Directory Server”.
2. Enable the Certificate System repository. See Section 6.5, “Enabling the Certificate System Repository”.

3. Install the pki-ca package:

```
# yum install pki-ca
```

   If you require additional features, such as the Certificate System console, install the corresponding packages. For details, see Section 7.2, “Certificate System Packages”.

4. When setting up the CA on a host that uses an IPv6 address, apply the steps described in Section 11.3, “Enabling IPv6 for a Subsystem”.

5. Depending on your environment, this step differs.

   - When using a hardware security module (HSM):

     Create a deployment configuration file, for example **/root/pki-CA-deployment.txt**, with the following content:

```
[DEFAULT]
pki_instance_name=instance_name
pki_admin_password=caadmin_password
pki_client_pkcs12_password=pkcs12_file_password
pki_ds_password=DS_password
pki_hsm_enable=True
pki_hsm_libfile=path_to_HSM_library
pki_hsm_modulename=HSM_module_name
pki_token_name=HSM_token_name
pki_token_password=HSM_token_password
pki_ds_ldap_port=389
pki_existing=True

[CA]
pki_ca_signing_csr_path=/tmp/cs_bak/ca_signing.csr
pki_ca_signing_cert_path=/tmp/cs_bak/ca_signing.crt
pki_ca_signing_nickname=caSigningCert ca-pki-ca
pki_ca_signing_token=HSM_token_name
pki_ds_base_dn=o=pki-tomcat-CA
pki_ds_database=instance_name-CA
pki_serial_number_range_start=4e
pki_request_number_range_start=30
pki_master_crl_enable=False
pki_cert_chain_path=/tmp/cs_bak/rootca_signing.crt
pki_cert_chain_nickname=caSigningCert cert-top-rootca
pki_ca_signing_record_create=False
pki_ca_signing_serial_number=decimal_CA_signing_certificate_seria
l
```

     For descriptions of the parameters used in the previous example, see Table 18.1, “pkispawn Parameter Descriptions” at the end of this step.

   - When not using a hardware security module (HSM):

     1. Verify that the PKCS #12 file contains the CA signing certificate and key. For example:

```
# pki pkcs12-cert-find --pkcs12-file /tmp/cs_bak/ca.p12 \
     --pkcs12-password-file /tmp/cs_bak/pkcs12_password.txt
```

```
---------------
1 entries found
---------------
  Certificate ID: 308b4c7d4b5efc4052aec26e49a2c5e2e14c9e90
  Serial Number: 0x1
  Nickname: caSigningCert ca-pki-ca
  Subject DN: CN=CA Signing Certificate,O=EXAMPLE
  Issuer DN: CN=CA Signing Certificate,O=EXAMPLE
  Trust Flags: CTu,Cu,Cu
  Has Key: true

# pki pkcs12-key-find --pkcs12-file /tmp/cs_bak/ca.p12 \
     --pkcs12-password-file /tmp/cs_bak/pkcs12_password.txt
---------------
1 entries found
---------------
  Key ID: 308b4c7d4b5efc4052aec26e49a2c5e2e14c9e90
  Subject DN: CN=CA Signing Certificate,O=EXAMPLE
  Algorithm: RSA
```

Note that the file can additionally contain other certificates and keys.

2. Verify the trust flags of the CA signing certificate in the output of the previous step. Reset the flags if they are not set to **CTu,Cu,Cu** or if they are missing:

```
# pki pkcs12-cert-mod caSigningCert cert-pki-tomcat CA \
     --pkcs12-file /tmp/cs_bak/ca.p12 \
     --pkcs12-password-file /tmp/cs_bak/pkcs12_password.txt \
     --trust-flags "CTu,Cu,Cu"
```

3. Remove all other certificates and keys, except the CA signing certificate and key, from the PKCS #12 file. For example:

```
# pki pkcs12-cert-del Server-Cert root_CA_nickname \
     --pkcs12-file /tmp/cs_bak/ca.p12 \
     --pkcs12-password-file /tmp/cs_bak/pkcs12_password.txt

# pki pkcs12-cert-del "subsystemCert ca-pki-ca" \
     --pkcs12-file /tmp/cs_bak/ca.p12 \
     --pkcs12-password-file /tmp/cs_bak/pkcs12_password.txt

# pki pkcs12-cert-del "ocspSigningCert ca-pki-ca" \
     --pkcs12-file /tmp/cs_bak/ca.p12 \
     --pkcs12-password-file /tmp/cs_bak/pkcs12_password.txt

# pki pkcs12-cert-del "auditSigningCert ca-pki-ca" \
     --pkcs12-file /tmp/cs_bak/ca.p12 \
     --pkcs12-password-file /tmp/cs_bak/pkcs12_password.txt
```

4. If the CA being migrated is an intermediate CA, remove the root CA certificate from the PKCS #12 file. For example:

```
# pki pkcs12-cert-del ca-pki-ca \
     --pkcs12-file /tmp/cs_bak/ca.p12 \
     --pkcs12-password-file /tmp/cs_bak/pkcs12_password.txt
```

5. Create a deployment configuration file, for example **/root/pki-CA-deployment.txt**, with the following content:

```
[DEFAULT]
pki_instance_name=instance_name
pki_admin_password=caadmin_password
pki_client_pkcs12_password=pkcs12_file_password
pki_ds_password=DS_password
pki_ds_ldap_port=389
pki_existing=True

[CA]
pki_ca_signing_nickname=caSigningCert ca-pki-ca
pki_ca_signing_csr_path=/tmp/cs_bak/ca_signing.csr
pki_pkcs12_path=/tmp/cs_bak//cs_bak/ca.p12
pki_pkcs12_password=pkcs12_file_password
pki_ds_base_dn=o=pki-tomcat-CA
pki_ds_database=pki-tomcat-CA
pki_serial_number_range_start=43
pki_request_number_range_start=30
pki_master_crl_enable=False
pki_cert_chain_path=/tmp/cs_bak/rootca_signing.crt
pki_cert_chain_nickname=caSigningCert cert-top-rootca
pki_ca_signing_record_create=False
pki_ca_signing_serial_number=decimal_CA_signing_certificate_serial
```

For descriptions of the parameters used in the previous example, see Table 18.1, “pkispawn Parameter Descriptions”.

