Exercise 7

WS-Security: securing the JAXWS Service and configuring the client to use security

Prior Knowledge

Exercises 3 and 4 – creating a JAXWS Service and CXF Client.
Exercise 6 – creating a pair of keystores
Lectures on WS-Security

Objectives

Understand how to authenticate using UsernameToken and Sign/Encrypt a message using X.509 certificates.

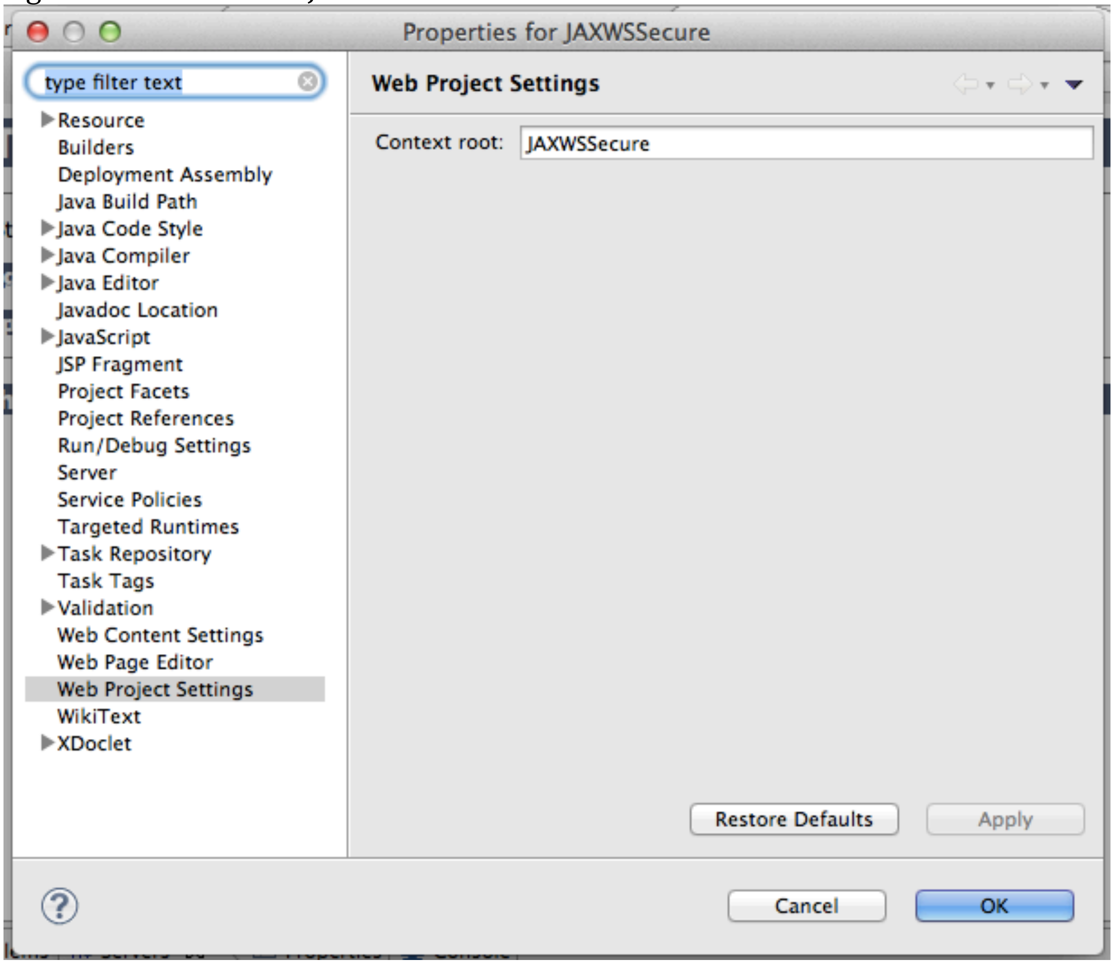
Software Requirements

- Java Development Kit 7
- Tomcat 7.0.57 or later
- Eclipse JEE workbench Kepler
- Apache CXF 2.7.13 or later
- Service running from Exercise 3, Client from Exercise 4

1. If you didn't complete the Keystore exercise you can find the .jks files you need in ~/backup_keys

2. Take a copy in Eclipse of your JAXWSSample service – rename it to **JAXWSSecure**

3. Two things you need to do manually to "really" rename this.
   a. First change the WebContent/WEB-INF/web.xml and edit the <display-name> element to match the project name.
   b. Go to your JAXWSSecure project properties (Right-click and then Properties). Edit the **Web Project Settings** to make sure it has the right **Context Root** of JAXWSSecure.



