

Configuring HTTPS

Background

HTTPS is HTTP over SSL or TLS, that is, secure HTTP. SSL and TLS normally rely on Public Key Cryptography, which requires a pair of *keys* to encrypt and decrypt messages – one key is a *private key* and the other is a *public key*. As the names imply, a private key is only known to the person/computer that created it, while the public key is made available to everyone. The public key and the private key complement each other. There are two main operations that can be performed: *encryption* and *signing*. Encryption is used to make the content of a message secret and is done using the public key; decryption can then only be performed using the corresponding private key. Signing is used to assert the authenticity of a message and involves similar operations, but is done using the private key; then, validating the signature is done using the corresponding public key.

The distribution of the public keys and their binding to an identity is done by the *Public Key Infrastructure (PKI)*, which uses certificates. A *certificate* (usually X.509) is essentially a signed statement that includes a public key and other information such as date of validity, subject distinguished name (which associates an identity with this) or the issuer distinguished name (which gives the identity of the signer). The signer of such a certificate is usually a *certificate authority (CA)*, which may be a company such as Thawte or VeriSign, who may charge you for this service, or can be a local CA created within your institution or company. PKIs describe the relationships and trust models between the CAs and are associated with legal documents describing the intended use of various X.509 attributes (depending on CA policies). However, certificates can also be *self-signed*, in which case the trust model has to be established by some other means (for example, someone you trust gives you this certificate). Self-certification is fine for testing in a limited and controlled environment, but is not generally suitable for production environments.

In HTTPS, HTTP lies on top of SSL (or TLS). The client first establishes the SSL connection (which is not aware of HTTP): the client initiates a conversation with a server. As part of the SSL handshake, the server first sends its certificate to the client as proof that the server is who/what it claims to be. This certificate is checked by the client against a list of certificates that it 'trusts'. How does a client know which certificates to trust? It depends on the client. A Java program uses a trust store (which is by default the 'cacerts' file within the JRE and which comes with a pre-populated list of trusted certificates/certification authorities); what the trust store is may be configured in Java. Similarly, browsers may maintain their own lists or – in the case of Windows or OSX – may use a feature of the operating system (e.g. Internet Options or Keychain). You can add your own certificates to these lists, or tell the Java VM, browser and/or your client operating system to trust the certificates you have created. If as part of this SSL handshake, the server certificate is not verified and trusted by the client, the SSL connection will be closed before any HTTP traffic.

When connecting to a server, not only the certificate should be verified, but the identity of the host name of the server should be checked against the certificate. Clients check that they are indeed communicating with the server they intend by checking that the *common name (CN)* field of the subject distinguished name in the server certificate is the host name intended. This host name can also be placed in the subject alternative name attribute in the certificate.

A Java container of keys and certificates is called a *keystore*. There are two usages for keystores: as a *keystore* (the name may be confusing indeed) and as a *truststore*. The keystore contains the material of the local entity, that is the private key and certificate that will be used to connect to the remote entity (necessary on a server, but can be used on a client for client-side authentication). Its counterpart, the truststore, contains the certificates that should be used to check the authenticity of the remote entity's certificates. There are various keystore types supported by the Sun JVM, mainly Java's own JKS format (default) or the PKCS#12 format which tends to be used by browsers and tools such as OpenSSL.

Step 1 : Creating Keys and a Self-Signed Certificate

Java comes with a command-line utility called *keytool* that can be used to handle keystores, in particular to create keys, certificate requests (CSR) and self-signed certificates (there are other utilities, such as *OpenSSL* and *KeyMan* from IBM). More precisely, keytool will create a *keystore* file that can contain one or more key pairs and certificates (we will only create one of each). By default, when you create a pair of keys, keytool will also create a self-signed certificate. The most important thing in this step is that you correctly specify the name of the machine where the certificate will be used as the common name, in the '-dname' option (see below). In the following example, the machine is called 'serverX' (the command-line options are put onto separate lines for readability only):

```
keytool -genkey
-v
-alias serverX
-dname "CN=serverX,OU=IT,O=JPC,C=GB"
-keypass password
-keystore serverX.jks
-storepass password
-keyalg "RSA"
-sigalg "SHA1withRSA"
-keysize 2048
-validity 3650
```

