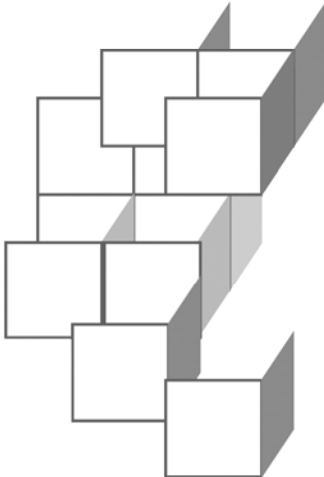
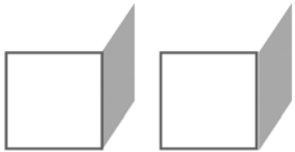




CHAPTER 5
WS-SECURITY



OBJECTIVES

After completing “WS-Security,” you will be able to:

- Describe the usefulness of the WS-Security specification and the specific profiles for security tokens in SOAP headers.
- Use the XML Web Services Security APIs to implement message-level security with various token types and features:
 - Username tokens
 - Digital signature and verification
 - Encryption and decryption

The WS-Security Specifications

- **WS-Security** is the earliest and most successful product from OASIS – and in fact it is a suite of several products.
- There is a core specification called **SOAP Message Security**.
- This defines the basic model of embedding **security tokens** in SOAP message headers.
- There are several **profiles** that refine the data model for what sorts of content makes up those tokens:
 - The **Username Token Profile**
 - The **X.509 Token Profile**
 - The **SAML Token Profile**
 - The **Kerberos Token Profile**
 - The **Rights Expression Language Token Profile**
- We'll see examples of the first three of these as we get started using XWSS later in this chapter.
- A separate specification treats the issues involved in securing a SOAP message that carries attachments.

A Secure SOAP Message

EXAMPLE

- This example, from the specification itself, shows a minimal use of security tokens in a SOAP message:

```

<S:Envelope xmlns:S="..." xmlns:ds="...">
  <S:Header>
    <wsse:Security xmlns:wsse="...">
      <wsse:UsernameToken Id="MyID">
        <wsse:Username>Zoe</wsse:Username>
      </wsse:UsernameToken>
      <ds:Signature>
        <ds:SignedInfo>
          <ds:CanonicalizationMethod
            Algorithm="..." />
          <ds:SignatureMethod Algorithm="..." />
          <ds:Reference URI="#MsgBody">
            <ds:DigestMethod Algorithm="..." />
            <ds:DigestValue>LyLsF0Pi4wPU...
          </ds:DigestValue>
          </ds:Reference>
        </ds:SignedInfo>
        <ds:SignatureValue>DJbchm5gK...
      </ds:SignatureValue>
      <ds:KeyInfo>
        <wsse:SecurityTokenReference>
          <wsse:Reference URI="#MyID"/>
        </wsse:SecurityTokenReference>
      </ds:KeyInfo>
    </ds:Signature>
  </wsse:Security>
</S:Header>
<S:Body Id="MsgBody">
  <tru:StockSymbol xmlns:tru="..." >
    QQQ
  </tru:StockSymbol>
</S:Body>
</S:Envelope>

