Public Key Infrastructure Best Practices

Open API for FSP Interoperability

Public Key Infrastructure Best Practices

Open API for FSP Interoperability Specification

Table of Contents

Table of Figures

Table of Tables

1 Preface

This section contains information about how to use this document.

1.1 Conventions Used in This Document

The following conventions are used in this document to identify the specified types of information

Type of Information	Convention	Example
Elements of the API, such at resources	Boldface	**/authorization**
Variables	Italics witin angle brackets	*<ID>*
Glossary terms	Italics on first occurence; defined in *Glossary*	The purpose of the API is to enable interoperable financial transactions between a *Payer* (a payer of electronic funds in a payment transaction) located in one *FSP* (an entity that provides a digital financial service to an end user) and a *Payee* (a recipient of electronic funds in a payment transaction) located in another FSP.
Library documents	Italics	User information should, in general, not be used by API deployments; the security measures detailed in *API Signature* and *API Encryption* should be used instead.

1.2 Document Version Information

Version	Date	Change Description
1.0	2018-03-21	Initial version

2 Introduction

This document explains *Public Key Infrastructure* (PKI)[1] best practices to apply in an *Open API for FSP Interoperability* (hereafter cited as “the API”*)* deployment. See Chapter 3, PKI Background, for more information about PKI.

The API should be implemented in an environment that consists of either:

- *Financial Service Providers* (FSPs) that communicate with other FSPs (in a bilateral setup) or
- A *Switch* that acts as an intermediary platform between FSP platforms. There is also an *Account Lookup System* (ALS) available to identify in which FSP an account holder is located.

For more information about the environment, see Chapter 4, Network Topology.

Chapters 5 and 6 identify management strategies for the CA and for the platform.

Communication between platforms is performed using a *REST* (REpresentational State Transfer)-based HTTP protocol (for more information, see *API Definition*). Because this protocol does not provide a means for ensuring either integrity or confidentiality between platforms, extra security layers must be added to protect sensitive information from alteration or exposure to unauthorized parties.

[1] This term, and other italicized terms, are defined in *Glossary* for *Open API for FSP Interoperability Specification*.

2.1 Transport-Level Protection

To protect the transport level, *Transport Layer Security*[2] (TLS) should be used. TLS is a fundamental technique used for securing point-to-point communication. It has been proven to be stable and secure when using strong algorithms with the most recent versions of TLS, and is widely used around the world. TLS is a secure mechanism for exchanging a shared symmetric key between two anonymous peers, with identity verification (that is, trusted certificates). It provides *confidentiality* (no one has read the content) and *integrity* (no one has changed the content). Using TLS efficiently as-is requires certificate management.

[2] https://tools.ietf.org/html/rfc5246 - The Transport Layer Security (TLS) Protocol - Version 1.2

2.2 Application-Level Protection

This layer provides end-to-end integrity and confidentiality. The API uses the JSON Web Signature[3] (JWS) standard for integrity and *non-repudiation* (provide proof of the integrity and origin of data), and the JSON Web Encryption (JWE)[4] standard for confidentiality. An extended version of JWE is used to support field level encryption.

Using these standards requires certificate management; therefore, *Certificate Authorities* (CA) and related PKI techniques are needed. For more information, see PKI Background.

[3] https://tools.ietf.org/html/rfc7515 - JSON Web Signature (JWS)

[4] https://tools.ietf.org/html/rfc7516 - JSON Web Encryption (JWE)

3 PKI Background

Public Key Infrastructure (PKI) is a set of standards, procedures, and software for implementing authentication using public key cryptography. PKI is used to request, install, configure, manage and revoke digital certificates. PKI offers authentication using digital certificates; these digital certificates are signed and provided by *Certificate Authorities* (CA).

PKI uses public key cryptography and works with X.509 standard certificates. It also provides features such as:

- User authentication
- Certificate production and distribution
- Certificate maintenance, management, and revocation

PKI consists of a number of components that enable the infrastructure to work; it is not a single process or algorithm.. In addition to authentication, PKI also enables the provision of integrity, non-repudiation and encryption.

