DZone

Securing Your Tomcat App with SSL and Spring Security

by Roger Hughes MVB · Dec. 14, 12 · Java Zone

If you've seen my last blog, you'll know that I listed ten things that you can do with Spring Security. However, before you start using Spring Security in earnest one of the first things you really must do is to ensure that your web app uses the right transport protocol, which in this case is HTTPS - after all there's no point in having a secure web site if you're going to broadcast your user's passwords all over the internet in plain text. To setup SSL there are three basic steps...

Creating a Key Store

The first thing you need is a private keystore containing a valid certificate and the simplest way to generate one of these is to use Java's keytool utility located in the $JAVA_HOME/bin directory.

```
keytool -genkey -alias MyKeyAlias -keyalg RSA -keystore /Users/Roger/tmp/roger.keystore
```

In the above example,

- **-alias** is the unique identifier for your key.
- **-keyalg** is the algorithm used to generate the key. Most examples you find on the web usually cite 'RSA', but you could also use 'DSA' or 'DES'
- **-keystore** is an optional argument specifying the location of your key store file. If this argument is missing then the default location is your $HOME directory.

RSA stands for Ron Rivest (also the creator of the RC4 algorithm), Adi Shamir and Leonard Adleman
DSA stands for Digital Signature Algorithm
DES stands for Data Encryption Standard
For more information on keytool and its arguments take a look at this Informit article by Jon Svede
When you run this program you'll be asked a few questions:

```
Roger$ keytool -genkey -alias MyKeyAlias -keyalg RSA -keystore /Users/Roger/tmp/roger.keystore
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  localhost
What is the name of your organizational unit?
  [Unknown]:  MyDepartmentName
What is the name of your organization?
```

Most of the fields are self explanatory; however for the first and second name values, I generally use the machine name - in this case localhost.

Updating the Tomcat Configuration

The second step in securing your app is to ensure that your tomcat has an SSL connector. To do this you need to find tomcat's server.xml configuration file, which is usually located in the 'conf' directory. Once you've got hold of this and if you're using tomcat, then it's a matter of uncommenting:

…and making it look something like this:

```
<Connector SSLEnabled="true" keystoreFile="/Users/Roger/tmp/roger.keystore" keystorePass="password" port="8443" sche
```

Note that the password "password" is in plain text, which isn't very secure. There are ways around this, but that's beyond the scope of this blog. If you're using Spring's tcServer, then you'll find that it already has a SSL connector that's configured something like this:

```
<Connector SSLEnabled="true" acceptCount="100" connectionTimeout="20000" executor="tomcatThreadPool" keyAlias="tcser
```

…in which case it's just a matter of editing the various fields including keyAlias, keystoreFile and keystorePass.

Configuring your App

If you now start tomcat and run your web application, you'll now find that it's accessible using HTTPS. For example typing https://localhost:8443/my-app will work, but so will http://localhost:8080/my-app This means that you also need to do some jiggery-pokery on your app to ensure that it only responds to HTTPS and there are two approaches you can take.

If you're not using Spring Security, then you can simply add the following to your web.xml before the last web-app tag:

If you are using Spring Security, then there are a few more steps to getting things going. Part of the general Spring Security setup is to add the following to your web.xml file. Firstly you need to add a Spring Security application context file to the contextConfigLocation context-param:

```
<context-param>
          <param-name>contextConfigLocation</param-name>
          <param-value>/WEB-INF/spring/root-context.xml
           /WEB-INF/spring/appServlet/application-security.xml
          </param-value>
     </context-param>
```

Secondly, you need to add the Spring Security filter and filter-mapping:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/security"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
           http://www.springframework.org/schema/security
           http://www.springframework.org/schema/security/spring-security-3.1.xsd">

       <http auto-config='true' >
          <intercept-url pattern="/**" requires-channel="https" />
       </http>

       <authentication-manager>
       </authentication-manager>

</beans:beans>
```

In the example above intercept-url element has been set up intercept all URLs and force them to use the https channel.

The configuration details above may give the impression that it's quicker to use the simple web.xml config change, but if you're already using Spring Security, then it's only a matter of adding a requires-channel attribute to your existing configuration.

A sample app called tomcat-ssl demonstrating the above is available on git hub at: https://github.com/roghughe/captaindebug

Topics: JAVA , FRAMEWORKS , SECURITY , TIPS AND TRICKS

Published at DZone with permission of Roger Hughes, DZone MVB. See the original article here.