Setting up Apache Tomcat and a Simple Apache SOAP Client for SSL Communication.

By Peter Glynn and Darrell Drake, with minor updates by Jonathan Chawke, April 2001.

Introduction

This document gives steps involved in setting up Apache Tomcat and a simple Apache SOAP client for SSL communication.
The aim of this document is to allow a person with minimum Java security to be able to set up SSL connection in a Apache SOAP/Tomcat Application. The steps you will carry out are:

1. Install the Java Secure Socket Extensions (JSSE) package (available from Sun).
2. Generate Client and Server Certificates for SSL communication.
   a. Generate a Server Key and Certificate
   b. Export the Server Certificate
   c. Generate a Client Key and Certificate
   d. Export the Client Certificate
   e. Import the Certificates into the Keystores
3. Set up Tomcat for SSL Communication
4. Modify the SOAP Client to use SSL

Also included in this document is:

- A short tutorial on creating X.509 Certificate Chains
- A section on troubleshooting SOAP and SSL installations

Assumptions

It is assumed that you have installed Apache SOAP and Apache Tomcat, and that the sample SOAP applications are working.

Tools needed for Installation

- Apache SOAP – download at http://xml.apache.org/dist/soap/
- Java (tm) Secure Socket Extension (JSSE) 1.0.2 (jsse1.0.2) – download at http://java.sun.com/products/jsse/ (free registration required)

Step 1: Install JSSE

- Before you do anything, **read** the installation instructions for JSSE! (they are available online at: http://java.sun.com/products/jsse/install.html).
- Add the JSSE jars to your classpath. This should hopefully add it to the classpath of your Tomcat server. If not, add it to the classpath of the Tomcat server or just copy the JSSE jar files to the lib directory of Tomcat (e.g. `C:\jakarta-tomcat-3.2.1\lib`). This will automatically load on start-up of Tomcat.
  The JSSE 1.0.2 jars that you need in your classpath are:
  - `jsse.jar`
  - `jcert.jar`
  - `jnet.jar`

Step 2: Generate Client and Server Certificates

It is necessary to generate a Certificate for the client and the server. These Certificates are then imported into a keystore, to which the client and server connect.
The keystore acts as a database for security certificates.
You are going to use the `keytool` utility in the JDK to do these tasks (see Sun's documentation for more information on this tool).

Step 2a: Generate a Server Key and Certificate

Launch `keytool` from a shell (or command prompt) to generate your public and private key.
Note that the Certificate and keystore files will be generated in the directory you run `keytool` from.

Use `keytool` as follows:
```
keytool -genkey -alias tomcat-sv -dname "CN=[Common Name],OU=[Organisation Unit],
O=[Organisation Name], L=[Locality], S=[State Name], C=[Two-Letter Country Code]" -
keyalg RSA -keypass [private key password] -storepass [keystore password] -keystore
[keystore file name]
```

For example, to generate a keystore (in file `server.keystore`) for server `soapsvr.test.tcd.ie` using password `changeit` (for both the keystore and the certificate) in the Computer Engineering group at Trinity College Dublin, Ireland, one would type the following: `keytool -genkey -alias tomcat-sv -dname "CN=soapsvr.test.tcd.ie, OU=ComputerEngineering, O=Trinity College Dublin, L=Dublin, S=Dublin, C=IE" -keyalg RSA -keypass changeit -storepass changeit -keystore server.keystore`

Note that

- The RSA algorithm is used to generate certificates.
- Ensure that the 'CN' field that you specify when you create the server certificate matches the name of the machine on which you're running tomcat, or your browser will complain about certificate name mis-matches (not a problem on a test server, a big problem on a production server!).

Step 2b: Export the Server Certificate

>From command prompt run this command to export your certificate from the keystore into an external file (we do this so we can import the certificate into the client's keystore as a trusted certificate).

```
keytool -export -alias tomcat-sv -storepass changeit -file server.cer -keystore server.keysto
```

If everything works, you should now have a file called `server.cer` which contains your server's certificate.