To obtain a public key, a company must obtain a digital certificate. It should request this certificate from a CA or a *Registration Authority* (RA) - an organization that processes requests on behalf of a CA. All participants must trust the CA to manage and maintain certificates. The CA requires the company to supply a number of details (*Common Name* (CN), *Organization* (O), *Country* (C) and so on) and validate their request before it provides a certificate. This certificate is proof that the company is who it says it is in the digital world (like a passport in the real world).

PKI combines well with a *Diffie-Hellman solution* (a secure mechanism for exchanging a shared symmetric key between two anonymous peers) in providing secure key exchanges. Because Diffie-Hellman does not provide authentication, PKI is used with additional protocols, such as *Pretty Good Privacy* (PGP) and *Transport Layer Security* (TLS).

4 Network Topology

This section identifies the platforms that constitute the API.

4.1 Platforms Point-to-Point Layout

Figure 1 shows an example platform point-to-point layout.

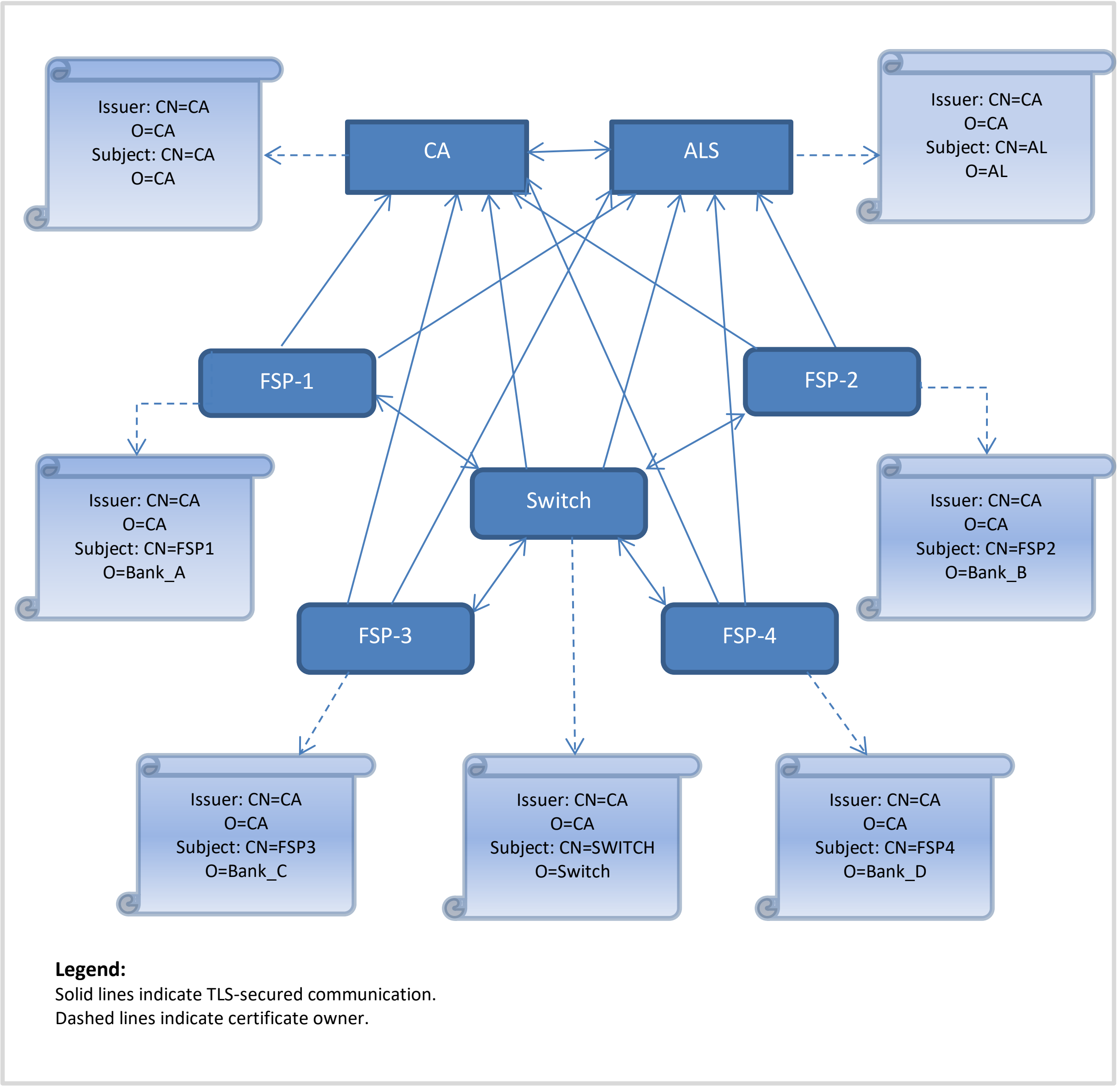

Figure 1 - Platforms layout

All communication between platforms must be TLS-secured using *client authentication*, also known as *mutual authentication*.

4.2 Platform Roles

4.2.1 Certificate Authority (CA)

The CA performs the following functions:

- Sign *Certificate Signing Requests* (CSRs) - CSRs is a secure mechanism for exchanging a shared symmetric key between two anonymous peers. The CA signs different types of certificates (for example, TLS, Content signature, Content encryption).
- Revoke certificates – Mark one or more certificates as invalid.
- Support CRLs – Maintain and provide certificate revocation lists (CRLs) to be downloaded by clients so that they can see which certificates have been revoked.
- Support Online Certificate Status Protocol (OCSP) – Provide real-time certificate revocation checks.

4.2.2 Account Lookup System (ALS)

- Holds basic information about account holders.
- Answer questions like "Where should I send my financial transaction request for account holder with **MSISDN 0123456**?"

4.2.3 Financial Services Provider (FSP)

Has account holders to which or from which money is transferred.

4.2.4 Switch

- Relays transaction information to other platforms.
- Can perform financial services, as specified in *API Definition*.

5 Certificate Authority PKI Management Strategy

This section describes the PKI management strategy for Certificate Authorities.

5.1 Certificate Authority Importance and Selection Criteria

The Certificate Authority (CA) role is important because:

- The CA provides a single legal entity that is trustworthy for all platforms.
- The point-to-point TLS protocol depends on certificates.
- The end-to-end protocols JWS and JWE depend on certificates for proof of non-repudiation and confidentiality.

5.1.1 Reasons Not to Use a Public CA

- A public CA can revoke the intermediate certificate used for signing our certificates, thus effectively shutting down all communication between platforms.
- A public CA also signs certificates which are not part of the *Open API for FSP Interoperability* setup. Because you trust the certificate the CA signed for you, you then trust all certificates signed by that CA.
- There is no service in the *Open API for FSP Interoperability* setup that is open to the public, so there is no reason to have a public CA already trusted by public clients (such as web browsers).

5.1.2 Private CA Options

- Build your own CA from scratch
- Build a CA using existing tools (for example, *openssl*)
- Use a full-featured CA (for example, the open source product *EJBCA*)

5.2 Trusted Certificate Chain

A centralized CA makes certificate management easier for the involved platforms. By trusting the same CA certificate, each platform need only trust one certificate; the CA's self-signed root certificate.

Thus, the CA should sign all types of certificates (TLS, signature, and encryption), but only for the participants in the *Open API for FSP Interoperability* setup.

No intermediate CA certificates are required, because a segmented layout is not used in the *Open API for FSP Interoperability* setup.

5.3 CA Root Certificate

- The CA must create a self-signed root certificate to be used for signing all supported types of certificates (TLS, signing, and encryption).
- The minimum key length of the asymmetric RSA key pair is 4096 bits, and the signature algorithm should be sha512WithRSAEncryption.
- The *X.509 Basic Constraints* must have the attribute **`CA`** set to **`TRUE`**.
- The time validity of the CA root certificate should be ten years. After eight years, a new self-signed root certificate should be created by the CA; the asymmetric RSA key pair must be recreated, not reused. This certificate becomes the certificate to use for signing the platforms CSRs. However, the old CA root certificate must still be active until it expires. This gives the platforms a window of two years to change the CA root certificate without disturbing their ongoing secured communication.

5.4 Signing Platforms CSRs

The CA must provide a mechanism for the platforms to get their CSRs signed. Common signing methods are by email and by a web page. The chosen solution and policy must be known to the platforms.

