WS-Security

WS-Security provides means to secure your services above and beyond transport level protocols such as HTTPS. Through a number of standards such as XML-Encryption, and headers defined in the WS-Security standard, it allows you to:

- Pass authentication tokens between services
- Encrypt messages or parts of messages
- Sign messages
- Timestamp messages
- Manage public keys using XKMS

CXF relies on WSS4J in large part to implement WS-Security. Within your own services, WS-Security can be activated by using WS-SecurityPolicy, which provides a comprehensive and sophisticated validation of the security properties of a received message. A non-WS-SecurityPolicy approach is usually also possible by way of CXF interceptors added to your service and/or client as detailed in this article.

Please note that there are some incompatibilities between WSS4J 1.6.x (used by Apache CXF 2.6.x and 2.7.x) and 2.0.x (used by Apache CXF 3.0.x and 3.1.x). The examples and links on this page mainly pertain to WSS4J 2.0.x and hence CXF 3.0.x. For more information on the changes in WSS4J 2.0.x please see the following migration page.

Overview of encryption and signing

WS-Security makes heavy use of public/private key cryptography. To really understand how to configure WS-Security, it is helpful - if not necessary - to understand these basics. The Wikipedia has an excellent entry on this, but we'll try to summarize the relevant basics here (This content is a modified version of the wikipedia content.)

With public key cryptography, a user has a pair of public and private keys. These are generated using a large prime number and a key function.

Alice

52ED879E70F71D92

Big random number

Key generation function

0

Alice's public key

Alice's private key

The keys are related mathematically, but cannot be derived from one another. With these keys we can encrypt messages. For example, if Bob wants to send a message to Alice, he can encrypt a message using her public key. Alice can then decrypt this message using her private key. Only Alice can decrypt this message as she is the only one with the private key.

Bob

Hello Alice!

Encrypt

6EB6957008E03CE4

Alice's public key

Alice

Hello Alice!

Decrypt

Alice's private key
Messages can also be signed. This allows you to ensure the authenticity of the message. If Alice wants to send a message to Bob, and Bob wants to be sure that it is from Alice, Alice can sign the message using her private key. Bob can then verify that the message is from Alice by using her public key.

### Configuring the WSS4J Interceptors

To enable WS-Security within CXF for a server or a client, you'll need to set up the WSS4J interceptors. You can either do this via the API for standalone web services or via Spring XML configuration for servlet-hosted ones. This section will provide an overview of how to do this, and the following sections will go into more detail about configuring the interceptors for specific security actions.

It is important to note that:

1. If you are using CXF 2.0.x, you must add the SAAJ(In/Out)Interceptors if you're using WS-Security (This is done automatically for you from CXF 2.1 onwards). These enable creation of a DOM tree for each request/response. The support libraries for WS-Security require DOM trees.
2. The web service provider may not need both in and out WS-Security interceptors. For instance, if you are just requiring signatures on incoming messages, the web service provider will just need an incoming WSS4J interceptor and only the SOAP client will need an outgoing one.

### Adding the interceptors via the API

On the Server side, you'll want to add the interceptors to your CXF Endpoint. If you're publishing your service using the JAX-WS APIs, you can get your CXF endpoint like this:

```java
import org.apache.cxf.endpoint.Endpoint;
import org.apache.cxf.jaxws.EndpointImpl;

EndpointImpl jaxWsEndpoint = (EndpointImpl) javax.xml.ws.Endpoint.publish("http://host/service", myServiceImpl);
Endpoint cxfEndpoint = jaxWsEndpoint.getServer().getEndpoint();
```

On the client side, you can obtain a reference to the CXF endpoint using the ClientProxy helper:

```java
import org.apache.cxf.endpoint.Endpoint;
import org.apache.cxf.endpoint.Server;
import org.apache.cxf.frontend.ServerFactoryBean;

ServerFactoryBean factory = ...;
...
Server server = factory.create();
Endpoint cxfEndpoint = server.getEndpoint();
```
import org.apache.cxf.frontend.ClientProxy;
...

GreeterService gs = new GreeterService();
Greeter greeter = gs.getGreeterPort();
...
org.apache.cxf.endpoint.Client client = ClientProxy.getClient(greeter);
org.apache.cxf.endpoint.Endpoint cxfEndpoint = client.getEndpoint();

Now you're ready to add the interceptors:

import org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor;
import org.apache.cxf.ws.security.wss4j.WSS4JOutInterceptor;
...