```

Container vs. Component

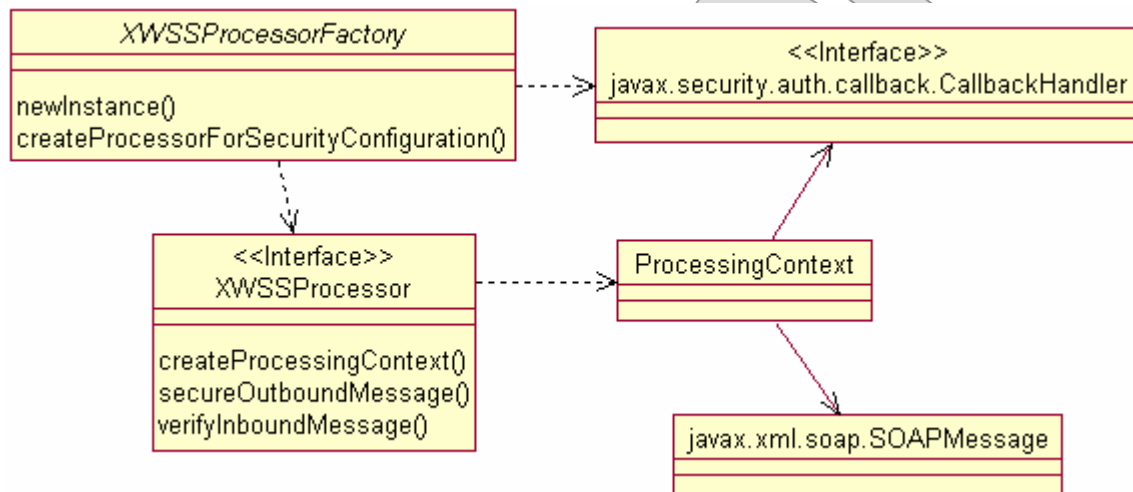
- **From an architectural standpoint, common strategies for managing SOAP message security are still evolving.**
 - WS-Security is a demanding specification, and any secure-messaging solution will be intricate.
 - It is also difficult to generalize a solution, since the point of message-level security (as opposed to the broad brush of HTTPS) is that different operations between different endpoints will require different features.
- **Many Java EE products roll in some sort of WSS support.**
 - Naturally, for container-managed security, one usually trades off a significant degree of control over details: which messages or parts of messages are signed or encrypted; what keys or certificates are used for what sorts of operations, etc.
 - Features tend to be proprietary, often resulting in vendor lock-in.
- **Often, at some point, application code has to get involved.**
- **We'll pursue this approach, letting the application code control message security and not asking the container for any help.**
 - This is useful for purposes of learning the details of WS-Security.
 - It also develops skills that are likely to be practical at some point down the road – whether or not the specific tools and APIs we use in these exercises are the ones in play elsewhere.

The XML Web Services Security Project

- One of the leading portable WS-Security implementations is the **XML Web Services Security** project, or **XWSS**.
 - This is part of the **Java Web Services Developer Pack**.
 - The **JWSDP** is in turn being folded into **Project GlassFish**, an open-source J2EE application server.
- **XWSS** supports several XML standards:
 - **WS-Security**, including the **Username, X.509, SAML**, and **SwA profiles**. (X.509 and SwA support is incomplete.)
 - **XML Signature**, by incorporating the JSR105 RI.
 - **XML Encryption**, by incorporating Apache XML Security.
- **XWSS** is not a standard; there is no JSR associated with it.
- It is closely aligned with several JSRs and implemented in a style very similar to other JCP standards, such as the JCE, Java XML Digital Signature API, and JAX-RPC itself.
 - In fact it folds an **enhanced JAX-RPC compiler** into the development environment, to provide nearly transparent service to JAX-RPC services and clients.
 - It also supports **SAAJ** services and clients directly.
- Find the XWSS project at
<https://xwss.dev.java.net>
- We'll spy on the message traffic between a few XWSS implementations, and let them give us a better vantage on WS-Security itself.

The XWSSProcessor Interface

- All XWSS message operations occur through a three-step process starting with a **com.sun.xml.wss.XWSSProcessorFactory**:



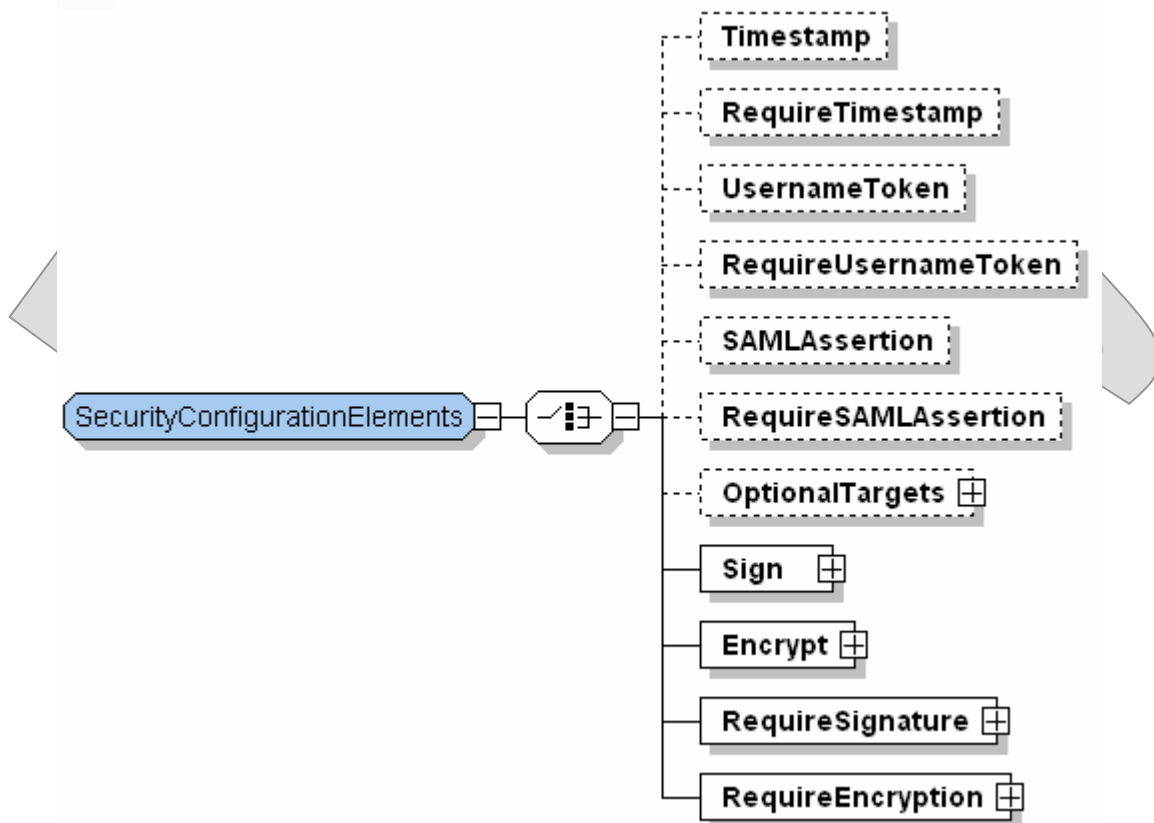
- The factory object can spin itself up in the usual way, with the static method **newInstance**.
- Then it can create a new **XWSSProcessor**, based on an XML configuration file that directs the processor to carry out various operations on SOAP messages: signature, encryption, and adding or checking other WS-Security headers.
- The processor operates on individual messages with the help of a **ProcessingContext**.

Configuration Schema

- Configuration information is supplied to the processor in an XML format governed by the namespace URI

`http://java.sun.com/xml/ns/xwss/config`

- The schema, strangely, is not available as a standalone document within the XWSS distribution.
 - Instead it is embedded in an appendix to the JWSDP tutorial.
 - It is provided as part of your lab file image, in `Examples/Schema/XWSS.xsd`.
- The `<SecurityConfiguration>` element calls out which message-security features are required of the processor:



Configuration Files

EXAMPLE

- A few examples of sender configurations follow:

- To supply a username token:

```
<SecurityConfiguration ... >
  <UsernameToken name="me" password="mypwd" />
</SecurityConfiguration>
```

- To sign the message body:

```
<SecurityConfiguration ... >
  <Sign />
</SecurityConfiguration>
```

- To sign specific parts of the message:

```
<SecurityConfiguration ... >
  <Sign>
    <SignatureTarget
      type="xpath"
      value="//Target[@attr='expected']"
      contentOnly="false"
    />
    <SignatureTarget type="qname"
      value="{myNS}myElem" />
  </Sign>
</SecurityConfiguration>
```