Table 18.1. pkispawn Parameter Descriptions

Parameters and Settings	Description
pki_hsm_* and ***pki_token_****	Enables communication with the HSM. Only set these parameters when setting up a CA with HSM.
pki_existing=True	Sets to use the existing CA mechanism.
pki_ca_signing_nickname	The CA signing nickname must be exactly the same as used in the previous installation, otherwise the installer cannot find the signing key.
pki_ca_signing_*	Sets the paths to the certificate signing request (CSR) and the certificate files copied from the existing machine.
pki_pkcs12_*	Sets the path to the PKCS #12 file and the password used to decrypt the file. Do not set this parameter when deploying a CA with HSM.

Parameters and Settings	Description
pki_ds_base_dn	Sets the Directory Server base distinguished name (DN). The value must be the same as on the previous CA. You can find this value on the previous host in the ***internaldb.basedn*** parameter in the **/var/lib/*instance_name*/conf/CS.cfg** file.
pki_serial_number_range_start	The serial number is critical. The value must be higher than the last number used in the previous CA. To display which numbers are already used, see the old CA's agent interface. This parameter is set in hex format without the leading **0x** prefix. The value used in the examples (**4e**) is **78** in decimal.
pki_request_number_range_start	The request number is critical. The value must be higher than the last number used in the previous CA. To display which numbers are already used, see the old CA's agent interface. The value is set in decimal format.
pki_master_crl_enable=False	Prevents the initial creation and publishing of a certificate revocation list (CRL) during the setup. Instead, the CRL will be imported from the old data during the database migration.
pki_cert_chain_path and ***pki_cert_chain_nickname***	Set these parameters only if the old CA is an intermediate CA. In this case, set the parameters to the path to the root CA certificate file and the nickname to use when storing the certificate in the network security services (NSS) database.
pki_ca_signing_record_create=False	Disables the recreation of the CA signing record at the end of the **pkispawn** process. This enables you to import the old database.
pki_ca_signing_serial_number	Sets the serial number of the CA signing certificate in decimal.

For further details and parameter descriptions, see the pkispawn(8) man page.

6. Create the new CA using the deployment configuration file. For example:

```
# pkispawn -s CA -f /root/pki-CA-deployment.txt
```

7. Verify that the CA signing key ID is the same in the existing and in the new CA. For example:

```
# grep "internal=" /var/lib/instance_name/conf/password.conf | \
     awk -F= '{print $2;}' > internal.txt
```

```
# certutil -K -d /var/lib/instance_name/alias/ -f internal.txt
...
< 2> rsa      7bd4dc662670ebe08a35086b054175559608ac20
caSigningCert ca-pki-ca
...
```

18.3. IMPORTING THE DATA INTO THE NEW CA

After finishing setting up the new CA in Section 18.2, “Setting up the CA on the New Host”, you can import the data to the Directory Server database:

1. When migrating from a previous version, it can be necessary to manually clean up the LDAP data interchange format (LDIF) file. Before Red Hat Directory Server 10, syntax checking was disabled by default. Therefore, data from a previous version can include entries that are now invalid in Directory Server 10. For example:

   - Values of boolean attributes must be set either to **TRUE** or **FALSE** (all capitalized).

     IMPORTANT

     Do not automatically update all occurrences to uppercase by using a search and replace utility. Some attributes in the LDIF file contain these strings, but are not using the boolean type. Updating these attributes' values can cause the import to fail. Typically, boolean attributes are only used in the **cn=CAList,ou=Security Domain,*CS_instance_name*** security domain database entries.

   - Empty strings must be removed. The Directory Server syntax validation does not allow to set empty strings.

     Empty strings often appear in ***userType*** and ***userState*** attributes in **cmsUser** entries in **ou=People,*CS_instance_name***.

   During the import, other entries can fail, too. It is important to verify the log file after the database import. Optionally, you can import the LDIF file into a temporary, empty database to find out which entries caused the import to fail.

2. Shut down the CA service:

   ```
   # systemctl stop pki-tomcatd@instance_name.service
   ```

3. Optionally, back up the CA database on the new host:

   ```
   # db2bak
   ```

   The backup is stored in the **/var/lib/dirsrv/*instance_name*/bak/*host_name-time_stamp*/** directory.

4. Import the data into the new database. For example:

```
# ldapmodify -x -W -D 'cn=Directory Manager' -a -c -f
/tmp/ds_bak/old_ca.ldif | \
     tee /root/import.log
```

The **ldapmodify** utility only adds new entries and does not update existing entries, created when you installed the CA. For example:

- Top level entries. For example: **o=pki-tomcat-CA**.
- Default groups. For example: **cn=Certificate Manager Agents,ou=groups,o=pki-tomcat-CA**.

  Because the standard groups are not updated, the users are not automatically added to these groups. After the import, you must add members to each default group manually. See Section 18.4, “Reassigning Users to Default Groups”.
- Default access control lists (ACL) for the CA.

As mentioned earlier, Directory Server 10 uses syntax validation. Verify the output in the **/root/import.log** file and search for failed actions, such as**ldap_add: Invalid syntax (21)**. For further details, seeStep 1.

5. Remove the directory entry for the old security domain. For example:

```
# ldapmodify -W -x -D "cn=Directory Manager"
dn: cn=server.example.com:9445,cn=CAList,ou=Security Domain,o=pki-
tomcat-CA
changetype: delete
```

6. Enable the CA in the **/etc/pki/*instance_name*/ca/CS.cfg** file to act as the certificate revocation list (CRL) master:

```
ca.crl.MasterCRL.enable=true
```

7. Restart the CA service:

```
# systemctl start pki-tomcat@instance_name
```

18.4. REASSIGNING USERS TO DEFAULT GROUPS

As mentioned in Section 18.3, “Importing the Data into the New CA”, members of the default groups are not restored during the data import.