The CA signs three types of platform certificates; TLS (for communication), JWS (for signature) and JWE (for encryption). Common requirements for these certificates are:

- The minimum key length of the asymmetric RSA key pair is 2048 bits, and the signature algorithm should be sha256WithRSAEncryption.
- The *X.509 Basic Constraints* must have the attribute `CA` set to `FALSE`.
- The subject distinguished names must contain at least the following attributes:
    - *Common Name* (CN) – This must be the hostname of the platform which has created the certificate. A N can never be the same for two different platforms or organizations.
    - *Organization* (O) – The name of the organization.
    - *Country* (C) – The country of the organization.
- The URL for platforms to download CRLs must be present.
- The URL for platforms to send OCSP requests must be present.
- The time validity should be two years.

Depending on the type of certificate to sign, the *X.509 Key Usage* and *X.509 Extended Key Usage* of the platform's certificate will differ; see Table 1 for more information.

Certificate Type	X.509 Key Usage	X.509 Extended Key Usage
TLS Certificates	Digital Signature, Key Encipherment	TLS Web Server Authentication, TLS Web Client Authentication
Signature Certificates	Digital Signature	
Encryption Certificates	Key Encipherment	

Table 1 – Certificate type and key usage

5.5 Revoking Certificates

- The CA must be able to revoke a platform's certificates. Revoking a certificate means that it will no longer be trusted by any party. It will be marked as invalid by the CA and its revocation status published to the platforms, either in a revocation list (CRL) to be downloaded by platforms, or through a real-time HTTP query using *Online Certificate Status Protocol* (OCSP) requested by platforms.
- The CA should support both CRL downloads and OCSP queries.
- The CA should update and sign the CRL daily and should provide (daily) a HTTP URL to the platforms that points to the CRL to be downloaded. The URL should be stored in the *CRL Distribution Points* for each signed certificate.
- The OCSP URL should be stored in the *Authority Information Access* for each signed certificate. An OCSP response must be signed. Nonce values in an OCSP request should be supported to prevent reply attacks.
- The CA has the right to revoke certificates, but no obligation to inform the platforms about such revocations. The platforms do not have the right to revoke certificates, but they do have an obligation to check regularly for a certificate's revocation status.

5.6 Certificate Enrollment and Renewal

The CA does not support enrollment or renewal of certificates for which an asymmetric key pair is reused. For increased security, the asymmetric key pairs must be recreated every time a certificate is enrolled or renewed through a new CSR.

6 Platform PKI Management Strategy

This section describes the PKI management strategy for platforms.

6.1 Keys, Certificates, and Stores

A certificate provides the identity of its owner through a trusted CA that has signed the certificate. This can be validated through its signed certificate chain. It also provides the technical means to ensure integrity (signature) and confidentiality (encryption) of data using its asymmetric key pair (known as a *public key* and *private key*). The public key is within the signed certificate, but its corresponding private key must be kept private and protected. A common solution for handling certificates and private keys is to put them into a protected storage area known as a *store*. A store can be a file, directory, or something else that provides access and confidentiality.

A single private key and its associated certificate can be used for all cryptographic purposes, but to enhance security, the following set of keys and certificates is required for each platform:

- One private key and certificate for TLS communication
- One private key and certificate for end-to-end integrity using JWS
- One private key and certificate for end-to-end confidentiality using JWE

Different keys and types of certificates can all be in the same store, but a more common setup is the following:

- For TLS communication:
    - One protected key store to hold the private key, its associated certificate, and certificate chain. These items are used for server- and client authentication in TLS.
    - One protected certificate store to hold all trusted TLS certificates. These certificates will be trusted by your platform during a TLS handshake, meaning that they will allow secure communication with the owners of the certificates. Since all participants trust the same CA, it is sufficient to keep only the CA's root certificate here.
- For signature and encryption:
    - One protected key store to hold your private keys, their associated certificates, and certificate chains used for signatures and encryption.
    - One private key and certificate chain used for signing using JWS, and one private key and certificate chain used for encryption using JWE.
    - One protected certificate store to hold all trusted signature and encryption certificates from other platforms. Here you must store the certificates (signature and encryption certificates) for each platform that you want to trust for end-to-end integrity and confidentiality.

