# Platform SDK Developer's Guide

## Using and Configuring Security Providers

12/15/2025

# Using and Configuring Security Providers

## Java

> **Tip**
> The contents of this page only apply to Java implementations.

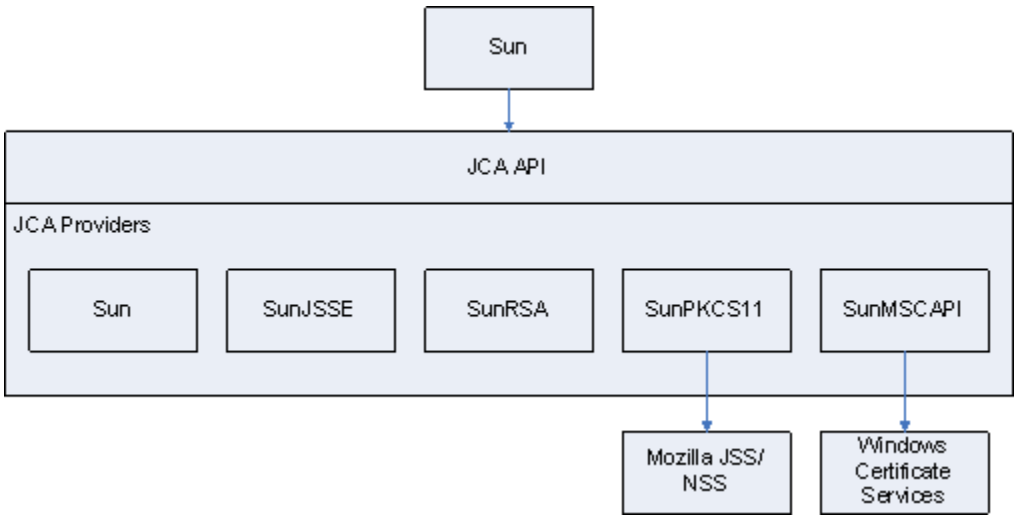
## Introduction

This page deals with Security Providers — an umbrella term describing the full set of cryptographic algorithms, data formats, protocols, interfaces, and related tools for configuration and management when used together. The primary reasons for bundling together such diverse tools are: compatibility, support for specific standards, and implementation restrictions.

The security providers listed here were tested with the Platform SDK 8.1.1 implementation of TLS, and found to work reliably when used with the configuration described below.

### Java Cryptography Architecture Notes

Java Cryptography Architecture (JCA) provides a general API, and a pluggable architecture for cryptography providers that supply the API implementation.




Some JCA providers (Sun, SunJSSE, SunRSA) come bundled with the Java platform and contain actual

algorithm implementations, they are named PEM provider since they are used when working with certificates in PEM files. Some other (SunPKCS11, SunMSCAPI) serve as a façade for external providers. SunPKCS11 supports PKCS#11 standard for pluggable security providers, such as hardware cryptographic processors, smartcards or software tokens. Mozilla NSS/JSS is an example of pluggable software token implementation. SunMSCAPI provides access to Microsoft Cryptography API (MSCAPI), in particular, to Windows Certificate Services (WSC).

## PEM Provider: OpenSSL

**Note:** Working with certificates and keys is also covered in the *Genesys 8.1 Security Deployment Guide*.

PEM stands for "Privacy Enhanced Mail", a 1993 IETF proposal for securing email using public-key cryptography. That proposal defined the PEM file format for certificates as one containing a Base64-encoded X.509 certificate in specific binary representation with additional metadata headers. Here, the term is used to refer to Java built-in security providers that are used in conjunction with certificates and private keys loaded from X.509 PEM files.

One of the most popular free tools for creating and manipulating PEM files is OpenSSL. Instructions for installing and configuring OpenSSL are provided below.

### Installing OpenSSL

OpenSSL is available two ways:

- distributed as a source code tarball: http://www.openssl.org/source/
- as a binary distribution (specific links are subject to change): http://www.openssl.org/related/binaries.html

The installation process is very easy when using a binary installer; simply follow the prompts. The only additional step required is to add the *<OpenSSL-home>\bin* folder to your *Path* system variable so that OpenSSL can run from command line directly with the `openssl` command.