The output should be:

```
Generating 2,048 bit RSA key pair and self-signed certificate (MD5withRSA) with
a validity of 3,650 days
for: CN=serverX, OU=IT, O=JPC, C=GB
[Storing serverX.jks]
```

To explain each option:

<i>-genkey</i>	generate a pair of keys and a self-signed certificate
<i>-v</i>	display the output message
<i>-alias</i>	a unique name for the keys (does not need to be the name of the machine)
<i>-dname</i>	details of the machine where the keys and certificate will be used
<i>-keypass</i>	the password for the key pair identified by the '-alias' option
<i>-keystore</i>	the name of the operating system file where the keys/certificate will be saved
<i>-storepass</i>	the password for the keystore file
<i>-keyalg</i>	the encryption algorithm to use; "DSA" or "RSA"
<i>-sigalg</i>	the signature algorithm to use; "Sha1withDSA" and "MD5withRSA" are two
<i>-keysize</i>	the size of the key (larger = more secure; e.g. 512, 1024 or 2048)
<i>-validity</i>	the number of days before the certificate expires (3650 = approx. 10 years)

Keystore files can have different formats. The example above, the default format of “JKS” (Java Key Store) was used. The type for PKCS#12 files (.p12) is "PKCS12", which can be specified by adding the following options to the keytool command line:

```
-storetype "PKCS12"
```

Step 2: Exporting the Self-Signed Certificate

The file 'serverX.jks' now contains the keys and a self-signed certificate. To be useful, the certificate needs to be exported so that it can be imported into other keystores such as those used by the Java VM or Windows. To export the certificate, use keytool with the following options:

```
keytool -export
-v
-alias serverX
-file serverX.cer
-keystore serverX.jks
-storepass password
```

The output should be:

```
Certificate stored in file <serverX.cer>
```

To explain each option:

<i>-export</i>	export the certificate
<i>-alias</i>	the name of the public key/certificate to export
<i>-file</i>	the name of a file where the certificate is to be saved
<i>-keystore</i>	the name of the keystore file containing the certificate
<i>-storepass</i>	the password for the keystore file

Note that the name of the alias and the keystore/certificate files are not significant, but they are named 'serverX' for consistency and clarity.

Step 3: Importing the Self-Signed Certificate

You should now have a file called 'serverX.cer' that contains your server's self-signed certificate. The server will present this certificate whenever an HTTPS client sends a request. There are different ways of installing the certificate on the server; in the Restlet example server code below, the certificate is loaded from the keystore when the Restlet server is started.

There are different ways to get a HTTPS client to trust your certificate. If you are using a browser, there may be an option to add it to a list of trusted certificates. In Windows XP, the certificate can be added to the 'Trusted Root Certification Authorities' via Internet Options (in IE7 or Control Panel - Internet Options). On the 'Content' tab, click 'Certificates', then go to 'Trusted Root Certification Authorities' tab, click 'Import...' and follow the steps to import your certificate file ('serverX.cer'). It will give warnings about not being verified, which is ok for testing, but it must be properly signed by proper Certification Authority for production. Firefox 3 also has the ability to add exceptions to trust individual certificates (self-signed or issued by an unknown CA).

If you are using another Java program instead of a browser, then you need to let the Java VM know about the certificate. There are several ways to do this, but here are two:

1. Import the certificate to the Java VM trusted certificates file, which is called 'cacerts' by default and located in the *lib/security* directory of the Java home directory, for example *C:\Program Files\Java\jre6\lib\security\cacerts*

The keytool command to do this is:

```
keytool -import
-alias serverX
-file serverX.cer
-keystore "C:\Program Files\Java\jre6\lib\security\cacerts"
-storepass "changeit"
```

Note that the default password for the cacerts keystore file is 'changeit'.