- To encrypt the message body:

```
<SecurityConfiguration ... >
  <Encrypt>
    <X509Token certificateAlias="yourCert" />
  </Encrypt>
</SecurityConfiguration>
```

Configuration Files

EXAMPLE

- For the message receiver, we usually declare requirements:

```
<SecurityConfiguration ... >  
  <RequireUsernameToken />  
  <RequireSignature />  
  <RequireEncryption />  
</SecurityConfiguration>
```

- **Either service or client can use any of these.**
 - For example a service may both require signed requests and sign its own responses.
- **Also, not every detail that's needed for a given WSS feature will be supplied by the configuration.**
 - Some information must be derived dynamically.
 - So the XWSS processor will ask the application to fill in the blanks, via a callback interface.
- **Some information is optional: if it's provided in the config, the processor won't ask for it via the callback interface; if it isn't provided, the callback handler will have to be ready to provide it.**

JAAS

- The **Java Authentication and Authorization Service**, or **JAAS**, addresses several needs in Java application security:
 - It integrates the concept of a **Subject** with existing code-authorization policy architecture, so that users as well as code bases can be authorized to perform work in the system.
 - It provides the necessary abstractions of authentication logic, and a runtime-configuration mechanism, so that different **LoginModules** can be plugged in to an application
 - It abstracts the application's or user's role in the login process to a **CallbackHandler**, so that the same **LoginModule** might be used in very different styles, from command-line arguments for user name and password to dialog boxes to browser-based authentication over HTTP.
- Increasingly, security solutions are consolidating around JAAS
 - especially those having to do with authentication processes.

Interface Model

- The primary roles in JAAS are shown here:

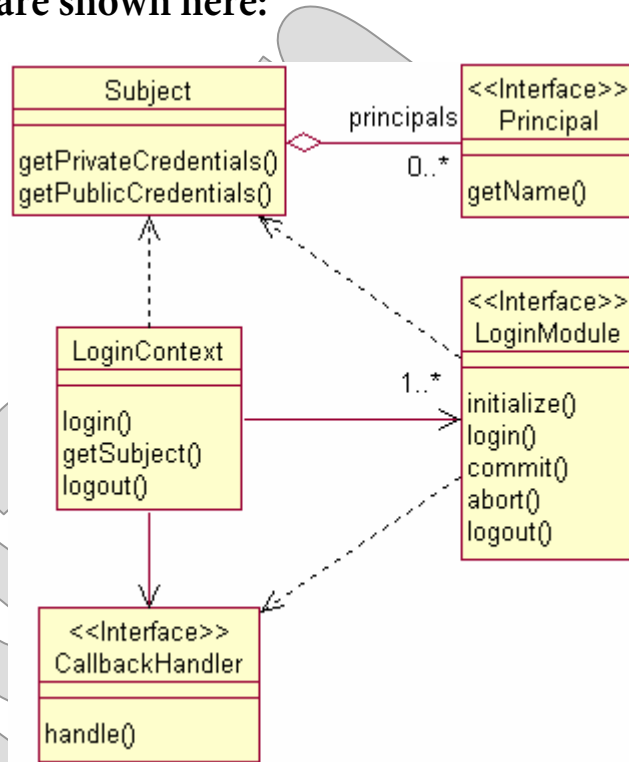
- **Subject** is a logged-in user, or perhaps remote software that has performed automated login – basically any entity that can be authorized to do work in the system at runtime.

- To derive a **Subject**, an application creates a **LoginContext** and asks it to perform a login. **LoginContext** is the

broker that connects the parties that participate in login, and at the same time keeps them isolated and hence pluggable/replaceable.

- Based on runtime configuration, **LoginContext** locates one or more **LoginModule** implementations. These know the ins and outs of a particular authentication system and, in the case of successful login, act as factories for **Subjects**.