Map<String, Object> inProps = new HashMap<String, Object>();
... // how to configure the properties is outlined below;
WSS4JInInterceptor wssIn = new WSS4JInInterceptor(inProps);
cxfEndpoint.getInInterceptors().add(wssIn);
Map<String, Object> outProps = new HashMap<String, Object>();
... // how to configure the properties is outlined below;
WSS4JOutInterceptor wssOut = new WSS4JOutInterceptor(outProps);
cxfEndpoint.getOutInterceptors().add(wssOut);

Spring XML Configuration

If you're using Spring to build endpoints (e.g., web services running on a servlet container such as Tomcat), you can easily accomplish the above using your bean definitions instead.

<import resource="classpath:META-INF/cxf/cxf.xml" />
<import resource="classpath*:META-INF/cxf/cxf-extension-*.xml" />

<jaxws:endpoint id="myService"
  implementor="com.acme.MyServiceImpl"
  address="http://localhost:9001/MyService">
  <bean id="myPasswordCallback"
    class="com.mycompany.webservice.ServerPasswordCallback"/>
  <jaxws:inInterceptors>
    <bean class="org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor">
      <constructor-arg>
        <map>
          <entry key="action" value="UsernameToken"/>
          <entry key="passwordType" value="PasswordDigest"/>
          <entry key="signaturePropFile" value="..."/>
          <entry key="passwordCallbackRef"/>
          <ref bean="myPasswordCallback"/>
        </map>
      </constructor-arg>
    </bean>
  </jaxws:inInterceptors>
</jaxws:endpoint>

The entry keys and values given in the constructor-arg element above (action, signaturePropFile, etc.) map to the text strings in WSS4J's WSHandlerConstants and WSConstants classes for the corresponding WSHandlerConstants.XXXXX and WSConstants.XXXX constants you see in the section below (also see the WSS4J configuration page). So by viewing WSHandlerConstants, for example, you can see that the WSHandlerConstants.USERNAME_TOKEN value given below would need to be "UsernameToken" instead when doing Spring configuration.
If you want to avoid looking up the text keys for the WSHandlerConstants.XXXXX and WSConstants.XXXX constants, you can also use the Spring util namespace to reference static constants in your Spring context as shown below.

```xml
<beans
... xmlns:util="http://www.springframework.org/schema/util"
... xsi:schemaLocation="
  http://www.springframework.org/schema/util
  http://www.springframework.org/schema/util/spring-util.xsd">

  <bean class="org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor">
    <constructor-arg>
      <map>
        <entry value="UsernameToken">
          <key>
            <util:constant
              static-field="org.apache.wss4j.dom.handler.WSHandlerConstants.ACTION"/>
          </key>
        </entry>
        ...
      </map>
    </constructor-arg>
  </bean>

</beans>
```

Additional Configuration Options

While the CXF WSS4J interceptors support the standard configuration properties available in WSHandlerConstants.XXXXX and WSConstants.XXXX, CXF also provides access to some additional low level configuration capabilities in WSS4J and some other security related interceptors.

Validating Signature and/or Encryption of Message Contents

As of CXF 2.2.8, the CryptoCoverageChecker interceptor allows one to validate signature and encryption coverage of message contents without migrating to a WS-SecurityPolicy based configuration. The interceptor can support enforcement of signature and encryption coverage at both the element and content level (be aware that the combination of signature and content do not represent a valid combination of coverage type and coverage scope). To configure this interceptor using the API, follow the example below.

```java
import org.apache.cxf.ws.security.wss4j.CryptoCoverageChecker;
import org.apache.cxf.ws.security.wss4j.CryptoCoverageChecker.XPathExpression;
import org.apache.cxf.ws.security.wss4j.CryptoCoverageUtil.CoverageType;
import org.apache.cxf.ws.security.wss4j.CryptoCoverageUtil.CoverageType;

Map<String, String> prefixes = new HashMap<String, String>();
prefixes.put("ser", "http://www.sdj.pl");
prefixes.put("soap", "http://schemas.xmlsoap.org/soap/envelope/");

List<XPathExpression> xpaths = Arrays.asList(
    new XPathExpression("//ser:Header", CoverageType.SIGNED, CoverageScope.ELEMENT),
    new XPathExpression("//soap:Body", CoverageType.ENCRYPTED, CoverageScope.CONTENT));

CryptoCoverageChecker checker = new CryptoCoverageChecker(prefixes, xpaths);
```

The interceptor can also be configured in Spring using the conventional bean definition format.