### Configuring OpenSSL

The OpenSSL configuration file contains settings for OpenSSL itself, and also many field values for the certificates being generated including issuer and subject names, host names and URIs, and so on. You will need to customize your OpenSSL file with your own values before using the tool. An example of a customized configuration file is available here.

The OpenSSL database consists of a set of files and folders, similar to the sample database described in the table below. To start using OpenSSL, this structure should be created manually except for files marked as "Generated by OpenSSL". Other files can be left empty as long as they exist in the expected location.

**OpenSSL database file/folder structure**

| File or Folder | Generated by OpenSSL? | Description |
|---|---|---|
| openssl-ca\ | | |
| openssl-ca\openssl.cfg | | OpenSSL configuration file |

| File or Folder | Generated by OpenSSL? | Description |
|---|---|---|
| openssl-ca\.rnd | Yes | File filled with random data, used in key generation process. |
| openssl-ca\ca-password.txt | | Stores the password for the CA private key.<br>Reduces typing required, but is very insecure. Should only be used for testing and development. |
| openssl-ca\export-password.txt | | Stores the password used to encrypt the private keys when exporting PKCS#12 files.<br>Reduces typing required, but is very insecure. Should only be used for testing and development. |
| openssl-ca\ca\ | | CA root folder. |
| openssl-ca\ca\certs\ | | All generated certificates are copied here.<br>Folder contents can be safely deleted. |
| openssl-ca\ca\crl\ | | Generated CRLs stored here.<br>Folder contents can be safely deleted. |
| openssl-ca\ca\newcerts\ | | Certificates being generated are stored here.<br>Folder contents can be safely deleted *once generation process is finished*. |
| openssl-ca\ca\private\ | | CA private files. |
| openssl-ca\ca\private\cakey.pem | Yes | CA private key.<br>Must be kept secret. |
| openssl-ca\ca\crlnumber | | Serial number of last exported CRL. |
| openssl-ca\ca\serial | | Serial number of last signed certificate. |
| openssl-ca\ca\cacert.pem | Yes | CA certificate. |
| openssl-ca\ca\index.txt | | Textual database of all certificates. |

## Short Command Line Reference

- This section assumes that the OpenSSL *bin* folder was added to the local PATH environment variable, and that *openssl-ca* is the current folder for all issued commands.
- Placeholders for parameters are shown in the following form: "<param-placeholder>".
- The frequently used parameter "<request-name>" should be a unique name that identifies the certificate files.

| Task | Description | Command |
|---|---|---|
| Create a CA Certificate/Key | This is performed in three steps:<br>1. Create CA Private Key<br>2. Create CA Certificate<br>3. Export CA Certificate | 1. `openssl genrsa -des3 -out ca\private\cakey.pem 1024 -passin file:ca-password.txt`<br>2. `openssl req -config openssl.cfg -new -x509 -days <days-ca-cert-is-valid> -key ca\private\cakey.pem -out ca\cacert.pem -passin file:ca-password.txt`<br>3. `openssl x509 -in ca\cacert.pem -outform PEM -out ca.pem` |
| Create a Leaf Certificate/Key Pair | This is performed in three steps:<br>1. Create certificate request. Certificate fields and extensions are defined during this step, and the certificate's public and private keys are created in the process.<br>2. Sign the request.<br>3. Export the certificate. | 1. `openssl req -new -nodes -out requests\<request name>-req.pem -keyout requests\<request name>-key.pem -days 3650 -config openssl.cfg`<br>2. `openssl ca -out requests\<request-name>-signed.pem -days 3650 -config openssl.cfg -passin file:ca-password.txt -infiles requests\<request-name>-req.pem`<br>3. `openssl pkcs12 -export -in requests\<request-name>-signed.pem -inkey requests\<request-name>-key.pem -certfile ca\cacert.pem -name "<entry-name-in-p12-file>" -out <request-name>.p12 -passout file:export-password.txt`<br>`openssl x509 -in requests\<request-name>-signed.pem -outform PEM -out <request-name>-cert.pem`<br>`openssl pkcs8 -topk8 -nocrypt -in requests\<request-name>-key.pem -out <request-name>-key.pem` |

| Task | Description | Command |
|---|---|---|
| Revoke a Certificate | | `openssl ca -revoke <certificate-pem-file> -config openssl.cfg -passin file:ca-password.txt` |
| Export the CRL | | `openssl ca -gencrl -crldays <days-crl-is-valid> -out crl.pem -config openssl.cfg -passin file:ca-password.txt` |