Add members to the default groups manually, using the Certificate System Console or the **pki** utility. For example:

1. Set up the client:

```
# pki -c password client-init
------------------
Client initialized
------------------
# pk12util -i ~/.dogtag/instance_name/ca_admin_cert.p12 -d
~/.dogtag/nssdb/
```

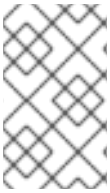
```
Enter Password or Pin for "NSS Certificate DB":
Enter password for PKCS12 file:
pk12util: PKCS12 IMPORT SUCCESSFUL
```

2. Add the **user** account to the **Certificate Manager Agents**, **Administrators**, and **Security Domain Administrators** groups:

```
# pki -n "PKI Administrator for example.com" -c password \
     user-membership-add user_name "Certificate Manager Agents"

# pki -n "PKI Administrator for example.com" -c password \
     user-membership-add user "Administrators"

# pki -n "PKI Administrator for example.com" -c password \
     user-membership-add user "Security Domain Administrators"
```

CHAPTER 19. MIGRATING AN OPENSSL CA TO CERTIFICATE SYSTEM

Red Hat Certificate System provides a mechanism to migrate an existing OpenSSL Certificate Authority (CA) to a new Certificate System CA that uses the existing CA signing key.



NOTE

The migration of Certificate System 8 to 9 is a special implementation of this procedure and is described in Chapter 18, *Migrating From Certificate System 8 to 9*.

Depending on your environment, see:

- Section 19.1, “Migrating an OpenSSL CA to Certificate System When Not Using an HSM”
- Section 19.2, “Migrating an OpenSSL CA to Certificate System When Using an HSM”

19.1. MIGRATING AN OPENSSL CA TO CERTIFICATE SYSTEM WHEN NOT USING AN HSM

1. Create a file with a password that will be used in the next step. For example:

   ```
   # echo password > ~/password.txt
   ```

2. Import the OpenSSL CA certificate and key into a PKCS #12 file using the **openssl pkcs12** command. Use the following options:

   - **-export** instructs the **openssl** command to export the data.
   - **-in *path_to_ca_certificate*** sets the path to the OpenSSL CA certificate.
   - **-inkey *path_to_CA_signing_key*** sets the path to the OpenSSL CA signing key.
   - **-out *path_to_PKCS_#12_file*** sets the path to the PKCS #12 file in which the output is stored.
   - **-name "*friendly_name*"** sets the friendly name of the certificate and key.
   - **-passout file:*path_to_password_file*** sets the path to the text file that contains the password used to encrypt the PKCS #12 file.

   For example, to export the OpenSSL CA certificate and key into the **~/ca.p12** file:

   ```
   # openssl pkcs12 -export -in ~/ca.crt -inkey ~/ca.key -out ~/ca.p12
   \
        -name "CA Certificate" -passout file:~/password.txt
   ```

3. Initialize a password protected Network Security Services (NSS) database for the Public Key Infrastructure (PKI) command-line interface. For example:

```
# pki -c password client-init
```

4. Set the **CTu,Cu,Cu** trust flags for the CA certificate with the **CA Certificate** nick name stored in the **~/ca.12** file using the password in the **~/password.txt** file:

```
# pki pkcs12-cert-mod --pkcs12-file ~/ca.p12 "CA Certificate" \
     --pkcs12-password-file ~/password.txt --trust-flags "CTu,Cu,Cu"
```

IMPORTANT

Enter the trust flags without spaces.

5. Display the CA certificate stored in the **~/ca.p12** file:

```
# pki pkcs12-cert-find --pkcs12-file ~/ca.p12 \
     --pkcs12-password-file ~/password.txt
---------------
1 entries found
--------------
  Certificate ID: 9311084d08b37d12e856b904b7e52eb3b1cece4a
  Serial Number: 0xe3f2b350edcd875c
  Nickname: CA Certificate
  Subject DN: O=Example,CN=CA Certificate
  Issuer DN: O=Example,CN=CA Certificate
  Trust Flags: CTu,Cu,Cu
  Has Key: true
```

6. Display the CA signing key stored in the **~/ca.p12** file:

```
# pki pkcs12-key-find --pkcs12-file ~/ca.p12 \
     --pkcs12-password-file ~/password.txt
---------------
1 entries found
---------------
  Key ID: 9311084d08b37d12e856b904b7e52eb3b1cece4a
  Subject DN: CA Certificate
  Algorithm: RSA
```

7. Copy the following files to the new Certificate System host:

   - OpenSSL CA Signing Certificate Request (CSR)
   - OpenSSL CA certificate chain (if available)
   - PKCS #12 file that contains the OpenSSL CA signing certificate and key
   - Password file used to protect the PKCS #12 file

   For example, to copy the files using secure copy:

```
# scp ~/ca.csr ~/certificate_chain.p7b ~/ca.p12 ~/password.txt
new_server:~/
```

8. Set up the CA on the new host. For details, see Section 18.2, “Setting up the CA on the New Host”.

After the migration, you can deactivate the OpenSSL CA or run it in read-only mode, where it only responds to Online Certificate Status Protocol (OCSP) requests.

19.2. MIGRATING AN OPENSSL CA TO CERTIFICATE SYSTEM WHEN USING AN HSM

When using an HSM and the keys cannot be extracted, make the HSM available to the Certificate System CA. For this, copy the following files to the new Certificate System host:

1. Copy the following files to the new Certificate System host:
   - CA signing certificate in PEM format
   - OpenSSL CA CSR
   - Certificate CA certificate chain (if available)
2. Set up the CA on the new host. For details, see Section 18.2, “Setting up the CA on the New Host”.