Web Project Settings

Context root: JAXWSSecure

4. We are going to start with just authentication using UsernameToken. This is the WS-Security equivalent of HTTP Basic Authentication.

5. Go to the webcontent/orderserviceimpl.wsdl and use the Source View to edit the WSDL to include the following security policy. Put it right at the top of the WSDL inside the main **definition** tag.

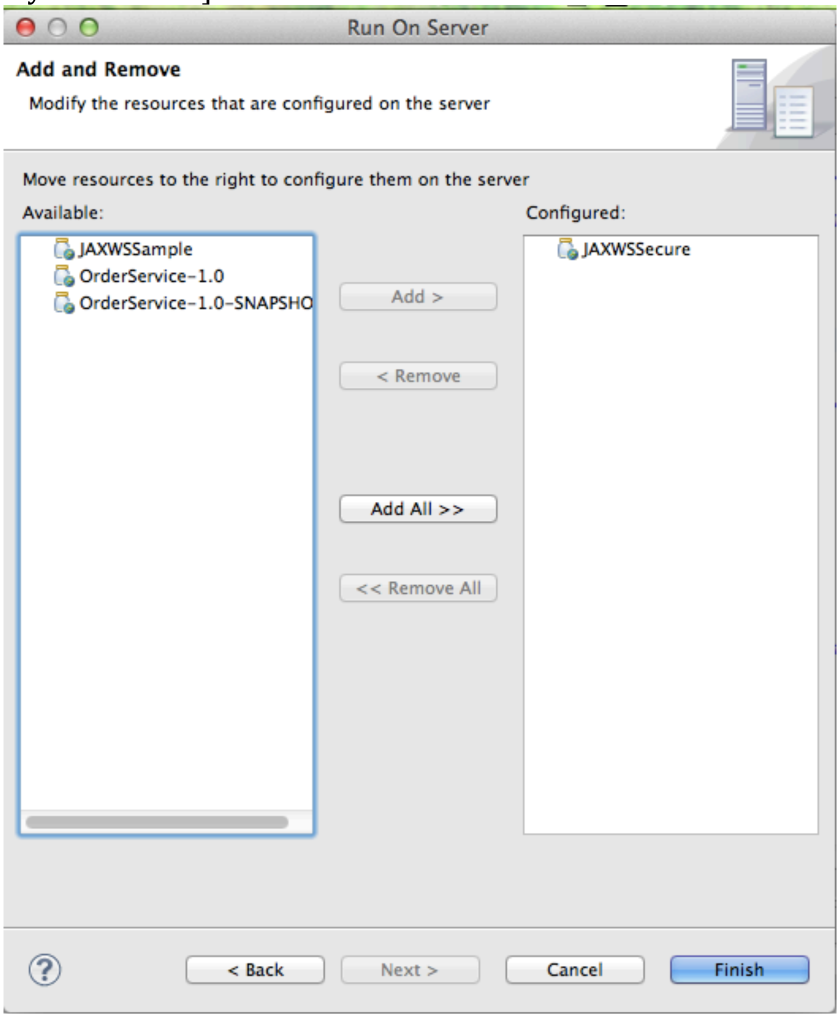
   [You can find these snippets online here: http://freo.me/TMH3iq]

```
<wsp:Policy wsu:Id="UsernameToken" xmlns:wsu=
     "http://docs.oasis-open.org/wss/2004/01/oasis-200401-
wss-wssecurity-utility-1.0.xsd"

xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
      xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702">
    <wsp:ExactlyOne>
      <wsp:All>
        <sp:SupportingTokens>
          <wsp:Policy>
            <sp:UsernameToken
sp:IncludeToken=".../IncludeToken/AlwaysToRecipient"/>
          </wsp:Policy>
        </sp:SupportingTokens>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>
```

6. Now we need to *apply* this policy to the service. We can do this by attaching this policy into the service in several places. We are going to apply this to the whole service. Find the **service** element in your WSDL and add the next short snippet so it looks like this:

```
<wsdl:service name="OrderServiceImplService">
      <wsp:PolicyReference
xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/polic
y"
                 URI="#UsernameToken" />

           <wsdl:port… etc
```

7. Run the server in Eclipse under Tomcat, using **Run As->Run on Server** to test it is running on the right root. I would recommend removing the other webapps so you just have the JAXWSSecure service running: [Hint.. try remove all.]