- Typically, the **LoginModule** will require further information. It will request this information by passing **Callbacks** to a **CallbackHandler** that was originally supplied via the **LoginContext**.



The CallbackHandler Interface

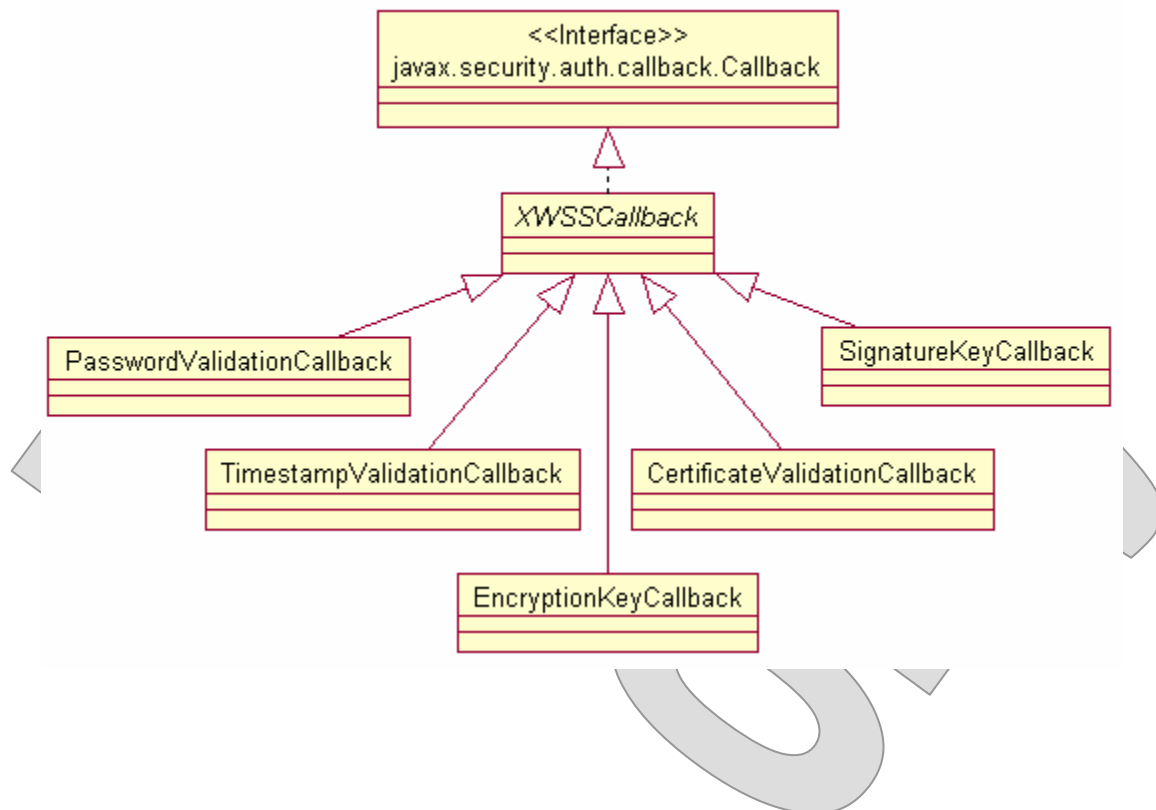
- Many **LoginModules** will require additional information from the application in order to make an authentication decision.
- For this purpose the application supplies a **CallbackHandler** when it creates the **LoginContext**, and this is passed along to **LoginModule.initialize**.

```
public interface CallbackHandler
{
    public void handle (Callback[] callbacks)
        throws UnsupportedOperationException;
}
```

- The **LoginModule** can then initiate a conversation with the handler, by calling **handle** and passing an array of **Callback** objects.
 - This may happen not at all, or once, or several times in series – which is why we call it a conversation.
 - Most often there will be a single call to **handle**, with all information requests batched up so that the handler can fetch everything in one shot.
- Each **Callback** object is an “in/out parameter:” it is meant to carry both the request and the response.
 - So the handler will set requested information into the **Callback**, rather than returning values from the method call.