Step 2c: Generate a Client Key and Certificate

This step is very similar to the generation of the server key and certificate - it uses the same `keytool` tool with different parameters.
Note that the keystore file name has changed (it is now `client.keystore`). Use `keytool` as follows:

```
keytool -genkey -alias tomcat-cl -dname "CN=Client,OU=TRL, O=IBM, L=Yamato-shi,
S=Kanagawa-ken, C=JP" -keyalg RSA -keypass changeit -storepass changeit -keystore
```

```
client.keystore
```

Step 2d: Export the Client Certificate

This step is very similar to the export of the server certificate - it uses the same `keytool` tool with different parameters:

```
keytool -export -alias tomcat-cl -storepass changeit -file client.cer -keystore client.keysto
```

If everything works, you should now have a file called `client.cer` which contains your client's certificate.

Step 2e: Import the Certificates into the Keystores

We want the client certificate to be added to the server's keystore, and the server's certificate to be added to the client's keystore.
Doing this will mean that the client and server trust one another.
Import the server certificate into the client's keystore:
`keytool -import -v -trustcacerts -alias tomcat -file server.cer -keystore client.keystore -keypass changeit -storepass changeit`
Import the client certificate into the server's keystore: `keytool -import -v -trustcacerts -alias tomcat -file client.cer -keystore server.keystore -keypass changeit -storepass changeit`

Step 3: Set up Tomcat for SSL Communication

Step 3a: Modify your Tomcat Configuration File

You need to amend server.xml (located in the `conf` directory of Apache Tomcat). Add the following lines to the xml file:

```
<Connector className ="org.apache.tomcat.service.PoolTcpConnector">

<Parameter name="handler" value ="org.apache.tomcat.service.http.HttpConnectionHandler"/>

<Parameter name="port" value="8443"/>

<Parameter name="socketFactory" value="org.apache.tomcat.net.SSLSocketFactory" />

<Parameter name="keystore" value="c:\apache\soap-2_3\bin\server.keystore" />

<Parameter name="keypass" value="changeit"/>

<Parameter name="clientAuth" value="true"/>

</Connector>
```

Note that the value used for the `keystore` parameter (shown in bold above) may be different on your machine; it should contain the full path and filename of the server keystore file (`server.keystore`) generated in Step 2a above. Note also that the port number we chose to use for SSL in the above configuration is 8443. The port normally used for HTTPS is 443, but for testing we are using 8443.

Step 3a: Test your HTTPS server

At this point, you should restart your Tomcat server. It will probably take a bit longer to start up. You can now use a web browser to test that it is working.
Test the SSL installation by opening your browser and typing in the following URL:

```
        https://servername:8443/index.html
```

Note that `servername` should be replaced with the name of the server on which you are running Tomcat.
If SSL is working then you should see the default home page for your Tomcat installation.
Your browser may generate a warning about un-trusted certificates or unrecognised authorities (just click OK).

Step 4: Modify the SOAP Client to use SSL

Step 4a: Java SSL Client

You need to set up properties before you call the URL in the SOAP client. Here is an example SOAP client that calls a SOAP service using HTTPS on a Tomcat server:

```
// classes for ssl

import javax.net.ssl.SSLSocketFactory;

import java.security.Security;



...



        //

        // setup some ssl-specific stuff

        //


        // specify the location of where to find key material for the default TrustManager (tl

        System.setProperty("javax.net.ssl.trustStore","C:\\jdk1.3\\bin\\client.keystore");

        // use Sun's reference implementation of a URL handler for the "https" URL protocol t

        System.setProperty("java.protocol.handler.pkgs","com.sun.net.ssl.internal.www.protoco

        // dynamically register sun's ssl provider

        Security.addProvider(new com.sun.net.ssl.internal.ssl.Provider());

        // note that the url is using https protocol and not http

        URL urls = new URL( "https://localhost:8443/soap/servlet/rpcrouter");


        //

        // prepare and then execute a SOAP method

        //
```

```
// output some basic information

System.out.println("\nUsing " + urls.getProtocol() + " to connect to " + urls.getHost


// prepare the service invocation as usual

Call call = new Call();

String urn = "urn:demo:checkflight";

call.setTargetObjectURI( urn );

call.setMethodName( "getFlightInfo" );


// set up any parameters as usual

...


// Invoke the call

Response resp;

try

{

  resp = call.invoke(urls, "");

}

catch (SOAPException e)

{

  System.err.println("Caught SOAPException (" + e.getFaultCode() + "): " + e.getMessa

  e.printStackTrace();

  return;

}
```