## MSCAPI Provider: Windows Certificate Services

**Note:** Working with Windows Certificate Services (WCS) is also covered in *Genesys 8.1 Security Deployment Guide*.

MSCAPI stands for Microsoft CryptoAPI. This provider offers the following features:

- It is available only on Windows platform.
- It implies usage of WCS to store and retrieve certificates, private keys, and CA certificates.
- Every Windows account has its own WCS storage, including the System account.
- Depends heavily on OS configuration and system security policies.
- Has its own set of supported cipher suites, different from what is provided by Java.
- When used with Java, please use the latest available version of Java to run the application.
- Java does not support CRLs located in WCS. With Java MSCAPI, CRL should be specified as a file.
- Does not accept passwords from Java code programmatically via CallbackHandler. If private key is password-protected or prompt-protected, OS popup dialog will be shown to user.
- Certificates in WCS are configured using the Certificates snap-in for Microsoft Management Console (MMC).

**Note:** If the version of Java being used does not support MSCAPI, a "WINDOWS-MY KeyStore not available" exception appears in the application log. If you receive such exceptions, please consider switching to a newer version of Java.

### Starting Certificates Snap-in

There are two methods for accessing the Certificates Snap-in:

- Enter "certmgr.msc" at the command line. (This only gives access to Certificates for the current user account.)
- Launch the MMC console and add the Certificates Snap-in for a specific account using the following steps:
  1. Enter "mmc" at the command line.
  2. Select *File* > *Add/Remove Snap-in...* from the main menu.

3. Select *Certificates* from the list of available snap-ins and click *Add*.
4. Select the account to manage certificates for (see Account Selection for important notes) and click *Finish*.
5. Click *OK*.

### Account Selection

It is important to place certificates under the correct Windows account. Some applications are run as services under the Local Service or System account, while others are run under user accounts. The account chosen in MMC must be the same as the account used by the application that certificates are configured for, otherwise the application will not be able to access this WCS storage.

**Note:** Currently, most Genesys servers do not clearly report this error so WCS configuration must be checked every time there is a problem with the MSCAPI provider.