XWSS Callback Types

- XWSS relies on JAAS for its callback architecture.
- The XML configuration file gives the processor broad marching orders: sign this, encrypt that, etc.
- The processor then relies on a JAAS **CallbackHandler** and a package of **Callback** types to carry out a specific operation that relates to specific keys, certificates, passwords, etc.
 - The callback types – a partial list is represented below – are found in package **com.sun.xml.wss.impl.callback**:



Where to Embed XWSS Processing

- XWSS is designed to be integrated in one of several ways.
- The fundamental behavior of the XWSSProcessor can be invoked by any piece of Java code – even a standalone application modifying local files.
- Practical options for web services and clients are:
 - An **SAAJ** application can invoke XWSS processing most simply, since the **ProcessingContext** wraps an SAAJ **SOAPMessage**.
 - **JAX-RPC** services can plug into this interface by way of **message handlers**, which are given a simple path to the request and response messages, again as SAAJ **SOAPMessages**.
 - XWSS provides an **enhanced JAX-RPC compiler** that can produce stubs and ties that automatically trigger XWSS processing.
- For most, the middle option is the most attractive.
 - The enhanced compiler and “smart stubs” do automate the process, potentially saving time and trouble.
 - The compiler only works with proprietary configuration files designed for use with Sun’s application server; neither the compilation process nor its products are usable on other servers.
 - This is a show-stopper for just about everyone.
- So integrating XWSS via a JAX-RPC message handler offers the best blend of convenience, reusability, and portability.

A Reusable Handler

EXAMPLE

- A great deal of complex logic goes into the JAAS callback handler.
- We'll use a prepared handler class **WSSMessageHandler** that's largely configurable by properties files, XML configurations, and keystores.
- We'll see this class in use in the upcoming example – here's a bit of its **init** method:

```
props.load
(WSSMessageHandler.class.getResourceAsStream
 ("security.properties"));

keystore = KeyStoreUtil.getKeyStore
(WSSMessageHandler.class.getResourceAsStream
 (props.getProperty ("keystore")),
 props.getProperty ("storepass"));

String userDBFilename =
 props.getProperty ("userDB");
if (userDBFilename != null)
 userDB.load
 (WSSMessageHandler.class.getResourceAsStream
 (userDBFilename));

xwss = XWSSProcessorFactory.newInstance ()
 .createProcessorForSecurityConfiguration
 (WSSMessageHandler.class.getResourceAsStream
 (props.getProperty ("security-config")),
 this);
```

A Reusable Handler

EXAMPLE

- The **handleInbound** method derives a reference to a SOAP message, creates a processing context around it, and triggers XWSS processing:

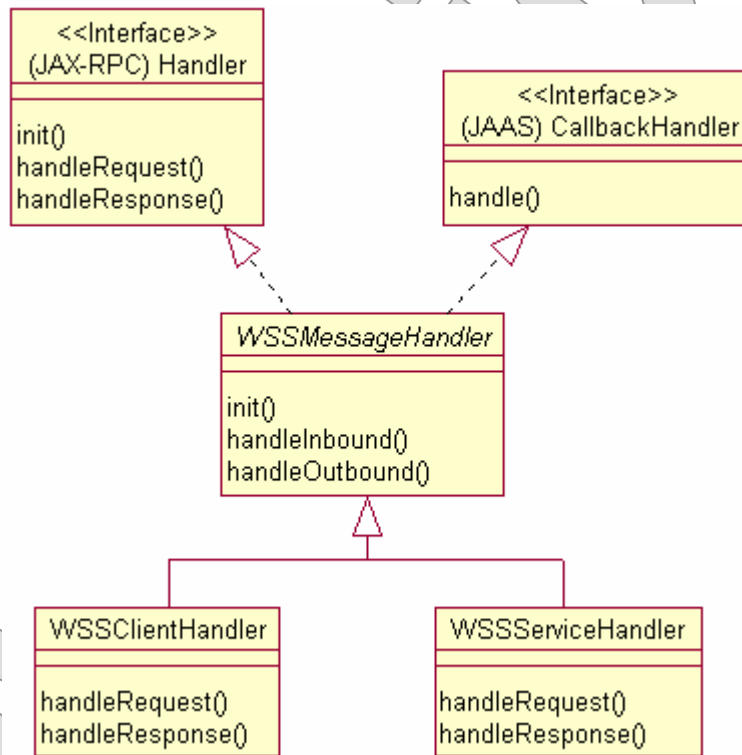
```
public boolean handleInbound
    (MessageContext context)
    throws JAXRPCException
{
    SOAPMessage msg =
        ((SOAPMessageContext) context).getMessage ();
    try
    {
        ProcessingContext xwssContext =
            xwss.createProcessingContext (msg);
        xwss.verifyInboundMessage (xwssContext);
    }
    ...
}
```