Run On Server

Add and Remove

Modify the resources that are configured on the server

Move resources to the right to configure them on the server

Available:
- JAXWSSample
- OrderService-1.0
- OrderService-1.0-SNAPSHO

Configured:
- JAXWSSecure

8. Look at the WSDL that the server is publishing (using your browser) and validate that the WS-Policy is correctly published.

9. CXF is smart enough to read and implement this policy. To check this out, create a new client using the command-line/Ant approach. See the JAX-WS Client exercise to remind you how to do this. Import the code into Eclipse as before.

10. Try running your client. You should get an error like this:
```
WARNING: Interceptor for
{http://jaxws.me.freo/}OrderServiceImplService#{http
://jaxws.me.freo/}getOrders has thrown exception,
unwinding now
org.apache.cxf.ws.policy.PolicyException: No
username available
```

11. We need to pass the username and password to the client code, so that it can send them to the server. In your generated client you will find a line like the following:
    ```
    OrderService port = ss.getOrderServiceImplPort();
    ```

    Add the following lines afterwards:

    ```
    Map<String,Object> ctx =
    ((BindingProvider)port).getRequestContext();
    ctx.put("ws-security.username", "paul");
    ctx.put("ws-security.password", "paul");
    ```

    Note you will need to make sure the imports are right:

    ```
    import java.util.Map;
    import javax.xml.ws.BindingProvider;
    ```

12. Now the client should be happy, but the server is still not ok yet.
    You can test this out if you like.

    We need to provide the server with a way of validating usernames and passwords. This is done using a "callback" class. What happens is that the CXF/WSS4J code is running and it needs to validate the password. It calls back into YOUR code with the username and password[1], and asks to validate it. Your code would normally go talk to LDAP or a user store at this point. In our code we won't do that. We'll have a fixed password and username for ease of use.

[1] Actually, it used to call into your code with the password, allowing you to hash it and compare it to a hashed value. More recently this was changed so that you have to pass the correct password to the callback and it validates. See http://coheigea.blogspot.co.uk/2011/02/usernametoken-processing-changes-in.html (http://freo.me/RPmkLN) This is a retrograde step in my opinion, but there does seem to be a workaround/new approach in mentioned in that blog. I haven't yet tried it.

13. In your **Service** code, create a package me.freo.security and a class me.freo.security.PasswordCallback

14. Cut and paste this code (also available here: http://freo.me/TNfipS)

```
package me.freo.security;

import java.io.IOException;

import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;

import org.apache.ws.security.WSPasswordCallback;

public class PasswordCallback implements CallbackHandler {

      @Override
      public void handle(javax.security.auth.callback.Callback[]
cbs)
                  throws IOException, UnsupportedCallbackException
{

            for (int i = 0; i < cbs.length; i++) {
                  WSPasswordCallback pwcb = (WSPasswordCallback)
cbs[i];
                  String id = pwcb.getIdentifier();

                  switch (pwcb.getUsage()) {
                  case WSPasswordCallback.USERNAME_TOKEN:

                        if ("paul".equals(id)) {
                              pwcb.setPassword("paul");
                        }
                        break;
                  }
            }
      }
}
```

15. Your server needs to find this class. We do this by configuring the cxf-beans.xml in the WEB-INF. You need to add the following snippet into your beans.xml as a child of **jaxws:endpoint:**

```
<jaxws:properties>
     <entry key="ws-security.callback-handler"
     value="me.freo.security.PasswordCallback" />
</jaxws:properties>
```

16. Now run the service in the embedded Tomcat mode and test it with your client. Try changing the password to check its really checking it. Also look at the console logs that the server prints and verify that the password is being

passed in plain text.