2. Add the following Java VM arguments to your Java client command line:

```
-Djavax.net.ssl.trustStore=C:\\somedir\\serverX.jks
1-Djavax.net.ssl.trustStoreType=JKS
-Djavax.net.ssl.trustStorePassword=password
```

These arguments tell the Java VM where to find your certificate. **Please note that this approach should only be used in a test environment, not in production, as the password is shown in plain text.**

Step 4: Sample Restlet Server Code

In addition to the standard Restlet jar files, you also need to reference jar files for HTTPS. The 'Simple' HTTPS connector uses these jar files:

```
lib/com.noelios.restlet.ext.simple_3.1.jar
lib/org.simpleframework_3.1/org.simpleframework.jar lib/com.noelios.restlet.ext.ssl.jar
lib/org.jsslutils_0.5/org.jsslutils.jar
```

The server code in this example will explicitly load the certificate from the keystore file (serverX.jks):

```
package com.jpc.samples;

import org.restlet.Component;
import org.restlet.Server;
import org.restlet.data.Parameter;
import org.restlet.data.Protocol;
import org.restlet.util.Series;

public class SampleServer { public static void main(String[] args) throws Exception {
// Create a new Component.
Component component = new Component();

// Add a new HTTPS server listening on port 8183
Server server = component.getServers().add(Protocol.HTTPS, 8183);
Series<Parameter> parameters = server.getContext().getParameters();

// Note refactoring in Restlet version 1.2:
// com.noelios.restlet.ext.ssl.PkixSslContextFactory moved to org.restlet.ext.ssl.PkixSslContextFactory
//and
// com.noelios.restlet.util.DefaultSslContextFactory moved to
org.restlet.engine.security.DefaultSslContextFactory");
//
// DefaultSslContextFactory is another sslContextFactory that can be used if desired.

// Using Restlet version 1.1 package:
parameters.add("sslContextFactory", "com.noelios.restlet.ext.ssl.PkixSslContextFactory");
parameters.add("keystorePath", "<path>serverX.jks");
parameters.add("keystorePassword", "password");
parameters.add("keyPassword", "password");
```

1. <http://wiki.restlet.org/somedir/serverX.jks>

```
parameters.add("keystoreType", "JKS");  
  
// Attach the sample application.  
component.getDefaultHost().attach("", new SampleApplication());  
  
// Start the component.  
component.start();  
}  
}
```

Step 5: Sample Restlet Client Code

```
package com.jpc.samples;  
import java.io.IOException;  
import org.restlet.Client;  
import org.restlet.data.Form;  
import org.restlet.data.Protocol;  
import org.restlet.data.Reference;  
import org.restlet.data.Response;  
import org.restlet.resource.Representation;  
  
public class SampleClient {  
  
    public static void main(String[] args) throws IOException {  
        // Define our Restlet HTTPS client.  
        Client client = new Client(Protocol.HTTPS);  
  
        // The URI of the resource "list of items".  
        Reference samplesUri = new Reference("https://serverX:8183/sample");  
  
        // Create 9 new items  
        for (int i = 1; i < 10; i++)  
        {  
            Sample sample = new Sample(Integer.toString(i), "sample " + i, "this is sample " + i + ".");  
            Reference sampleUri = createSample(sample, client, samplesUri);  
            if (sampleUri != null) {  
                // Prints the representation of the newly created resource.  
                get(client, sampleUri);  
            }  
        }  
  
        // Prints the list of registered items.  
        get(client, samplesUri);  
    }  
  
    ...other code not shown (similar to original HTTP Restlet example)...
```

Conclusion

That's it! Your client and server should now be communicating via HTTPS. Note that the information contained in this article is meant only as rudimentary starting point for using HTTPS with Restlet and not as a comprehensive guide to secure web applications. Comments and feedback welcome!