After configuring the interceptor as above, simply add the interceptor to your client or server interceptor chain as shown previously with the WSS4J interceptors. Ensure that you include the WSS4JInInterceptor in the chain or all requests will be denied if you enforce any coverage XPaths.
The CryptoCoverageChecker is somewhat complex to set up for the most common use-cases for signature verification and decryption, as it involves adding XPath expressions and the corresponding prefix/namespace pairs. In Apache CXF 2.4.9, 2.5.5 and 2.6.2, a new subclass of CryptoCoverageChecker has been introduced. The DefaultCryptoCoverageChecker provides an easy way to ensure that the SOAP Body is signed or encrypted, that the Timestamp is signed, and that the WS-Addressing ReplyTo and FaultTo headers are signed (if they are present in the message payload).

The default configuration is that the SOAP Body, (WSU) Timestamp and WS-Addressing ReplyTo and FaultTo headers must be signed (if they exist in the message payload). This provides an out-of-the-box way of preventing XML Signature wrapping attacks. All that is required is that the DefaultCryptoCoverageChecker be added to the in-interceptor chain. For example:

```xml
<jaxws:inInterceptors>
    <bean class="org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor">
        <constructor-arg>
            <map>
                <entry key="action" value="Signature Timestamp"/>
                <entry key="signaturePropFile" value="..."/>
                <entry key="passwordCallbackClass" value="..."/>
            </map>
        </constructor-arg>
    </bean>
    <bean class="org.apache.cxf.ws.security.wss4j.DefaultCryptoCoverageChecker"/>
</jaxws:inInterceptors>
```

As of CXF 2.5.11, 2.6.8 and 2.7.5, it is possible to only check that a received message meets cryptographic requirements via the CryptoCoverageChecker if it is not a fault. This is useful in the scenario where a client is using the CryptoCoverageChecker interceptor to verify security requirements from a service response. In this scenario, you may want to get the original service Fault rather than have the CryptoCoverageChecker throw an exception if a Fault message from the service isn’t secured. To enable this behaviour, then set the "checkFaults" boolean property on CryptoCoverageChecker to "false".

### Custom Processors

As of CXF 2.0.10 and 2.1.4, you can specify custom WSS4J Processor configurations on the WSS4JInInterceptor. To activate this configuration option, one provides a non-WSS4J defined property, wss4j.processor.map, to the WSS4JInInterceptor as shown in the following Spring example. The same configuration can be achieved through the API as well. The key value is an XML qualified name of the WS-Security header element to process with the given processor implementation. The entry values can be a String representing a class name of the processor to instantiate, an Object implementing Processor, or null to disable processing of the given WS-Security header element.

```xml
<bean class="org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor">
    <constructor-arg>
        <map>
            <!-- This reconfigures the processor implementation that WSS4J uses to process a WS-Security Signature element. -->
            <entry key="wss4j.processor.map">
                <map key-type="javax.xml.namespace.QName">
                    <entry value="my.class">
                        <key>
                            <bean class="javax.xml.namespace.QName">
                                <constructor-arg value="http://www.w3.org/2000/09/xmldsig#"/>
                                <constructor-arg value="Signature"/>
                            </bean>
                        </key>
                    </entry>
                </map>
            </entry>
        </map>
    </constructor-arg>
</bean>
```

### Custom Actions

As of CXF 2.2.6, you can specify custom WSS4J Action configurations on the WSS4JOutInterceptor. To activate this configuration option, one provides a non-WSS4J defined property, wss4j.action.map, to the WSS4JOutInterceptor as shown in the following Spring example. The same configuration can be achieved through the API as well. The key value is an integer representing the WSS4J action identifier. The entry values can be a String representing a class name of the action to instantiate or an Object implementing Action. This configuration option allows you to override built-in action implementations or add your own.
For the case that adding new custom action, if the new key int number is 12345, you must also specify new action name as string "12345".

### Configuring WS-Security Actions

#### Username Token Authentication

WS-Security supports many ways of specifying tokens. One of these is the UsernameToken header. It is a standard way to communicate a username and password or password digest to another endpoint. Be sure to review the OASIS UsernameToken Profile Specification for important security considerations when using UsernameTokens.

If a nonce is present in a UsernameToken then it should be cached by the message recipient to guard against replay attacks. This behaviour is enabled by default starting with CXF 2.6.0. This functionality is also available from Apache CXF 2.4.7 and 2.5.3 onwards, but is not enabled by default at all for backwards-compatibility reasons. The following properties control nonce caching:

- "ws-security.enable.nonce.cache" - Whether to cache UsernameToken nonces. The default value (for CXF 2.6.0) is "true" for message recipients, and "false" for message initiators. Set it to true to cache for both cases. The default value for CXF 2.4.x and 2.5.x is false. See [here](#) for more information.

