Secure sockets communications in C and Java

1. Public key certificates based mutual authentication

By using SSL/TLS it is very simple to implement a very solid mutual authentication schema between two network applications communication through an insecure network (e.g., the Internet). With two enrolled applications **A** and **B**, the goal is assuring application **A** will interact only with the authentic application **B**, and application **B** will interact only with the authentic application **A**.

The suggested authentication is based on public keys, if data is encrypted with the public key of application **B**, then only application **B** will be able do decrypt it, likewise, if data is encrypted with the public key of application **A**, then only application **A** will be able to decrypt it. This authentication schema is used by SSL/TLS, during the SSL/TLS handshake each application sends to the counterpart its own public key certificate, which contains its own public key.

When one application receives a public key certificate from a remote application, it must be checked in several ways to see if it is trusted, including checking if it was issued by a trusted CA (Certification Authority). If trusted, this means it really belongs to the intended authentic remote application, and thus when data is encrypted with that public key, only the intended authentic remote application will be able to decrypt it.

In our case we can skip most of the steps in checking the public key certificate because we will simply pick the application **A** public key certificate and manually add it to application **B** as being a trusted certificate, likewise, we will pick the application **B** public key certificate and manually add it to application **A** as being a trusted certificate. By doing this we establish a trusting relationship between these two applications.

2. Practical implementation notes

2.1. Client application or server application

For mutual authentication, both applications A, and B must have a public key certificate. Under SSL/TLS public key authentication, only the server is obliged to send its public key certificate, however, on the server side we may configure the handshake to demand a public key certificate from the client.

Therefore, to implement mutual authentication, on the server side we must use the following code:

- In C/OpenSSL: **SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER|SSL_VERIFY_FAIL_IF_NO_PEER_CERT, NULL);**
- In Java: **setNeedClientAuth(true);**

2.2. Generating and using the public key certificates

Each application will have its own public key certificate and corresponding private key, it will also have the public key certificate of the remote trusted application.

To generate and manage public key certificates and private keys we are going to use:

- In C/OpenSSL: the **openssl** command
- In Java: the **keytool** command

Let's take for instance a scenario where we have a network application **A** developed in C/OpenSSL and a network application **B** developed in Java, to create the corresponding self-signed public key certificates and corresponding private keys we could use the following command lines:

```
openssl req -nodes -x509 -keyout A.key -out A.pem -days 365 -newkey rsa:2048

keytool -genkey -v -alias B -keyalg RSA -keysize 2048 -validity 365 -keystore B.jks –storepass secret
```

The following files are created:

A.key – application A private key, **must be kept secret at all costs**.

A.pem – application A public key certificate.

B.jks – Java Key Store containing the application B private key and the public key certificate.

The Java Key Store is protected by a password (secret), this is mandatory. Notice that the encryption algorithm used to protect the Java Key Store may be different depending on the Java version, be sure you have created it with the keytool command of the same Java version.

2.3. Setting the application's own public key certificate

Each application must load its own public key certificate and corresponding private key, this public key certificate will be provided to the remote application during the SSL/TLS handshake.

For the **C/OpenSSL** application **A**, in the earlier described scenario, they are loaded from different files:

```
SSL_CTX_use_certificate_file(…, "A.pem" , …)

SSL_CTX_use_PrivateKey_file(…, "A.key", … )
```

For the **Java** application **B**, they are loaded from the Java Key Store:

```
System.setProperty("javax.net.ssl.keyStore", "B.jks");

System.setProperty("javax.net.ssl.keyStorePassword","secret");
```

2.4. Setting the trusting relationship

The trusting relationship is established by mutual authentication based on the counterpart's public key certificate:

- In application A, the public key certificate of application B must be trusted.
- In application B, the public key certificate of application A must be trusted.

2.4.1. For application A (C/OpenSSL)

The public key certificate of application **B** is in the Java Key Store at file **B.jks**, but it may be exported to PEM format by using the keytool command:

```
keytool -exportcert -alias B -keystore B.jks -storepass secret -rfc -file B.pem
```

This file **B.pem** must now be provided to application **A** as a trusted certificate location to be loaded:

```
SSL_CTX_load_verify_locations(ctx, ”B.pem”, NULL);
```

2.4.2. For application B (Java)

The public key certificate of application **A** is in the **A.pem** file, and it must be provided to application **B** as a trusted certificate, for that it must be in a Java Key Store, we could create another one, but we can as well use **B.jks** and import to it the certificate:

```
keytool -import -alias A -keystore B.jks -file A.pem -storepass secret
```

Now, on application **B** we must set **B.jks** as a trust store, meaning public key certificate in there are to be trusted:

```
System.setProperty("javax.net.ssl.trustStore", “B.jks”);
System.setProperty("javax.net.ssl.trustStorePassword", “secret”);
```