After the migration, you can deactivate the OpenSSL CA or run it in read-only mode, where it only responds to Online Certificate Status Protocol (OCSP) requests.

PART V. UNINSTALLING CERTIFICATE SYSTEM SUBSYSTEMS

It is possible to remove individual subsystems or to uninstall all packages associated with an entire subsystem. Subsystems are installed and uninstalled individually. For example, it is possible to uninstall a KRA subsystem while leaving an installed and configured CA subsystem. It is also possible to remove a single CA subsystem while leaving other CA subsystems on the machine.

CHAPTER 20. REMOVING A SUBSYSTEM

Removing a subsystem requires specifying the subsystem type and the name of the server in which the subsystem is running. This command removes all files associated with the subsystem (without removing the subsystem packages).

```
pkidestroy -s subsystem_type -i instance_name
```

The **-s** option specifies the subsystem to be removed (such as CA, KRA, OCSP, TKS, or TPS). The **-i** option specifies the instance name, such as**pki-tomcat**.

Example 20.1. Removing a CA Subsystem

```
$ pkidestroy -s CA -i pki-tomcat
Loading deployment configuration from /var/lib/pki/pki-
tomcat/ca/registry/ca/deployment.cfg.
Uninstalling CA from /var/lib/pki/pki-tomcat.
Removed symlink /etc/systemd/system/multi-user.target.wants/pki-
tomcatd.target.

Uninstallation complete.
```

The **pkidestroy** utility removes the subsystem and any related files, such as the certificate databases, certificates, keys, and associated users. It does not uninstall the subsystem packages. If the subsystem is the last subsystem on the server instance, the server instance is removed as well.

CHAPTER 21. REMOVING CERTIFICATE SYSTEM SUBSYSTEM PACKAGES

A number of subsystem-related packages and dependencies are installed with Red Hat Certificate System; these are listed in Section 7.2, “Certificate System Packages”. Removing a subsystem removes only the files and directories associated with that specific subsystem. It does not remove the actual installed packages that are used by that instance. Completely uninstalling Red Hat Certificate System or one of its subsystems requires using package management tools, like **yum**, to remove each package individually.

To uninstall an individual Certificate System subsystem packages:

1. Remove all the associated subsystems. For example:

   ```
   pkidestroy -s CA -i pki-tomcat
   ```

2. Run the uninstall utility. For example:

   ```
   yum remove pki-subsystem_type
   ```

   The subsystem type can be **ca**, **kra**, **ocsp**, **tks**, or **tps**.

3. To remove other packages and dependencies, remove the packages specifically, using **yum**. The complete list of installed packages is atSection 7.2, “Certificate System Packages”.

GLOSSARY

A

access control

The process of controlling what particular users are allowed to do. For example, access control to servers is typically based on an identity, established by a password or a certificate, and on rules regarding what that entity can do. See also access control list (ACL).

access control instructions (ACI)

An access rule that specifies how subjects requesting access are to be identified or what rights are allowed or denied for a particular subject. See access control list (ACL).

access control list (ACL)

A collection of access control entries that define a hierarchy of access rules to be evaluated when a server receives a request for access to a particular resource. See access control instructions (ACI).

administrator

The person who installs and configures one or more Certificate System managers and sets up privileged users, or agents, for them. See also agent.

Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES), like its predecessor Data Encryption Standard

(DES), is a FIPS-approved symmetric-key encryption standard. AES was adopted by the US government in 2002. It defines three block ciphers, AES-128, AES-192 and AES-256. The National Institute of Standards and Technology (NIST) defined the AES standard in U.S. FIPS PUB 197. For more information, see http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

agent

A user who belongs to a group authorized to manage agent services for a Certificate System manager. See also Certificate Manager agent, Key Recovery Authority agent.

agent services

1. Services that can be administered by a Certificate System agent through HTML pages served by the Certificate System subsystem for which the agent has been assigned the necessary privileges.

2. The HTML pages for administering such services.

agent-approved enrollment

An enrollment that requires an agent to approve the request before the certificate is issued.

APDU

Application protocol data unit. A communication unit (analogous to a byte) that is used in communications between a smart card and a smart card reader.

attribute value assertion (AVA)

An assertion of the form *attribute* = *value*, where *attribute* is a tag, such as **o** (organization) or **uid** (user ID), and *value* is a value such as "Red Hat, Inc." or a login name. AVAs are used to form the distinguished name (DN) that identifies the subject of a certificate, called the subject name of the certificate.

audit log

A log that records various system events. This log can be signed, providing proof that it was not tampered with, and can only be read by an auditor user.

auditor

A privileged user who can view the signed audit logs.

authentication

Confident identification; assurance that a party to some computerized transaction is not an impostor. Authentication typically involves the use of a password, certificate, PIN, or other information to validate identity over a computer network. See also password-based authentication, certificate-based authentication, client authentication, server authentication.

authentication module

A set of rules (implemented as a Java™ class) for authenticating an end entity, agent, administrator, or any other entity that needs to interact with a Certificate System subsystem. In the case of typical end-user enrollment, after the user has supplied the

information requested by the enrollment form, the enrollment servlet uses an authentication module associated with that form to validate the information and authenticate the user's identity. See servlet.

authorization

Permission to access a resource controlled by a server. Authorization typically takes place after the ACLs associated with a resource have been evaluated by a server. See access control list (ACL).

automated enrollment

A way of configuring a Certificate System subsystem that allows automatic authentication for end-entity enrollment, without human intervention. With this form of authentication, a certificate request that completes authentication module processing successfully is automatically approved for profile processing and certificate issuance.

B

bind DN

A user ID, in the form of a distinguished name (DN), used with a password to authenticate to Red Hat Directory Server.

C

CA certificate

A certificate that identifies a certificate authority. See also certificate authority (CA), subordinate CA, root CA.