6.2 Creating a CSR and Obtaining CA Signature

To be able to communicate with other platforms, you must create a key store (if one does not already exist), an asymmetric key pair, and an associated certificate that identifies your platform. This unsigned certificate must be signed by the CA before it can be trusted by other platforms. The procedure of getting a certificate signed by a CA begins with a CSR (*Certificate Signing Request*).

When you create your keys, certificates, and CSRs, the following requirements apply:

- The minimum key length of the asymmetric RSA key pair is 2048 bits, and the signature algorithm is sha256WithRSAEncryption.
- The following attributes in the subject distinguished names are mandatory:
    - Common Name (CN) – This must be the hostname of the platform who created the certificate. A CN can never be the same for two different platforms or organizations.
    - Organization (O) – The name of the organization.
    - Country (C) – The country of the organization.

For examples of how to create a store, certificate, and CSR, see Appendix C – Common PKI Tasks

How to create a store, certificate, and CSR.

Create your CSR and send it to your CA according to their instructions.

Note: The same procedure must be performed for all your different keys and certificates (TLS, signature, and encryption).

6.3 Importing a Signed CSR

After signing your CSR, you will have two certificates in your response from the CA. The first certificate is your own signed certificate. The second is the CA's own root certificate, which was used for signing your certificate. That certificate must now be trusted by you.

First you must import your signed certificate into your key store. It must replace your unsigned certificate. For examples, see How to import a signed certificate into your key store.

Then you must import the CA's root certificate into the same key store to complete the chain between your certificate and the CA. For examples, see How to import the CA certificate into your key store.

6.3.1 TLS Certificates

For TLS certificates, you must also import the CA's root certificate into your TLS trust store to automatically trust the other platforms when establishing new communication channels with them. For examples, see How to import the CA certificate into TLS trust store.

The above procedure should be performed for each of your signed CSRs. Remember to send your signed certificate and the CA's root certificate to all other platforms with which you need to communicate.

The CA will create a validity period of two years for your signed certificate. After 18 months, you should create a new CSR to be signed by the same CA. Your newly signed certificate and the CA's root certificate should once again be imported into your stores and sent to the other platforms. This will give you and your peers a window of six months to make sure that your new signed certificate works. After two years, the old expired certificate can be deleted.

6.4 Trusting Certificates from Other Platforms

Just as your peers must have your certificates to be able to trust you, they must send their certificates to you for you to trust them.

Make sure that you always receive at least two certificates from a trusted peer:

- The signed certificate of the peer to trust.
- The root certificate of the CA that signed the peer's certificate.

Note: A peer's CA certificate can differ from yours. This can happen if the CA has created a new CA root certificate when the old one is about to expire.

You should always view a peer's certificates (check CN, validity period, and so on) and validate the certificate chain (that is, that each certificate is correctly signed by the given CA) before you import them into your trust store.
For examples, see How to view a certificate.

Next, import the peer's certificate and its CA root certificate into your trust store. For examples, see How to import a trusted certificate.

6.5 Checking Certificate Revocation Status

Certificates can be revoked by the CA. A revoked certificate is considered not trusted and should be removed from the trust store. A certificate can also be *on hold*, which means that it is in pending state due to investigation and should not be removed.

All platforms should perform revocation checks on a regular basis. There are two common ways to find out which certificates have been revoked; either through viewing a revocation list (CRL) to be downloaded by platforms, or through a real-time HTTP query (OCSP) requested by platforms.

6.5.1 Certificate Revocation List

This is a file maintained by a CA. It contains a list of certificate serial numbers that have been revoked by the CA. This file can be downloaded by any platform at any time. The URL to the CRL file to be downloaded is included in the certificate itself and can be found in *CRL Distribution Points*. A CRL is signed by the CA and should be validated.

A CRL should be downloaded once per day and cached by the platform.

For examples, see How to verify a certificate through a CRL.

6.5.2 Online Certificate Status Protocol

The status of a certificate can be retrieved by sending an OCSP request to an *OCSP Responder* (which can be the CA). The request contains the serial number of the certificate to be checked. The responder sends back a signed response including the status of the certificate.

