Platform SDK Developer's Guide

Secure connections using TLS

12/12/2025

Contents

Secure connections using TLS

This page provides an introduction to creating and configuring Transport Layer Security (TLS) for your Platform SDK connections, as introduced in release 8.1.1.

Introduction to TLS

This page provides an overview of the TLS implementation provided in the 8.1.1 release of Platform SDK. It introduces Platform SDK users to TLS concepts and then provides links to expanded articles and examples that describe implementation details.

Before working with TLS to create secure connections, you should have a basic awareness of how public key cryptography works.

Certificates

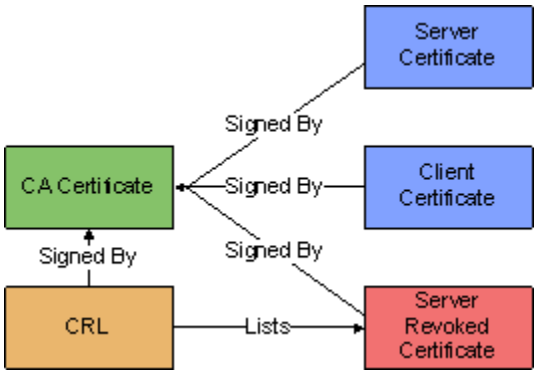
Transport Layer Security (TLS) technology uses public key cryptography, where the key required to encrypt and decrypt information is divided into two parts: a public key and a private key. These parts are reciprocal in the sense that data encrypted using a private key can be decrypted with the public key and vice versa, but cannot be decrypted using the same key that was used for encryption.

There is an X.509 standard for public key (certificate) format, and public-key cryptography standards (PKCS) that define format for private key (PKCS#8) and related data structures.

Certificate Authority (CA)

In the context of TLS, a CA is an entity that is trusted by both sides of network connection. Each CA has a public X.509 certificate and owns a related private key that kept secret. A CA can generate and sign certificates for other parties using its private key, and then that CA certificate can be used by the parties to validate their certificates. A CA can also issue public Certificate Revocation Lists (CRLs), which are also used by parties for certificate validation.

The relation between certificates and CRL can be depicted like this:



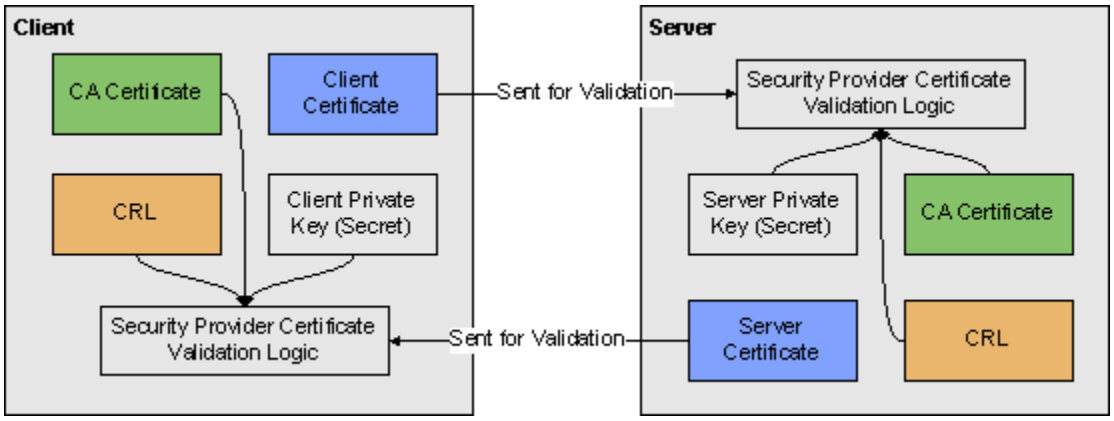
Certificate Usage

To create a secure connection, each party must have a copy of:

- a public CA certificate
- a CRL issued by the CA
- their own public certificate (with a corresponding private key)

When a network connection is established, the client initiates a TLS handshake process during which the parties exchange their public certificates, prove that they own corresponding private keys, create a shared session encryption key, and negotiate which cipher suite will be used.

Placement and exchange of certificate data is shown on the following diagram:



TLS only requires that servers send their certificates, but the client certificates can also be exchanged depending on server settings. Cases where the client certificates are demanded by the server are called “Mutual TLS”, as both sides send their certificates.

