Using Self-Signed Certificates for Web Service Security

How to compete with trusted certificate authorities

by Michael J. Remijan

One of the great things about the Java programming language is the Open Source community that provides great applications at little or no cost. An example of this is Apache Tomcat, which provides a solid Web server for development using servlet or JSP technology. Now that Web Service technology is maturing there's a potential for a whole scenario of applications to take advantage of a Swing feature-rich thin client on the front-end coupled to the data verification and business logic already developed in the Web or ejb tier. Such applications are only viable if they can be secure, however, security doesn't have to come at a great cost. The purpose of this article is to demonstrate how Web Service clients can use self-signed security certificates over the secure HTTPS protocol.

The Problem with Using Self-Signed Certificates

HTTPS typically works seamlessly with the non-secure HTTP protocol and doesn't interrupt the user's experience. This is because SSL certificates are designed to be verified and signed by a trusted third party. Verisign is a popular certificate authority. If a Web application requires secure communication, you can pay Verisign to sign your SSL certificate. Once Verisign does that, users on your Web site can switch between HTTP and HTTPS without interruption because all major Web browsers trust certificates signed by Verisign. Verisign is not the only option for getting certificates signed. To save operating costs, or for personal use, you can self-sign your own certificate. However, self-signing your certificate will interrupt your Web site user's experience. Typically the Web browser will display a dialog box asking if you want to trust a certificate you signed.

Michael J. Remijan is a senior Java consultant at Technology Partners in St. Louis, MO. He designs and develops mission-critical J2EE applications. Michael has a BS in mathematics/computer science from the University of Illinois and an MBA in technology management from the University of Phoenix.

?????@??????.com

Web browsers are nice because when they get a certificate signed by an unknown certificate authority there's an option to proceed. When developing Web Service clients for communication over HTTPS it's not so easy. When running Java code there's no dialog box asking about trusting a distrusted certificate authority. The JRE will throw an exception trying to connect over HTTPS to a Web site with a distrusted certificate:

```
Caused by: sun.security.validator.ValidatorException:
No trusted certificate found
```

There's no way to catch this exception and continue. To get the Web Service to work with a self-signed certificate the JRE has to somehow trust you as a certificate authority.

Solution Outline

To demonstrate a solution to this problem I'll do the following:

1. Generate and self-sign my own certificate
2. Configure Tomcat for SSL and make it use that certificate
3. Create an example Web Service to be called over HTTPS
4. Generate Web Service client code from WSDL
5. Demonstrate a client using a custom keystore solution

Generating a Self-Signed Certificate

The JDK comes with a tool, keytool.exe, that is used to mange SSL public/private keys. Keys are added and removed from a binary file on the file system. The default keystore file is JAVA_HOME\jre\lib\security\cacerts. This file contains the list of certificate authorities that the JRE will trust. A list of well-known trusted companies like Verisign is already in the keystore. To see this list, execute with password "changeit":

```
D:\>keytool -list -rfc -keystore JAVA_HOME\
jre\lib\security\cacerts
```

The keytool application can be used to edit this file. However, just in case something goes wrong it's better to create a new file. If keytool isn't told which file to use it creates HOME/.keystore by default.

To generate your own self-signed certificate execute:

```
D:\>keytool.exe -genkey -alias Tomcat -key-
alg RSA -storepass bigsecret –keypass big-
secret -dname "cn=localhost"
```

After executing this command there will be a .keystore file in your HOME directory. Here's what the switches mean.

- ***genkey***: Tells the keytool application to generate new public/private key pair.
- ***alias***: The name used to refer to the keys. Remember, the .keystore file can contain many keys.
- ***keyalg***: Generates public/private keys using the RSA algorithm.
- ***storepass***: What password is needed to access the .keystore file.
- ***keypass***: What password is needed to manage the keys.

- ***dname***: This value is very important. I used "localhost" because this example is designed to run locally. If a Web application is registered as http://www.myserver.com then this value must be www.myserver.com. If the names don't match the certificate will automatically be rejected.