CA hierarchy

A hierarchy of CAs in which a root CA delegates the authority to issue certificates to subordinate CAs. Subordinate CAs can also expand the hierarchy by delegating issuing status to other CAs. See also certificate authority (CA), subordinate CA, root CA.

CA server key

The SSL server key of the server providing a CA service.

CA signing key

The private key that corresponds to the public key in the CA certificate. A CA uses its signing key to sign certificates and CRLs.

certificate

Digital data, formatted according to the X.509 standard, that specifies the name of an individual, company, or other entity (the subject name of the certificate) and certifies that a public key, which is also included in the certificate, belongs to that entity. A certificate is issued and digitally signed by a certificate authority (CA). A certificate's validity can be verified by checking the CA's digital signature through public-key cryptography techniques. To be trusted within a public-key infrastructure (PKI), a certificate must be issued and signed by a CA that is trusted by other entities enrolled in the PKI.

certificate authority (CA)

A trusted entity that issues a certificate after verifying the identity of the person or entity the certificate is intended to identify. A CA also renews and revokes certificates and generates CRLs. The entity named in the issuer field of a certificate is always a CA. Certificate authorities can be independent third parties or a person or organization using certificate-issuing server software, such as Red Hat Certificate System.

certificate chain

A hierarchical series of certificates signed by successive certificate authorities. A CA certificate identifies a certificate authority (CA) and is used to sign certificates issued by that authority. A CA certificate can in turn be signed by the CA certificate of a parent CA, and so on up to a root CA. Certificate System allows any end entity to retrieve all the certificates in a certificate chain.

certificate extensions

An X.509 v3 certificate contains an extensions field that permits any number of additional fields to be added to the certificate. Certificate extensions provide a way of adding information such as alternative subject names and usage restrictions to certificates. A number of standard extensions have been defined by the PKIX working group.

certificate fingerprint

A one-way hash associated with a certificate. The number is not part of the certificate itself, but is produced by applying a hash function to the contents of the certificate. If the contents of the certificate changes, even by a single character, the same function produces a different number. Certificate fingerprints can therefore be used to verify that certificates have not been tampered with.

Certificate Management Message Formats (CMMF)

Message formats used to convey certificate requests and revocation requests from end entities to a Certificate Manager and to send a variety of information to end entities. A proposed standard from the Internet Engineering Task Force (IETF) PKIX working group. CMMF is subsumed by another proposed standard, Certificate Management Messages over Cryptographic Message Syntax (CMC). For detailed information, see https://tools.ietf.org/html/draft-ietf-pkix-cmmf-02.

Certificate Management Messages over Cryptographic Message Syntax (CMC)

Message format used to convey a request for a certificate to a Certificate Manager. A proposed standard from the Internet Engineering Task Force (IETF) PKIX working group. For detailed information, see https://tools.ietf.org/html/draft-ietf-pkix-cmc-02.

Certificate Manager

An independent Certificate System subsystem that acts as a certificate authority. A Certificate Manager instance issues, renews, and revokes certificates, which it can publish along with CRLs to an LDAP directory. It accepts requests from end entities. See certificate authority (CA).

Certificate Manager agent

A user who belongs to a group authorized to manage agent services for a Certificate Manager. These services include the ability to access and modify (approve and reject) certificate requests and issue certificates.

certificate profile

A set of configuration settings that defines a certain type of enrollment. The certificate profile sets policies for a particular type of enrollment along with an authentication method in a certificate profile.

Certificate Request Message Format (CRMF)

Format used for messages related to management of X.509 certificates. This format is a subset of CMMF. See also Certificate Management Message Formats (CMMF). For detailed information, see http://www.ietf.org/rfc/rfc2511.txt.

certificate revocation list (CRL)

As defined by the X.509 standard, a list of revoked certificates by serial number, generated and signed by a certificate authority (CA).

Certificate System

See Red Hat Certificate System, Cryptographic Message Syntax (CS).

Certificate System console

A console that can be opened for any single Certificate System instance. A Certificate System console allows the Certificate System administrator to control configuration settings for the corresponding Certificate System instance.

Certificate System subsystem

One of the five Certificate System managers: Certificate Manager, Online Certificate Status Manager, Key Recovery Authority, Token Key Service, or Token Processing System.

certificate-based authentication

Authentication based on certificates and public-key cryptography. See also password-based authentication.

chain of trust

See certificate chain.

chained CA

See linked CA.

cipher

See cryptographic algorithm.

client authentication

The process of identifying a client to a server, such as with a name and password or with a certificate and some digitally signed data. See certificate-based authentication, password-based authentication, server authentication.

client SSL certificate

A certificate used to identify a client to a server using the SSL protocol. See Secure Sockets Layer (SSL).

CMC

See Certificate Management Messages over Cryptographic Message Syntax (CMC).

CMC Enrollment

Features that allow either signed enrollment or signed revocation requests to be sent to a Certificate Manager using an agent's signing certificate. These requests are then automatically processed by the Certificate Manager.

CMMF

See Certificate Management Message Formats (CMMF).

CRL

See certificate revocation list (CRL).

CRMF

See Certificate Request Message Format (CRMF).

cross-certification

The exchange of certificates by two CAs in different certification hierarchies, or chains. Cross-certification extends the chain of trust so that it encompasses both hierarchies. See also certificate authority (CA).

cross-pair certificate

A certificate issued by one CA to another CA which is then stored by both CAs to form a circle of trust. The two CAs issue certificates to each other, and then store both cross-pair certificates as a certificate pair.

cryptographic algorithm

A set of rules or directions used to perform cryptographic operations such as encryption and decryption.

Cryptographic Message Syntax (CS)

The syntax used to digitally sign, digest, authenticate, or encrypt arbitrary messages, such as CMMF.

cryptographic module

See PKCS #11 module.

cryptographic service provider (CSP)

A cryptographic module that performs cryptographic services, such as key generation, key storage, and encryption, on behalf of software that uses a standard interface such as that defined by PKCS #11 to request such services.