The request should contain a nonce value, which the responder will put in the response, for the platform to validate on receipt. This is to prevent reply attacks.

An OCSP request/response is considered very fast and can be executed for each received client certificate operation, but depending on the load towards the responder, it might have to be cached by the platform as well.

The OCSP URL is included in the certificate itself and can be found in *Authority Information Access*.

For examples, see How to verify a certificate through an OCSP request.

6.6 Renewing Certificates

Do not renew certificates for which an asymmetric key pair is reused. For increased security, the asymmetric key pairs must be recreated every time through a new CSR.

7 Transport Layer Protection

This section describes transport layer protection.

7.1 TLS

TLS provides point-to-point integrity and confidentiality, and is used for all communication between peers.

The setup requires both *server authentication* - meaning that the server presents itself through its own TLS certificate - and *client authentication* (also known as *mutual authentication*) in which the client presents itself through its own TLS certificate.

7.1.1 TLS Protocol Versions

The TLS protocol version used must be 1.2 or higher.

7.1.2 TLS Cipher Suites

The following cipher suites should be used:

- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256
- TLS_RSA_WITH_AES_256_GCM_SHA384
- TLS_RSA_WITH_AES_128_GCM_SHA256

8 Application Layer Protection

This section describes the application layer protection.

8.1 JSON Web Signature

The *JSON Web Signature* (JWS) standard is used for providing end-to-end integrity and non-repudiation; that is, to guarantee that the sender is who it claims to be, and that the message was not tampered with.

The use of JWS is mandatory and certificates should be used. For more information, see *API Signature*.

8.2 JSON Web Encryption

The *JSON Web Encryption* (JWE) standard is used for providing end-to-end confidentiality; that is, to protect data from being read by unauthorized parties.

The use of JWE is optional and is applied to specific fields. This is to fulfill regulatory requirements that might exist for certain type of data in certain countries.

For more information about how to apply JWE to fields in messages, see the extended JWE standard specification *API Encryption*.

Appendix A Key Lengths and Algorithms

Table 2 contains the required key length and algorithm for each entity.

Entity	Algorithm	Key length/hash size
CA asymmetric keys	RSA	4096 bits
CA signing algorithm	RSA with SHA2	>= 256 bits
TLS asymmetric keys	RSA	2048 bits
TLS signing algorithm	RSA with SHA2	>= 256 bits
Signature asymmetric keys	RSA	2048 bits
Signature algorithm	RSA with SHA2	>= 256 bits
Encryption asymmetric keys	RSA	2048 bits
HMAC	SHA2 - AES	>= 256 bits
Symmetric keys	AES	256 bits
Hashing	SHA2	>= 256 bits

Table 2 – Key lengths and algorithms

Appendix B Terminology

PKI	Public Key Infrastructure
API	Application Program Interface
TLS	Transport Layer Security
JWS	JSON Web Signature
JWE	JSON Web Encryption
FSP	Financial Service Provider
AL	Account Lookup
CA	Certificate Authority
CSR	Certificate Signing Request
CRL	Certificate Revocation List
OCSP	Online Certificate Status Protocol
PEM	Privacy Enhanced Mail
RSA	Rivest, Shamir, & Adleman
HMAC	Hashed Message Authentication Code
AES	Advanced Encryption Standard
SHA	Secure Hash Algorithm

Appendix C – Common PKI Tasks

Appendix C.1 How to create a store, certificate, and CSR

Using the Java keytool

Use the following command to create an asymmetric RSA key pair and associated certificate information to be signed by the CA. It will automatically create a JKS key store if the given one does not exist:

```
keytool -genkey -dname "CN=<common-name>" -alias <key-alias> -keyalg RSA -
keystore <ks-path> -keysize 2048
```

Example

```
keytool -genkey -dname "CN=bank-fsp" -alias tlscert -keyalg RSA -keystore
tlskeystore.jks -keysize 2048
```

Notes:

1. The certificate attribute CN specifies the hostname of the host who identifies itself using this certificate during a TLS session.
2. When asked for password, use the same password for the key as the key store.