Once the keytool application creates a new public/private key pair it automatically self-signs the key. You have just generated your own self-signed certificate, which can be used for HTTPS communications. You only need to extract the self-signed public key. I'll show how to do this later.

Configuring Tomcat for SSL

Now you have to configure Tomcat to use your self-signed certificate. I used Tomcat 5.0.30. Edit the TOMCAT/conf/server.xml file. Search the file for "8443" and uncomment the <Connector…/> bound to that port. Then you'll have to add the following property to the <Connector…/>:

```
keystorePass="bigsecret"
```

When the JRE starts, it will automatically find the HOME/.keystore file and Tomcat will try to access it using the password "bigsecret." When Tomcat starts there should be output to the console that looks similar to:

```
Feb 4, 2006 3:11:23 PM org.apache.coyote.
http11.Http11Protocol start
INFO: Starting Coyote HTTP/1.1 on http-8443
```

This means the <Connector…/> successfully read the .keystore file and you can now do secure HTTPS connections over the 8443 port. Open a Web browser and try https://localhost:8443/. Because the certificate is self-signed the Web browser will display a dialog box asking about trusting the connection. If accepted, all communications will be secure over HTTPS.

Creating the Web Service

I'm going to use the Apache Axis project to create a very simple Web Service. The Web Service will simulate checking for new e-mail messages. A Web Service client passes a token uniquely identifying a user. The Web Service returns a list of new e-mail messages (see Listing 1).

To get the Web Service deployed, follow these steps:

1. Cut and paste the code from Listing 1 into a file named Email.jws in Webapp's root directory.
2. Edit the Web.xml file, adding the Axis servlet and a *.jws mapping (Listing 2).
3. Put the Axis jar files in WEB-INF/lib. See References at the end of this article for the Axis project URL.

After deploying this article's accompanying WAR file (and configuring Tomcat for SSL), the Web Service is accessible securely over HTTPS at the following URL:

https://localhost:8443/JDJArticleWebService/Email.jws

Using WSDL2Java

The Axis project provides a tool named WSDL2Java that takes a Web Service WSDL and automatically create the Java source code needed to use the Web Service. See Listing 3 for the command line used to generate code for the Email.jws Web Service.

Notice the URL in Listing 3 used to access the WSDL. It's the non-secure HTTP protocol over port 8080. Why not use HTTPS over port 8443? Because of the self-signed certificate, the WSDL2Java tool will encounter the same exact certificate problem this article is trying to provide a solution for. So for now the non-secure protocol must be used. This means the generated code must be altered slightly replacing "http" and "8080" references with "https" and "8443." This article's accompanying client zip file contains the altered code.

Client with a Custom Keystore

The JRE's default keystore is JAVA_HOME\jre\lib\security\cacerts. Java applications will throw an exception whenever they are presented with your self-signed certificate because your certificate isn't in this keystore. Therefore, when developing a client there are two options. The first option is to put your self-signed certificate into the JRE's default keystore. Although this would work it's not a very good solution because customization is required on every client machine. The second option is to generate a custom keystore, put your self-signed certificate into it, and distribute the custom keystore as part of your application (typically inside a jar file).

To create a custom keystore for your client the following has to be done:

1. Export the self-signed public key from HOME/.keystore.
2. Import the self-signed public key into a new keystore for you client.

To export the self-signed public key from HOME/.keystore execute the following:

```
D:\>keytool.exe –export -rfc -alias Tomcat
–file Tomcat.cer -storepass bigsecret -key-
pass bigsecret
```

Now create the custom keystore for the client by importing Tomcat.cer:

```
D:>keytool.exe -import -noprompt -trust-
cacerts -alias Tomcat -file Tomcat.cer
-keystore CustomKeystore -storepass lit-
tlesecret
```

Using the switch "-keystore CustomKeystore" will create a new keystore file called "CustomKeystore" in the present working directory. You'll find the CustomKeystore file in the /classpath/resources/keystore directory of this article's client zip file. Replace this one with the file just generated.

Now all that's left to do is to create a client that uses this custom keystore. I'll demonstrate two ways to do this.