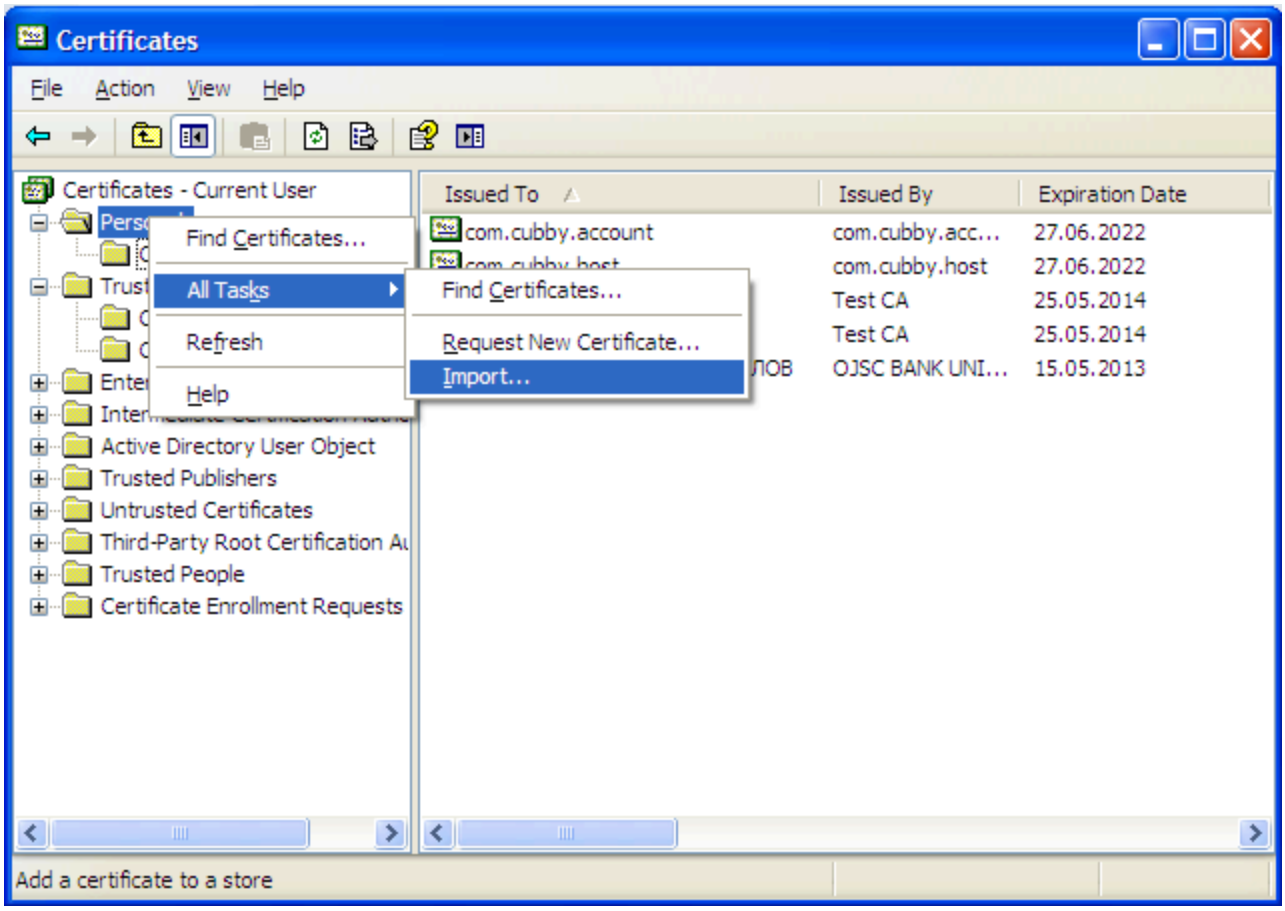
**Note:** Configuration Manager is also a regular application in this aspect and can access WCS only for the Local Computer (System) account on the local machine. It will not show certificates configured for different accounts or on remote machines. Please consult your system and/or security administrator for questions related to certificate configuration and usage.

## Importing Certificates

There are many folders within WCS where certificates can be placed. Only two of them are used by Platform SDK:

- Personal/Certificates – Contains application certificates used by applications to identify themselves.
- Trusted Root Certification Authorities/Certificates – Contains CA certificates used to validate remote party certificates.

To import a certificate, right-click the appropriate folder and choose *All Tasks > Import...* from the context menu. Follow the steps presented by the Certificate Import Wizard, and once finished the imported certificate will appear in the certificates list.




Although WCS can import X.509 PEM certificate files, these certificates cannot be used as application certificates because they do not contain a private key. It is not possible to attach a private key from a PKCS#7 PEM file to the imported certificate. To avoid this problem, import application certificates only from PKCS#12 files (*.*p12*) which contain a certificate and private key pair.

CA certificates do not have private keys attached, so it is safe to import CA certificates from X.509 PEM files.

It is possible to copy and paste certificates between folders and/or user accounts in the Management Console, but this approach is not recommended due to WCS errors which may result in the pasted certificate having an inaccessible private key. This error is not visible in Console, but applications would not be able to read the private key. A recommended and reliable workaround is to export the certificate to a file and then import from that file.

If you encounter the following error in the application log: “The credentials supplied to the package were not recognized”, the most likely cause is due to the private key being absent or inaccessible. In this case try deleting the certificate from WCS and re-importing it.

## Importing CRL Files

CRL files can be imported to the following folder in WCS:

- Trusted Root Certification Authorities/Certificate Revocation List

The import procedure is the same as for importing certificate. CRL file types are automatically recognized by the import wizard.

**Note:** Although an MSCAPI provider may choose to use CRL while validating remote party certificates, this functionality is not guaranteed and/or supported by Platform SDK. Platform SDK implements its own CRL matching logic using CRL PEM files.

## PKCS11 Provider: Mozilla NSS

PKCS11 stands for the PKCS#11 family of Public-Key Cryptography Standards (PKCS), published by RSA Laboratories. These standards define platform-independent API-to-cryptographic tokens, such as Hardware Security Modules (HSM) and smart cards, allowing you to connect to external certificate storage devices and/or cryptographic engines.