Once again the bold directory path is a pointer to the client keystore. This may have to be changed depending on where you generated it. Also note that the url is **https** and not http. It's an easy mistake to make!

Step 4b: Java SSL Client with Proxy [Optional]

If your client needs to use a proxy server in order to access the SOAP service, then add the following lines to your code:

```
System.setProperty("https.proxyHost", "proxy");  // set name of proxy server that sup
```

```
        System.setProperty("https.proxyPort", "8080");   // set port number for proxy server
```

Use the following for Proxy without SSL:

```
        System.setProperty("proxySet", "true"); // enable proxying
        System.setProperty("proxyHost", "proxy");       // set name of proxy server
        System.setProperty("proxyPort", "8080");        // set port number for proxy server
```

If you are using Socks proxy then set these (system) properties:

```
        System.setProperty("socksProxyHost", "hostname");       // set name of socks server
        System.setProperty("socksProxyPort", "1080");           // set port number for socks
```

Creating X.509 Certificate Chains

By Darrell Drake

Overview

This instruction set covers creating mutual trust between two entities using a certificate chains (my certificate with an attached lineage of public certificates), which is more practical in a production environment than the minimal security approach. In this case a principal is trusted if any of the certificates in its chain are trusted. Additional tool needed: IBM KeyMan, which can read and issue certificates from keystores in the JKS and PKCS12 formats. Other formats (e.g. IBM CMS Key Database) don't work well, at least in my experience.

Downloading KeyMan

KeyMan can be downloaded from: http://www.alphaworks.ibm.com/tech/keyman.

Creating a new Keystore and Keypair using KeyMan

- From the initial window, click on the "create new" icon on the left
  OR
  from an already-open KeyMan window, under the File menu choose New.
- Specify the type of keystore (pkcs 7, 12 and JKS are options)
- Immediately the template is created
- Under the Actions menu choose generate key
- Select the desired key algorithm and length, click OK
- WAIT

Opening an existing keystore file

- From the initial window, click on the "open" icon on the right
  OR
  from an already-open KeyMan window, under the File menu choose Open
- Select "local resource", click next

- Enter or browse for the filename, click next
- If applicable, enter the password protecting the file

Generating a self-signed certificate using KeyMan (for new serverkeystore)

- Make sure the new key pair (and just the key pair) is selected
- Under the Actions menu, choose Create Certificate
- Choose Self-Signed Certificate, click next
- Fill in the principal information as requested, click next
- Select the period of validity for the new certificate, click next

Requesting a client certificate using keytool

```
-- using keytool (after creating keyEntry "alias")
keytool -certreq {-alias alias} {-sigalg sigalg} {-file certreq_file} [-keypass
keypass] {-storetype storetype} {-keystore keystore} [-storepass storepass] {-v} {-
Jjavaoption}
```

If you leave the "`-file certreq_file`" part out, the pkcs10 request will be printed to your standard output, which you can highlight and copy to the system clipboard, transfer to a file or directly into KeyMan (see next step "issuing certificate"). The principal information in the request will be what you entered when using the "-genkey" command earlier.

Requesting a client certificate using KeyMan (after creating a key pair)

- Make sure the new key pair (and just the key pair) is selected
- Under the Actions menu, choose Request Certificate
- Choose "Generate a PKCS#10 Request", click next
  OR
  If you want to get a commercial certificate for your server, choose "Go online to a CA" (I don't know the remaining steps for doing that)
- Fill in the principal information as requested, click next
- Enter the filename where you want to save the request,
  OR
  Choose to copy the request to the system clipboard, click next
- The request is now stored in the location that you specified.