17. Now that we've got the password in plain text, it would make sense to encrypt the message so we can't see it. We could use HTTPS but that wouldn't be as fun or interesting as getting Signature and Encryption to work ☺

18. The first step is to add in a Signature and Encryption Policy into your server-side WSDL. Do not remove the UsernameToken policy but comment out the line that applies the policy to the service (<wsp:PolicyReference>…</wsp:PolicyReference>)

19. Next to where you added the Username policy, add in the second policy xml that you will find in the snippets. It starts: `<wsp:Policy wsu:Id="SignEncrypt"`

20. Now once again in the service definition, apply this policy using the following snippet:
```
<wsp:PolicyReference
xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy
" URI="#SignEncrypt" />
```

21. We need to do quite a lot to tell the server about how to do signature and encryption. First create a file **crypto.properties** in the WEB-INF directory of your web app with the following:
```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=serverpass
org.apache.ws.security.crypto.merlin.keystore.alias=server
org.apache.ws.security.crypto.merlin.file=/home/ox-soa/backup_keys/serverkeystore.jks
```

    This is available here:
    http://freo.me/12fn3Hp

    Remember to change the key directory path to point to your server keystore (if necessary).

22. The password callback handler now has to do two things:
    1) We might still want to validate client passwords
    2) We need the password to the keystore

23. Copy and Paste your existing PasswordCallback.java to PasswordAndSignEncrCallback.java

24. Add the following code to the switch-case:

```
case WSPasswordCallback.DECRYPT:
case WSPasswordCallback.SIGNATURE:

      // Return password for Keystore
      if ("server".equals(id)) {
            pwcb.setPassword("serverpass");
      }
      break;
```

25. We also have to sort out the beans.xml so that the server knows where to find all the new things we've created! Change the <jaxws:properties> section to read:

```
<jaxws:properties>
  <entry key="ws-security.signature.properties"
     value="WEB-INF/crypto.properties" />
  <entry key="ws-security.encryption.properties"
     value="WEB-INF/crypto.properties" />
  <entry key="ws-security.signature.username"
     value="server" />
  <entry key="ws-security.encryption.username"
     value="useReqSigCert" />
  <entry key="ws-security.callback-handler"
     value="me.freo.security.PasswordAndSignEncrCallback" />
</jaxws:properties>
```

You can find docs for all these here:
http://cxf.apache.org/docs/ws-securitypolicy.html

If you are eagle-eyed you might have noticed that there is no user useReqSigCert defined. This is actually a special word that means use the certificate that came in for signature to encrypt the response. This means that you can have many different clients each automatically get encrypted responses using their own certificate.

26. That is the server done. But we also need to give the client a config for sign/encrypt. This again requires us to create a properties file. This time create it in the same directory of your Eclipse Client project where the build.xml lives.

    The contents of client-sign.properties can be found here: http://freo.me/Uvrrxe

    Edit this to point to your client keystore (if necessary).

27. Guess what, we need a ClientCallback.java as well to manage the password for the client keystore.

    The code is available here: http://freo.me/120qQJc
    Add it to your **Client project** in package freo.me.security.

28. Edit your client code so that the context manipulation looks like this:

```
Map<String,Object> ctx =
((BindingProvider)port).getRequestContext();
//        ctx.put("ws-security.username", "paul");
//        ctx.put("ws-security.password", "paul");
ctx.put("ws-security.signature.properties",
  "client-sign.properties");
ctx.put("ws-security.encryption.properties",
  "client-sign.properties");
ctx.put("ws-security.signature.username", "client");
ctx.put("ws-security.encryption.username", "server");
ctx.put("ws-security.callback-handler",
  "freo.me.security.ClientCallback");
```

29. Notice that we are using the client's key to sign the message, and the server's certificate to encrypt it.

30. So in theory, you are now ready to test your client with Signature and Encryption. Try it! Good luck.

31. If that all works, now re-enable Authentication by uncommenting the server-side policy for UsernameToken. Notice that you can have multiple policies applied and the server has the job of "merging them" into an effective policy.

32. You will need to uncomment out your u/p in the client code too.

33. Congratulations if you got this far. It wasn't easy!

34. Extension: try applying the policy more selectively to just parts of the service. For example requiring Sign/Encrypt all the time, but only need UsernameToken for createOrder.