In Java, the PKCS#11 interface is a simple pass-through and all processing is done externally. When used together with a FIPS-certified security provider, such as Mozilla NSS, the whole provider chain is FIPS-compliant.

Platform SDK uses PKCS11 because it is the only way to achieve FIPS-140 compliance with Java.

### Installing Mozilla NSS

Currently Platform SDK only supports FIPS when used with the Mozilla NSS security provider. (Java has FIPS certification only when working with a PKCS#11-compatible pluggable security provider, and the only provider with FIPS certification and Java support is Mozilla NSS.)

**Note:** In theory, BSafe can be used since it supports JCA interfaces. However, Platform SDK was not tested with RSA BSafe and such system would not be FIPS-certifiable as a while.

Generally, some security parameters and data must be configured on client host, requiring the involvement of a system/security administrator. At minimum, the client host must have a copy of the CA Certificate to be able to validate the Configuration Server certificate. The exact location of the CA certificate depends on the security provider being used. It can be present as a PEM file, Java Keystore file, a record in WCS, or as an entry in the Mozilla NSS database. Once the application is connected to Configuration Server, the Application Template Application Block can be used to extract connection parameters from Configuration Server and set up TLS.

Mozilla NSS is the most complex security provider to deploy and configure. In order to use NSS, the following steps must be completed:

1. Deploy Mozilla NSS.
2. Create Mozilla NSS database (a "soft token" in terms of NSS), and set it to FIPS mode.
3. Adjust the Java security configuration, or implement dynamic loading for the Mozilla NSS provider.
4. Import the CA certificate to the Mozilla NSS database.
5. Use the Platform SDK interface to select PKCS11 as a provider (with no specific configuration options required).

## Configuring FIPS Mode in Mozilla NSS

To configure FIPS mode in Mozilla NSS, create a file named *nss-client.cfg* in Mozilla NSS deployment folder with the following values configured:

- name - Name of a software token.
- nssLibraryDirectory - Library directory, located in the Mozilla NSS deployment folder.
- nssSecmodDirectory - Folder where the Mozilla NSS database for the listed software token is located.
- nssModule - Indicates that FIPS mode should be used.

An example is provided below:

```
name = NSSfips
nssLibraryDirectory = C:/nss-3.12.4/lib
nssSecmodDirectory = C:/nss-3.12.4/client
nssModule = fips
```

More information about configuring FIPS mode can be found using external resources:

- https://davidvaleri.wordpress.com/2010/10/19/using-nss-for-fips-140-2-compliant-transport-security-in-cxf/ external sources

## Configuring FIPS Mode in Java Runtime Environment (JRE)

To configure your Java runtime to use Mozilla NSS, the *java.security* file should be located in Java deployment folder and edited as shown below:

(Changes are shown in **bold red**, insertions are shown in **bold blue**)

```
#
# List of providers and their preference orders (see above):
#
security.provider.1=sun.security.provider.Sun
security.provider.2=sun.security.rsa.SunRsaSign
security.provider.3=sun.security.ec.SunEC
#security.provider.4=com.sun.net.ssl.internal.ssl.Provider
security.provider.4=com.sun.net.ssl.internal.ssl.Provider SunPKCS11-NSSfips
security.provider.5=com.sun.crypto.provider.SunJCE
security.provider.6=sun.security.jgss.SunProvider
security.provider.7=com.sun.security.sasl.Provider
security.provider.8=org.jcp.xml.dsig.internal.dom.XMLDSigRI
security.provider.9=sun.security.smartcardio.SunPCSC
security.provider.10=sun.security.mscapi.SunMSCAPI
security.provider.11=sun.security.pkcs11.SunPKCS11 C:/nss-3.12.4/nss-client.cfg
```

After those updates are complete, the Java runtime instance works with FIPS mode, with only the PKCS#11/Mozilla NSS security provider enabled.