Issuing a client certificate using KeyMan

(aka acting as your own CA, signing a PKCS#10 certificate request)

- Open the keystore that contains your server's private certificate
- Under the Actions menu, choose Create Certificate
- Choose Sign a PKCS#10 request, click next
- Enter or browse for the filename containing the request,
  OR
  Choose to load the request from the system clipboard (if you stored it there), click next
- Review the principal info (recommended: verify it offline in production scenario)
- Select the period of validity for the new certificate, click next
- Enter the [path-qualified] filename where you will store the new certificate, click next
- NOT RECOMMENDED: saving the certificate to the clipboard (created a single cert entry on the

client side instead of chain in my experience).

Importing your new certificate from the CA's Using Keytool

(after importing server's cert as trusted CA cert)
```
keytool -import {-alias alias} {-file cert_file} [-keypass keypass] {-storetype
storetype} {-keystore keystore} [-storepass storepass] {-v} {-Jjavaoption}
```
It should simply say "certificate was added to keystore" and finish with no dialog. ["alias" is your original keyEntry; the default self-signed certificate should now be overwritten with the CA-signed certificate]

Importing your new certificate from the CA's Using KeyMan

- Under the File menu, choose Import
- Select "local resource", click next
- Enter or browse for the filename ("cert_file"), click next
- You should get a popup dialog saying "Private Certificate Received", click OK

Saving the new or changed keystore file using KeyMan

- Under the File menu choose Save
- Enter the [path-qualified] filename (leave the default file format if PKCS12), click ok
- Enter the password if prompted

NOTE: If the server gets a certificate chain from a higher CA entity, the client could import that higher entity's certificate as a trusted CA certificate (it may already have that certificate by default), and it would thus trust the server certificate.

Troubleshooting

One extremely useful tip for troubleshooting SSL is to use the built-in SSL debugging features:
```
java -djavax.net.debug=help YourClassname
```
(this will give you a help message for SSL debugging features)

Unknown Protocol Error

Problem: `Exception in thread "main" java.net.MalformedURLException: unknown protocol: https`
Solution: Ensure that the ssl-specific initialisation code happens before you create a `java.net.URL` object.

Unrecognized SSL handshake

Problem: `ContextManager: IOException reading request, ignored - javax.net.ssl.SSLException: Unrecognized SSL handshake.`
Solution: The most likely problem is either:
(a) your url is `http://` instead of `https://`
or
(b) your client and server are using different versions of soap.jar (possibly 2.0/2.1).

Bad Certificate Error

Redo Steps 2, 3 and 4.

Socket Write Error

Problem: `java.lang.reflect.InvocationTargetException: java.net.SocketException: Connection aborted by peer: socket write error`
Solution: This problem occurs because (for some reason) the server cannot authenticate the client. Change the following line in server.xml:

```
<Parameter name="clientAuth" value="false"/>
```

Keytool Error

Problem: `keytool error: java.io.IOException: Keystore was tampered with, or password was incorrect`
Solution: Try deleting the keystore file and re-creating it (see the instructions at the beginning of this page).

Further Information

SOAP

- The SOAP Protocol: http://www.w3.org/TR/SOAP/.
- Apache SOAP: http://xml.apache.org/soap/.

SSL

- Sun JSSE FAQ: http://java.sun.com/products/jsse/FAQ.html
- "Modifying your Java Client to use Secure Sockets Layer": http://as400bks.rochester.ibm.com/html/as400/v4r5/ic2924/index.htm?info/java/rzaha/sslcex02.htm
- Notes on SSL error codes: http://www.mozilla.org/projects/security/pki/nss/ref/ssl/sslerr.html
- Cryptographic Downloads (includes OpenSSL): http://www2.psy.uq.edu.au/~ftp/Crypto/