CSP

See cryptographic service provider (CSP).

D

decryption

Unscrambling data that has been encrypted. See encryption.

delta CRL

A CRL containing a list of those certificates that have been revoked since the last full CRL was issued.

digital ID

See certificate.

digital signature

To create a digital signature, the signing software first creates a one-way hash from the data to be signed, such as a newly issued certificate. The one-way hash is then encrypted with the private key of the signer. The resulting digital signature is unique for each piece of data signed. Even a single comma added to a message changes the digital signature for that message. Successful decryption of the digital signature with the signer's public key and comparison with another hash of the same data provides tamper detection. Verification of the certificate chain for the certificate containing the public key provides authentication of the signer. See also nonrepudiation, encryption.

distinguished name (DN)

A series of AVAs that identify the subject of a certificate. See attribute value assertion (AVA).

distribution points

Used for CRLs to define a set of certificates. Each distribution point is defined by a set of certificates that are issued. A CRL can be created for a particular distribution point.

dual key pair

Two public-private key pairs, four keys altogether, corresponding to two separate certificates. The private key of one pair is used for signing operations, and the public and private keys of the other pair are used for encryption and decryption operations. Each pair corresponds to a separate certificate. See also encryption key, public-key cryptography, signing key.

Key Recovery Authority

An optional, independent Certificate System subsystem that manages the long-term archival and recovery of RSA encryption keys for end entities. A Certificate Manager can be configured to archive end entities' encryption keys with a Key Recovery Authority before issuing new certificates. The Key Recovery Authority is useful only if end entities are encrypting data, such as sensitive email, that the organization may need to recover someday. It can be used only with end entities that support dual key pairs: two separate key pairs, one for encryption and one for digital signatures.

Key Recovery Authority agent

A user who belongs to a group authorized to manage agent services for a Key Recovery Authority, including managing the request queue and authorizing recovery operation using HTML-based administration pages.

Key Recovery Authority recovery agent

One of the *m of n* people who own portions of the storage key for theKey Recovery Authority.

Key Recovery Authority storage key

Special key used by the Key Recovery Authority to encrypt the end entity's encryption key after it has been decrypted with the Key Recovery Authority's private transport key. The storage key never leaves the Key Recovery Authority.

Key Recovery Authority transport certificate

Certifies the public key used by an end entity to encrypt the entity's encryption key for transport to the Key Recovery Authority. The Key Recovery Authority uses the private key corresponding to the certified public key to decrypt the end entity's key before encrypting it with the storage key.

E

eavesdropping

Surreptitious interception of information sent over a network by an entity for which the information is not intended.

Elliptic Curve Cryptography (ECC)

A cryptographic algorithm which uses elliptic curves to create additive logarithms for the mathematical problems which are the basis of the cryptographic keys. ECC ciphers are more efficient to use than RSA ciphers and, because of their intrinsic complexity, are stronger at smaller bits than RSA ciphers. For more information, see https://tools.ietf.org/html/draft-ietf-tls-ecc-12.

encryption

Scrambling information in a way that disguises its meaning. See decryption.

encryption key

A private key used for encryption only. An encryption key and its equivalent public key, plus a signing key and its equivalent public key, constitute adual key pair.

end entity

In a public-key infrastructure (PKI), a person, router, server, or other entity that uses a certificate to identify itself.

enrollment

The process of requesting and receiving an X.509 certificate for use in a public-key infrastructure (PKI). Also known as*registration*.

extensions field

See certificate extensions.

F

Federal Bridge Certificate Authority (FBCA)

A configuration where two CAs form a circle of trust by issuing cross-pair certificates to each other and storing the two cross-pair certificates as a single certificate pair.

fingerprint

See certificate fingerprint.

FIPS PUBS 140

Federal Information Standards Publications (FIPS PUBS) 140 is a US government standard for implementations of cryptographic modules, hardware or software that encrypts and decrypts data or performs other cryptographic operations, such as creating or verifying digital signatures. Many products sold to the US government must comply with one or more of the FIPS standards. See http://www.nist.gov/itl/fipscurrent.cfm.

firewall

A system or combination of systems that enforces a boundary between two or more networks.

I

impersonation

The act of posing as the intended recipient of information sent over a network. Impersonation can take two forms: spoofing and misrepresentation.

input

In the context of the certificate profile feature, it defines the enrollment form for a particular certificate profile. Each input is set, which then dynamically creates the enrollment form from all inputs configured for this enrollment.

intermediate CA

A CA whose certificate is located between the root CA and the issued certificate in a certificate chain.

IP spoofing

The forgery of client IP addresses.

J

JAR file

A digital envelope for a compressed collection of files organized according to the Java™ archive (JAR) format.

Java™ archive (JAR) format

A set of conventions for associating digital signatures, installer scripts, and other information with files in a directory.

Java™ Cryptography Architecture (JCA)

The API specification and reference developed by Sun Microsystems for cryptographic services. See http://java.sun.com/products/jdk/1.2/docs/guide/security/CryptoSpec.Introduction.

Java™ Development Kit (JDK)

Software development kit provided by Sun Microsystems for developing applications and

applets using the Java™ programming language.

Java™ Native Interface (JNI)

A standard programming interface that provides binary compatibility across different implementations of the Java™ Virtual Machine (JVM) on a given platform, allowing existing code written in a language such as C or C++ for a single platform to bind to Java™. See http://java.sun.com/products/jdk/1.2/docs/guide/jni/index.html.

Java™ Security Services (JSS)

A Java™ interface for controlling security operations performed by Netscape Security Services (NSS).

K

KEA

See Key Exchange Algorithm (KEA).

key

A large number used by a cryptographic algorithm to encrypt or decrypt data. A person's public key, for example, allows other people to encrypt messages intended for that person. The messages must then be decrypted by using the corresponding private key.

key exchange

A procedure followed by a client and server to determine the symmetric keys they will both use during an SSL session.