## Short Command Line Reference

Please refer to the following reference for more information:

- https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/Tools

| Task | Command |
|---|---|
| Create CA Certificate | `certutil -S -k rsa -n "<CA-cert-name>" -s "CN=Test CA, OU=Miratech, O=Genesys, L=Kyiv, C=UA" -x -t "CTu,u,u" -m 600 -v 24 -d ./client -f "<keystore-password-file>"` |
| Import CA Certificate | `certutil -A -a -n “<CA-cert-name>” -t "CTu,u,u" -i <ca-cert-file> -d ./client -f "<keystore-password-file>"` |
| Create New Leaf Certificate | `certutil -S -k rsa -n "<cert-name>" -s "CN=Test CA, OU=Miratech, O=Genesys, L=Kyiv, C=UA" -x -t "u,u,u" -m 666 -v 24 -d ./client -f "<keystore-password-file>" -z "<noise-file>"` |
| Import Leaf Certificate | `pk12util -i <cert-file.p12> -n <cert-name> -d ./client -v -h "NSS FIPS 140-2 Certificate DB" -K <keystore-password>` |
| Create CRL | `crlutil -d ./client -f "<keystore-password-file>" -G -c "<crl-script-file>" -n "<CA-cert-name>" -l SHA512` |
| Modify CRL | `crlutil -d ./client -f "<keystore-password-file>" -M -c "<crl-script-file>" -n "<CA-cert-name>" -l SHA512 -B` |
| Show Certificate Information | `certutil -d ./client -f "<keystore-password-file>" -L -n "<cert-name>"` |
| Show CRL Information | `crlutil -d ./client -f "<keystore-password-file>" -L -n "<CA-cert-name>"` |
| List Certificates | `certutil -d ./client –L` |
| List CRLs | `crlutil -L -d ./client` |

## JKS Provider: Java Built-in

This provider is supported by the Platform SDK Commons library, but the Application Template Application Block does not support this provider due to compatibility guidelines with Genesys Framework Deployment.

This provider can only be used when TLS is configured programmatically by Platform SDK users.

### Short Command Line Reference

Refer to the following references for more information:

- https://docs.oracle.com/javase/8/docs/technotes/tools/unix/keytool.html
- https://docs.oracle.com/javase/8/docs/technotes/tools/windows/keytool.html

| Task | Command |
|---|---|
| **Creating and Importing** - These commands allow you to generate a new Java Keytool keystore file, | |

| Task | Command |
|---|---|
| create a Certificate Signing Request (CSR), and import certificates. Any root or intermediate certificates will need to be imported before importing the primary certificate for your domain. | |
| Generate a Java keystore and key pair | `keytool -genkey -alias mydomain -keyalg RSA -keystore keystore.jks -keysize 2048` |
| Generate a certificate signing request (CSR) for an existing Java keystore | `keytool -certreq -alias mydomain -keystore keystore.jks -file mydomain.csr` |
| Import a root or intermediate CA certificate to an existing Java keystore | `keytool -import -trustcacerts -alias root -file Thawte.crt -keystore keystore.jks` |
| Import a signed primary certificate to an existing Java keystore | `keytool -import -trustcacerts -alias mydomain -file mydomain.crt -keystore keystore.jks` |
| Generate a keystore and self-signed certificate | `keytool -genkey -keyalg RSA -alias selfsigned -keystore keystore.jks -storepass password -validity 360 -keysize 2048` |
| **Java Keytool Commands for Checking** - If you need to check the information within a certificate, or Java keystore, use these commands. | |
| Check a stand-alone certificate | `keytool -printcert -v -file mydomain.crt` |
| Check which certificates are in a Java keystore | `keytool -list -v -keystore keystore.jks` |
| Check a particular keystore entry using an alias | `keytool -list -v -keystore keystore.jks -alias mydomain` |
| **Other Java Keytool Commands** | |
| Delete a certificate from a Java Keytool keystore | `keytool -delete -alias mydomain -keystore keystore.jks` |
| Change a Java keystore password | `keytool -storepasswd -new new_storepass -keystore keystore.jks` |
| Export a certificate from a keystore | `keytool -export -alias mydomain -file mydomain.crt -keystore keystore.jks` |
| List Trusted CA Certs | `keytool -list -v -keystore $JAVA_HOME/jre/lib/security/cacerts` |
| Import New CA into Trusted Certs | `keytool -import -trustcacerts -file /path/to/ca/ca.pem -alias CA_ALIAS -keystore $JAVA_HOME/jre/lib/security/cacerts` |