- "ws-security.nonce.cache.instance" - This holds a reference to a ReplayCache instance used to cache UsernameToken nonces. The default instance that is used is the [EHCacheReplayCache](#).

- "ws-security.cache.config.file" - Set this property to point to a configuration file for the underlying caching implementation. The default configuration file is used is `cxf-ehcache.xml` in the cxf-rt-ws-security module.

For the server side, you'll want to set up the following properties on your WSS4JInInterceptor (see [above](#) for code sample):

```java
inProps.put(WSHandlerConstants.ACTION, WSHandlerConstants.USERNAME_TOKEN);
// Password type : plain text
inProps.put(WSHandlerConstants.PASSWORD_TYPE, WSConstants.PW_TEXT);
// for hashed password use:
// properties.put(WSHandlerConstants.PASSWORD_TYPE, WSConstants.PW_DIGEST);
// Callback used to retrieve password for given user.
inProps.put(WSHandlerConstants.PW_CALLBACK_CLASS, ServerPasswordHandler.class.getName());
```

The password callback class allows you to retrieve the password for a given user so that WS-Security can determine if they're authorized. Here is a small example:
import java.io.IOException;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import org.apache.wss4j.common.ext.WSPasswordCallback;

public class ServerPasswordCallback implements CallbackHandler {

    public void handle(Callback[] callbacks) throws IOException,
            UnsupportedCallbackException {

        WSPasswordCallback pc = (WSPasswordCallback) callbacks[0];

        if (pc.getIdentifier().equals("joe")) {
            // set the password on the callback. This will be compared to the
            // password which was sent from the client.
            pc.setPassword("password");
        }
    }
}

Note that for up to and including CXF 2.3.x, the password validation of the special case of a plain-text password (or any other yet unknown password type)
is delegated to the callback class. In that case, the ServerPasswordCallback should be something like the following one:

public class ServerPasswordCallback implements CallbackHandler {

    public void handle(Callback[] callbacks) throws IOException,
            UnsupportedCallbackException {

        WSPasswordCallback pc = (WSPasswordCallback) callbacks[0];

        if (pc.getIdentifier().equals("joe") {
            if (!pc.getPassword().equals("password")) {
                throw new IOException("wrong password");
            }
        }
    }
}

For CXF 2.4 onwards, the callback handler supplies the password for all cases, and the validation is done internally (but can be configured). See here for
more information.

On the Client side you'll want to configure the WSS4J outgoing properties:

outProps.put(WSHandlerConstants.ACTION, WSHandlerConstants.USERNAME_TOKEN);
// Specify our username
outProps.put(WSHandlerConstants.USER, "joe");
// Password type: plain text
outProps.put(WSHandlerConstants.PASSWORD_TYPE, WSConstants.PW_TEXT);
// for hashed password use:
//properties.put(WSHandlerConstants.PASSWORD_TYPE, WSConstants.PW_DIGEST);
// Callback used to retrieve password for given user.
outProps.put(WSHandlerConstants.PW_CALLBACK_CLASS,
        ClientPasswordHandler.class.getName());

Once again we're using a password callback, except this time instead of specifying our password on the server side, we're specifying the password we
want sent with the message. This is so we don't have to store our password in our configuration file.
import java.io.IOException;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import org.apache.wss4j.common.ext.WSPasswordCallback;

public class ClientPasswordCallback implements CallbackHandler {
    public void handle(Callback[] callbacks) throws IOException,
        UnsupportedCallbackException {
        WSPasswordCallback pc = (WSPasswordCallback) callbacks[0];
        // set the password for our message.
        pc.setPassword("password");
    }
}

In the case of multiple users with different passwords, use the WSPasswordCallback's getIdentifier() method to obtain the username of the current SOAP request.

Here is an example of WS-Security implemented using annotations for interceptors (uses UsernameToken).

**WS-Security UsernameToken and Custom Authentication**

If needed, one may want to configure a javax.ws.security.validate.token property set to false and register a custom org.apache.cxf.interceptor.security.AbstractUsernameTokenInInterceptor implementation for using a WSS4J UsernameToken wrapped in a CXF specific UsernameToken for the custom authentication and Subject creation. The JAASLoginInterceptor will also recognize a CXF UsernameToken and thus can be used instead of the custom org.apache.cxf.interceptor.security.AbstractUsernameTokenInterceptor. (Prior to CXF 2.4.0, use "ws-security.ut.no-callbacks" instead of "ws-security.validate.token" with the value of true instead of false to postpone the validation of the token.)