Key Exchange Algorithm (KEA)

An algorithm used for key exchange by the US Government.

L

Lightweight Directory Access Protocol (LDAP)

A directory service protocol designed to run over TCP/IP and across multiple platforms. LDAP is a simplified version of Directory Access Protocol (DAP), used to access X.500 directories. LDAP is under IETF change control and has evolved to meet Internet requirements.

linked CA

An internally deployed certificate authority (CA) whose certificate is signed by a public, third-party CA. The internal CA acts as the root CA for certificates it issues, and the third- party CA acts as the root CA for certificates issued by other CAs that are linked to the same third-party root CA. Also known as "chained CA" and by other terms used by different public CAs.

M

manual authentication

A way of configuring a Certificate System subsystem that requires human approval of

each certificate request. With this form of authentication, a servlet forwards a certificate request to a request queue after successful authentication module processing. An agent with appropriate privileges must then approve each request individually before profile processing and certificate issuance can proceed.

MD5

A message digest algorithm that was developed by Ronald Rivest. See also one-way hash.

message digest

See one-way hash.

misrepresentation

The presentation of an entity as a person or organization that it is not. For example, a website might pretend to be a furniture store when it is really a site that takes credit-card payments but never sends any goods. Misrepresentation is one form of impersonation. See also spoofing.

N

Netscape Security Services (NSS)

A set of libraries designed to support cross-platform development of security-enabled communications applications. Applications built using the NSS libraries support the Secure Sockets Layer (SSL) protocol for authentication, tamper detection, and encryption, and the PKCS #11 protocol for cryptographic token interfaces. NSS is also available separately as a software development kit.

non-TMS

Non-token management system. Refers to a configuration of subsystems (the CA and, optionally, KRA and OCSP) which do *not* handle smart cards directly.

See Also token management system (TMS).

nonrepudiation

The inability by the sender of a message to deny having sent the message. A digital signature provides one form of nonrepudiation.

O

object signing

A method of file signing that allows software developers to sign Java code, JavaScript scripts, or any kind of file and allows users to identify the signers and control access by signed code to local system resources.

object-signing certificate

A certificate that is associated private key is used to sign objects; related to object signing.

OCSP

Online Certificate Status Protocol.

one-way hash

1. A number of fixed-length generated from data of arbitrary length with the aid of a hashing algorithm. The number, also called a message digest, is unique to the hashed data. Any change in the data, even deleting or altering a single character, results in a different value.

2. The content of the hashed data cannot be deduced from the hash.

operation

The specific operation, such as read or write, that is being allowed or denied in an access control instruction.

output

In the context of the certificate profile feature, it defines the resulting form from a successful certificate enrollment for a particular certificate profile. Each output is set, which then dynamically creates the form from all outputs configured for this enrollment.

P

password-based authentication

Confident identification by means of a name and password. See also authentication, certificate-based authentication.

PKCS #10

The public-key cryptography standard that governs certificate requests.

PKCS #11

The public-key cryptography standard that governs cryptographic tokens such as smart cards.

PKCS #11 module

A driver for a cryptographic device that provides cryptographic services, such as encryption and decryption, through the PKCS #11 interface. A PKCS #11 module, also called a *cryptographic module* or *cryptographic service provider*, can be implemented in either hardware or software. A PKCS #11 module always has one or more slots, which may be implemented as physical hardware slots in some form of physical reader, such as for smart cards, or as conceptual slots in software. Each slot for a PKCS #11 module can in turn contain a token, which is the hardware or software device that actually provides cryptographic services and optionally stores certificates and keys. Red Hat provides a built-in PKCS #11 module with Certificate System.

PKCS #12

The public-key cryptography standard that governs key portability.

PKCS #7

The public-key cryptography standard that governs signing and encryption.

private key

One of a pair of keys used in public-key cryptography. The private key is kept secret and is used to decrypt data encrypted with the corresponding public key.

proof-of-archival (POA)

Data signed with the private Key Recovery Authority transport key that contains information about an archived end-entity key, including key serial number, name of the Key Recovery Authority, subject name of the corresponding certificate, and date of archival. The signed proof-of-archival data are the response returned by the Key Recovery Authority to the Certificate Manager after a successful key archival operation. See also Key Recovery Authority transport certificate.

public key

One of a pair of keys used in public-key cryptography. The public key is distributed freely and published as part of a certificate. It is typically used to encrypt data sent to the public key's owner, who then decrypts the data with the corresponding private key.

public-key cryptography

A set of well-established techniques and standards that allow an entity to verify its identity electronically or to sign and encrypt electronic data. Two keys are involved, a public key and a private key. A public key is published as part of a certificate, which associates that key with a particular identity. The corresponding private key is kept secret. Data encrypted with the public key can be decrypted only with the private key.

public-key infrastructure (PKI)

The standards and services that facilitate the use of public-key cryptography and X.509 v3 certificates in a networked environment.

R

RC2, RC4

Cryptographic algorithms developed for RSA Data Security by Rivest. See also cryptographic algorithm.

Red Hat Certificate System

A highly configurable set of software components and tools for creating, deploying, and managing certificates. Certificate System is comprised of five major subsystems that can be installed in different Certificate System instances in different physical locations: Certificate Manager, Online Certificate Status Manager,Key Recovery Authority, Token Key Service, and Token Processing System.

registration

See enrollment.

root CA

The certificate authority (CA) with a self-signed certificate at the top of a certificate chain. See also CA certificate, subordinate CA.

RSA algorithm

Short for Rivest-Shamir-Adleman, a public-key algorithm for both encryption and authentication. It was developed by Ronald Rivest, Adi Shamir, and Leonard Adleman and introduced in 1978.

RSA key exchange

A key-exchange algorithm for SSL based on the RSA algorithm.

S

sandbox

A Java™ term for the carefully defined limits within which Java™ code must operate.

secure channel

A security association between the TPS and the smart card which allows encrypted communciation based on a shared master key generated by the TKS and the smart card APDUs.