The first is to use the Java system properties javax.net.ssl.trustStore and javax.net.ssl.trustStorePassword to point to the CustomKeystore file and provide the password to access it. My example Web Service client in the jdj.wsclient.truststore package takes this approach (see Listing 4). The main() method sets the system properties then creates the objects to use the Web Service. When the JRE needs to access a keystore it looks for the "classpath/resources/keystore/CustomKeystore" file on the file system. Although this is a simple solution it's problematic because the keystore file must be on the file system and the client code must know where to look for it.

The second is a more portable solution that keeps resources inside the jar file and avoids the file system issues. The client code is responsible for

reading the CustomKeystore file and somehow using it to create a secure connection to the server.

My example Web Service client in the jdj.wsclient.socketfactory package takes this approach (see Listing 5). Listing 5 shows how to read the CustomKeystore file as a resource and use it to create a javax.net.ssl.SSLSocketFactory. Configuring the Axis pluggable architecture, the MySocketFactory class can be then used to create secure Socket objects from this factory.

Conclusion

This article started with a simple problem: I wanted to secure Web Service communications over HTTPS using my own self-signed certificates. By default, the JRE will reject my application's self-signed certificate because I am not a trusted certificate authority. To get secure communications to work I had to get the Web Service client JRE to trust my self-signed certificate. To achieve this, I used the keytool application and generated a new public/private key pair, extracted the self-signed public key, and then created a new keystore and imported this self-signed certificate. I then created a totally self-contained Web Service client that doesn't require any client-side configuration.

References

- http://tomcat.apache.org/tomcat-5.0-doc/ssl-howto.html
- http://ws.apache.org/axis/

Listing 1

```
import java.util.*;

public class Email {

 public List
 getNewMessages(String id)
 {
   List l = new ArrayList(3);
   l.add("1");
   l.add("2");
   l.add("3");
   return l;
 }
}
```

Listing 2

```
<servlet>
  <servlet-name>
    AxisServlet
  </servlet-name>
  <display-name>
    Apache-Axis Servlet
  </display-name>
  <servlet-class>
    org.apache.axis.transport.http.AxisServlet
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>
    AxisServlet
  </servlet-name>
  <url-pattern>
    *.jws
  </url-pattern>
</servlet-mapping>
```

Listing 3

```
java
 -classpath
  .;axis.jar;log4j-1.2.8.jar
  ;commons-logging-1.0.4.jar
  ;commons-discovery-0.2.jar
  ;jaxrpc.jar;saaj.jar
  ;wsdl4j-1.5.1.jar
 org.apache.axis.wsdl.WSDL2Java
  -p jdj.wsclient.shared
  http://localhost:8080/JDJArticle/Email.jws?wsdl
```

Listing 4

```
public static
void main(String[] args)
throws Exception
{
  System.setProperty(
   "javax.net.ssl.trustStore",
   "classpath/resources/keystore/CustomKeystore");

  System.setProperty(
   "javax.net.ssl.trustStorePassword",
   "littlesecret");

  EmailServiceLocator wsl =
   new EmailServiceLocator();

  Email_PortType ews =
   wsl.getEmail();

  Object [] objects =
   ews.getNewMessages("12345");

  out("Msg Count: " + objects.length);
}
```

Listing 5

```
public MySocketFactory(Hashtable table)
throws Exception
{
  out("Created!");
  KeyStore ks =
   KeyStore.getInstance(
    KeyStore.getDefaultType()
   );

  char [] password =
   "littlesecret".toCharArray();

  String keystore =
   "/resources/keystore/CustomKeystore";

  Class tclass =
   this.getClass();

  InputStream is =
   tclass.getResourceAsStream(
    keystore
   );

  ks.load(is, password);

  KeyManagerFactory kmf =
   KeyManagerFactory.getInstance("SunX509");

  kmf.init(ks,password);

  TrustManagerFactory tmf =
   TrustManagerFactory.getInstance("SunX509");

  tmf.init(ks);

  SSLContext context =
   SSLContext.getInstance("SSL");

  context.init(
   kmf.getKeyManagers(),
   tmf.getTrustManagers(),
   new SecureRandom()
  );

  factory =
   context.getSocketFactory();
}
```