Use the following command to create the actual CSR to be signed by the CA:

```
keytool -certreq -alias <key-alias> -keystore <ks-path> –file <certification-
request>.csr
```

Example

```
keytool -certreq -alias tlscert -keystore tlskeystore.jks -file tlscert.csr
```

Using openssl

Use the following command to create an asymmetric RSA key pair, and CSR to be signed by the CA:

```
openssl req -new -newkey rsa:2048 -nodes -subj "/CN=<common-name>" -out
<certificate>.csr -keyout <private-key>.key
```

Example

```
openssl req -new -newkey rsa:2048 -nodes -subj "/CN=bank-fsp" -out tlscert.csr -
keyout tlscert.key
```

Note: The certificate attribute CN specifies the hostname of the host who identifies itself using this certificate during a TLS session.

Important to note is that the created private key is stored unencrypted. Use the following command to encrypt it:

```
openssl rsa -aes256 -in <unenrypted key file> -out <encrypted key file>
```

Example

```
openssl rsa -aes256 -in tlscert.key -out tlscert_encrypted.key
```

Appendix C.2 How to import a signed certificate into your key store

Using the Java keytool

Import your signed certificate into your key store. This certificate should replace the old unsigned one, so make sure that you use the correct alias.

```
keytool -importcert -alias <key-alias> -file <signed-certificate> -keystore <ks-
path>
```

Example

```
keytool -importcert -alias tlscert -file bank-fsp.pem -keystore tlskeystore.jks
```

Using openssl

Remove your old CSR file and replace it with the new signed certificate.

Appendix C.3 How to import the CA certificate into your key store

Using Java keytool

Import the CA certificate into your key store:

```
keytool -importcert -alias <CA-alias> -file <CA-file> -keystore <ks-path>
```

Example:

```
keytool -importcert -alias rootca -file rootca.pem -keystore tlskeystore.jks
```

When you are asked to trust the imported certificated, answer **yes**:

```
Trust this certificate? [no]:  yes
Certificate was added to keystore
```

Using openssl

Put the CA certificate with your other certificate files.

Appendix C.4 How to import the CA certificate into TLS trust store

Using Java keytool

Import the CA certificate into your trust store:

```
keytool -importcert -alias <CA-alias> -file <CA-file> -keystore <ks-path>
```

Example:

```
keytool -importcert -alias rootca -file rootca.pem -keystore tlstruststore.jks
```

When you are asked to trust the imported certificated, answer **yes**:

```
Trust this certificate? [no]:  yes
Certificate was added to keystore
```

Using openssl

Put the CA certificate with your other certificate files.

Appendix C.5 How to view a certificate

Using Java keytool

Use this command to list all certificates stored in a key store in readable format:

```
keytool -list -keystore <ks-path> -v
```

Example:

```
keytool -list -keystore tlskeystore.jks -v
```

Using openssl

Use this command to show the content of a PEM encoded certificate in readable format:

```
openssl x509 -in <certificate file> -text -nout
```

Example:

```
openssl x509 -in rootca.pem -text -nout
```

Appendix C.6 How to import a trusted certificate

Using Java keytool

Import the certificate into your trust store:

```
keytool -importcert -alias <cert-alias> -file <cert-file> -keystore <ks-path>
```

Example:

```
keytool -importcert -alias trustedcert -file cert.pem -keystore truststore.jks
```

When you are asked to trust the imported certificated, answer **yes**:

```
Trust this certificate? [no]:  yes
Certificate was added to keystore
```

Using openssl

Put the trusted certificate with your other certificate files.

Appendix C.7 How to verify a certificate through a CRL

Using Java

You can use a specific security library for this (for example, Bouncy Castle), or use the in-built support for it, see: https://stackoverflow.com/questions/10043376/java-x509-certificate-parsing-and-validating/10068006#10068006

Using openssl

A good example is found at the following link:
https://raymii.org/s/articles/OpenSSL_manually_verify_a_certificate_against_a_CRL.html

Appendix C.8 How to verify a certificate through an OCSP request

Using Java

Some Java examples is found in the following link:
https://stackoverflow.com/questions/5161504/ocsp-revocation-on-client-certificate

Using openssl

A good example is found at the following link:
https://raymii.org/s/articles/OpenSSL_Manually_Verify_a_certificate_against_an_OCSP.html