Secure Sockets Layer (SSL)

A protocol that allows mutual authentication between a client and server and the establishment of an authenticated and encrypted connection. SSL runs above TCP/IP and below HTTP, LDAP, IMAP, NNTP, and other high-level network protocols.

security domain

A centralized repository or inventory of PKI subsystems. Its primary purpose is to facilitate the installation and configuration of new PKI services by automatically establishing trusted relationships between subsystems.

self tests

A feature that tests a Certificate System instance both when the instance starts up and on-demand.

server authentication

The process of identifying a server to a client. See also client authentication.

server SSL certificate

A certificate used to identify a server to a client using the Secure Sockets Layer (SSL) protocol.

servlet

Java™ code that handles a particular kind of interaction with end entities on behalf of a Certificate System subsystem. For example, certificate enrollment, revocation, and key recovery requests are each handled by separate servlets.

SHA-1

Secure Hash Algorithm, a hash function used by the US government.

signature algorithm

A cryptographic algorithm used to create digital signatures. Certificate System supports the MD5 and SHA-1 signing algorithms. See also cryptographic algorithm, digital signature.

signed audit log

See audit log.

signing certificate

A certificate that is public key corresponds to a private key used to create digital signatures. For example, a Certificate Manager must have a signing certificate that is public key corresponds to the private key it uses to sign the certificates it issues.

signing key

A private key used for signing only. A signing key and its equivalent public key, plus an encryption key and its equivalent public key, constitute adual key pair.

single sign-on

1. In Certificate System, a password that simplifies the way to sign on to Red Hat Certificate System by storing the passwords for the internal database and tokens. Each time a user logs on, he is required to enter this single password.

2. The ability for a user to log in once to a single computer and be authenticated automatically by a variety of servers within a network. Partial single sign-on solutions can take many forms, including mechanisms for automatically tracking passwords used with different servers. Certificates support single sign-on within a public-key infrastructure (PKI). A user can log in once to a local client's private-key database and, as long as the client software is running, rely on certificate-based authentication to access each server within an organization that the user is allowed to access.

slot

The portion of a PKCS #11 module, implemented in either hardware or software, that contains a token.

smart card

A small device that contains a microprocessor and stores cryptographic information, such as keys and certificates, and performs cryptographic operations. Smart cards implement some or all of the PKCS #11 interface.

spoofing

Pretending to be someone else. For example, a person can pretend to have the email address **jdoe@example.com**, or a computer can identify itself as a site called **www.redhat.com** when it is not. Spoofing is one form ofimpersonation. See also misrepresentation.

SSL

See Secure Sockets Layer (SSL).

subject

The entity identified by a certificate. In particular, the subject field of a certificate contains a subject name that uniquely describes the certified entity.

subject name

A distinguished name (DN) that uniquely describes thesubject of a certificate.

subordinate CA

A certificate authority that is certificate is signed by another subordinate CA or by the root CA. See CA certificate, root CA.

symmetric encryption

An encryption method that uses the same cryptographic key to encrypt and decrypt a given message.

T

tamper detection

A mechanism ensuring that data received in electronic form entirely corresponds with the original version of the same data.

token

A hardware or software device that is associated with a slot in a PKCS #11 module. It provides cryptographic services and optionally stores certificates and keys.

token key service (TKS)

A subsystem in the token management system which derives specific, separate keys for every smart card based on the smart card APDUs and other shared information, like the token CUID.

token management system (TMS)

The interrelated subsystems — CA, TKS, TPS, and, optionally, the KRA — which are used to manage certificates on smart cards (tokens).

token processing system (TPS)

A subsystem which interacts directly the Enterprise Security Client and smart cards to manage the keys and certificates on those smart cards.

tree hierarchy

The hierarchical structure of an LDAP directory.

trust

Confident reliance on a person or other entity. In a public-key infrastructure (PKI), trust refers to the relationship between the user of a certificate and the certificate authority (CA) that issued the certificate. If a CA is trusted, then valid certificates issued by that CA can be trusted.

V

virtual private network (VPN)

A way of connecting geographically distant divisions of an enterprise. The VPN allows the divisions to communicate over an encrypted channel, allowing authenticated, confidential transactions that would normally be restricted to a private network.

INDEX

A

C

K

L

O

P

R

S

T

U

APPENDIX A. REVISION HISTORY

Note that revision numbers relate to the edition of this manual, not to version numbers of Red Hat Certificate System.

Revision 9.4-0 **Thu Oct 25 2018** **Marc Muehlfeld**
Red Hat Certificate System 9.4 release of the guide.

Revision 9.3-2 **Thu May 03 2018** **Marc Muehlfeld**
Added the *Enrollment Using the Command Line* section

Revision 9.3-1 **Tue Apr 10 2018** **Marc Muehlfeld**
For version 9.3: Rewrote the *Prerequisites and Preparation for Installation* and *Installing and Configuring Certificate System* chapters

Revision 9.2-2 **Tue Dec 12 2017** **Petr Bokoč**
Asynchronous update

Revision 9.2-1 **Tue Aug 01 2017** **Petr Bokoč**
Red Hat Certificate System 9.2 GA release

Revision 9.1-2 **Thu Mar 09 2017** **Petr Bokoč**
Asynchronous update

Revision 9.1-0 **Tue Nov 01 2016** **Petr Bokoč**
Red Hat Certificate System 9.1 release

Revision 9.0-3 **Tue Aug 02 2016** **Petr Bokoč**
Asynchronous update

Revision 9.0-2 **Thu Oct 22 2015** **Aneta Petrová**
Updated the transport key rotation section

Revision 9.0-1 **Fri Sep 04 2015** **Tomáš Čapek**
Section on transport key rotation added

Revision 9.0-0 **Fri Aug 28 2015** **Tomáš Čapek**
Version for Red Hat Certificate System 9 release