If all certificates pass validation and the ciphers are negotiated successfully, then a TLS connection is established and higher-level protocols may proceed.

Implementing and Configuring TLS

Genesys strongly recommends reading all TLS in Platform SDK articles in order to get understanding of how TLS works in general and how it is supported in Platform SDK. A Quick Start page is provided for reference, but the specific implementation details and expanded information provided in other pages will help you to better understand how to provide TLS support in your applications. Once you have an understanding of how TLS is implemented, you can use the Use Case guide to quickly find code snippets or relevant links for common tasks.

There are two main ways to implement TLS in your Platform SDK code:

1. Use the Platform SDK Commons Library to specify TLS settings directly when creating endpoints
2. Use the Application Template Application Block to read connection parameters inside configuration

objects retrieved from Configuration Server, then use those parameters to configure TLS settings.

Note: If using the Application Template Application Block, you will need to configure TLS Parameters in Configuration Manager before the application is tested.

Recommendations are also provided for the configuration and use of security providers. The security providers discussed on that page have been tested within the described configurations, and worked reliably.

Migrating TLS Support From Previous Versions of Platform SDK

Platform SDK for Java

Platform SDK 8.1.0 had the following connection configuration parameters for TLS:

- *Connection.TLS_KEY*
- *Connection.SSL_KEYSTORE_PATH_KEY*
- *Connection.SSL_KEYSTORE_PASS*

The *TLS_KEY* parameter is the equivalent of *enableTls* flag in the current release, while the other parameters specified the location and password for the Java keystore file containing certificates that were used by the application to authenticate itself. TLS configuration code looked like this:

```
ConnectionConfiguration connConf = new KeyValueConfiguration(new KeyValueCollection());
connConf.setOption(Connection.TLS_KEY, "1");
connConf.setOption(Connection.SSL_KEYSTORE_PATH_KEY, "c:/certificates/client-certs.keystore");
connConf.setOption(Connection.SSL_KEYSTORE_PASS, "pa$$w0rd");
```

In Platform SDK 8.1.1, this code can be translated to the following:

```
boolean tlsEnabled = true;
// By default, PSDK 8.1.0 trusted any certificate
TrustManager trustManager = TrustManagerHelper.createTrustEveryoneTrustManager();
// Keystore entries may be protected with individual password,
// but usually, these passwords are the same as keystore password
KeyManager keyManager = KeyManagerHelper.createJKSKeyManager(
    "c:/certificates/client-certs.keystore", "pa$$w0rd", "pa$$w0rd");
SSLContext sslContext = SSLContextHelper.createSSLContext(keyManager,
trustManager);
```

In most cases, certificates from other parties will need to be validated. Assuming there is a separate keystore file with a CA certificate, this can be achieved with the following code:

```
TrustManager trustManager = TrustManagerHelper.createJKSTrustManager(
    "c:/certificates/CA-cert.keystore", "pa$$w0rd", null, null);
```

Please note that different keystore files are used for the *KeyManager* and *TrustManager* objects. For more information, see Using the Platform SDK Commons Library.

Platform SDK for .Net

There were no significant changes to interfaces for the .NET version of Platform SDK 8.1.1. In this

case, the same code would work for 8.1.0 and 8.1.1 releases:

```
KeyValueConfiguration config = new KeyValueConfiguration(new KeyValueCollection());
config.TLSEnabled = true;
config.TlsCertificate = "29 3f 0d d9 65 a1 a9 92 dd 1c 8c 2a e7 20 74 06 c5 ba 0f 10";
Endpoint ep = new Endpoint(AppName, Host, Port, config);
```

Known Issues

For more details about the known issues listed here, refer to Using and Configuring Security Providers.

- Java 5: MSCAPI provider is not supported.
- Java 6:
  - MSCAPI provider is only supported in 32-bit version since update 27: http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6931562.
  - MSCAPI provider is only supported in 64-bit version since update 38: http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=2215540.
  - CRLs located in WCS are ignored, please use CRLs as files.
- Java 7:
  - CRL files without extension section cannot be loaded: http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=7166885.
    Note: Although the bug is marked as "Will not fix", it seems to be fixed since Java 7 update 7.
  - CRLs located in WCS are ignored, please use CRLs as files.
- MSCAPI: MSCAPI does not have a documented way of programmatic setting of password to private key stored in WCS. Regardless of password returned by CallbackHandler; if private key is protected with confirmation prompt or password prompt, user will be shown OS popup dialog.