**Using X.509 Certificates**

The X.509 Certificate Token Profile (pdf) provides another option for implementing WS-Security. For the Signature and Encryption actions, you'll need to create a public & private key for the entities involved. You can generate a self-signed key pair for your development environment via the following steps. Keep in mind these will not be signed by an external authority like Verisign, so are inappropriate for production use.

1. Creating private key with given alias and password like "myAlias"/"myAliasPassword" in keystore (protected by password for security reasons)

```bash
keytool -genkey -alias myAlias -keypass myAliasPassword -keystore \
privatestore.jks -storepass keyStorePassword -dname "cn=myAlias" -keyalg RSA
```

The alias is simply a way to identify the key pair. In this instance we are using the RSA algorithm.

2. Self-sign our certificate (in production environment this will be done by a company like Verisign).

```bash
keytool -selfcert -alias myAlias -keystore privatestore.jks \
-storepass keyStorePassword -keypass myAliasPassword
```

3. Export the public key from our private keystore to file named key.rsa

```bash
keytool -export -alias myAlias -file key.rsa -keystore privatestore.jks \
-storepass keyStorePassword
```

4. Import the public key to new keystore:

```bash
keytool -import -alias myAlias -file key.rsa -keystore publicstore.jks \
-storepass keyStorePassword
```

So now we have two keystores containing our keys - a public one (publicstore.jks) and a private one (privatestore.jks). Both of them have keystore password set to keyStorePass (this not recommended for production but ok for development) and alias set to myAlias. The file key.rsa can removed from filesystem, since it used only temporarily. Storing keys in keystores is strongly advised because a keystore is protected by a password.
A more detailed description of key generation can be found here:
http://java.sun.com/javase/6/docs/technotes/tools/solaris/keytool.html

How to create a production certificate can be found here:
http://support.globalsign.net/en/objectsign/java.cfm

**Signing**

Signing a message is used to validate to the recipient that the message could only have come from a certain sender, and that the message was not altered in transit. It involves the sender encrypting a digest (hash) of the message with its private key, and the recipient decrypting the hash with the sender's public key, and recalculating the digest of the message to make sure the message was not altered in transit (i.e., that the digest values calculated by both the sender and recipient are the same). For this process to occur you must ensure that the Client's public key has been imported into the server's keystore using keytool.

On the client side, our outgoing WS-Security properties will look like so (see above for code sample):

```java
outProps.put(WSHandlerConstants.ACTION, "Signature");
outProps.put(WSHandlerConstants.USER, "myAlias");
outProps.put(WSHandlerConstants.PW_CALLBACK_CLASS, ClientCallbackHandler.class.getName());
outProps.put(WSHandlerConstants.SIG_PROP_FILE, "client_sign.properties");
```

The USER that is specified is the key alias for the client. The password callback class is responsible for providing that key's password.

**Tip**

For X.509 support you will normally have multiple actions, e.g. Encryption with Signature. For these cases, just space-separate the actions in the ACTION property as follows:

```java
outProps.put(WSHandlerConstants.ACTION, 
            WSHandlerConstants.TIMESTAMP + " " + 
            WSHandlerConstants.SIGNATURE + " " + 
            WSHandlerConstants.ENCRYPT);
```

Alternatively, you may space-separate the string literals you see above in the Spring configuration (e.g., "Signature Encrypt")

Our client_sign.properties file contains several settings to configure WSS4J:

```properties
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=keyStorePassword
org.apache.ws.security.crypto.merlin.keystore.alias=myAlias
org.apache.ws.security.crypto.merlin.keystore.file=client_keystore.jks
```

On the server side, we need to configure our incoming WSS4J interceptor to verify the signature using the Client's public key.

```java
inProps.put(WSHandlerConstants.ACTION, "Signature");
inProps.put(WSHandlerConstants.SIG_PROP_FILE, "server.properties");
```

Our server_sign.properties file contains several settings to configure WSS4J:

```properties
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=amex123
org.apache.ws.security.crypto.merlin.keystore.file=server_keystore.jks
```

**Encryption**

Encryption involves the sender encrypting the message with the recipient's public key to ensure that only the recipient can read the message (only the recipient has its own private key, necessary for decrypting the message.) This requires the sender to have the recipient's public key in its keystore.

The process for encrypting is very similar to and indeed usually combined with the signature process above. Our WS-Security test sample (svn checkout http://svn.apache.org/repos/asf/cxf/trunk/systests/ws-security/src/test/java/org/apache/cxf/systest/ws/security/) provides an example of encrypting requests and responses, also check this blog entry for a more end-to-end example showing signature and encryption of both SOAP requests and responses.
Security Best Practices

Some guidelines are given at the WSS4J website about best practices when using WS-Security. It is recommended to study the guidelines carefully.