- **handleOutbound** does more or less the same thing, only calling **secureOutboundMessage**.

A Reusable Handler

EXAMPLE

- Since XWSS thinks in terms of “inbound” and “outbound” messages, we need to connect it to requests and responses.
 - This will be done differently for clients vs. services.
 - We therefore subclass the main handler two different ways:



- Services and clients can use these two subclasses directly, or can further subclass them for specific functionality.

Configuring Message Security

EXAMPLE

- In **Examples/Pizza/Step4**, the service and client have been reconfigured to use XWSS, even in favor of our lovingly-built digital-signature handlers.
 - The service is configured to use `cc.wss.WSSServiceHandler` instead of `cc.wss.ValidateHandler`.
 - The client uses `cc.wss.WSSClientHandler`.
- The starter version uses XWSS, but is configured to apply no security features to the messages.
 - Both `security.properties` files state:
`security-config=NoSecurity.xml`
- Build and test the service and client, passing requests through the SOAPSniffer, and you'll see messages much like the original, un-secured version of the service.
- But we have a platform now that allows us to quickly re-configure our message security, as we'll see on the next few pages.

Configuring Message Security

EXAMPLE

- In the **Caller** directory, the configuration file **Username.xml** directs the processor to affix a WS-Security username-token header entry to the request:

```
<SecurityConfiguration ... >
  <UsernameToken
    name="topping" password="anchovy"
  />
</SecurityConfiguration>
```

- Edit **security.properties** and set this as the XWSS configuration:

```
security-config=Username.xml
```

- Rebuild the client, and test it out:

```
asant
```

```
run http://localhost:8079/Pizza/Delivery
```

```
Ordering pizza ... 2 pizzas ordered.
```

- The request carries a username token, as shown on the following page.
- Notice that the password is hashed, and that the resulting digest is tied not only to the password but also to a nonce generated by XWSS on the client's behalf.

Configuring Message Security

EXAMPLE

- See also the result captured in **SOAP/XWSS_Username.txt**.

```
<env:Header>
  <wsse:Security
    xmlns:wsse="..."
    env:mustUnderstand="1"
  >
    <wsse:UsernameToken
      xmlns:wsu="..."
      wsu:Id="XWSSGID-11674041947502008040267"
    >
      <wsse:Username>topping</wsse:Username>
      <wsse:Password
        Type="...#PasswordDigest"
      >ytEf/2VzWCQ7fuPF85lkXGD/vD8=</wsse:Password>
      <wsse:Nonce
        EncodingType="...#Base64Binary"
      >CrBibhDAY8mkxJ43rHXKoeH/</wsse:Nonce>
      <wsu:Created>2006-12-29T14:56:35Z
        </wsu:Created>
      </wsse:UsernameToken>
    </wsse:Security>
  </env:Header>
```

Configuring Message Security

EXAMPLE

- Let's re-establish our signature requirement: the configuration file **RequireSignature.xml** in the **Delivery** directory tells the processor to insist on a WSS digital-signature header:

```
<SecurityConfiguration ... >
  <RequireSignature />
</SecurityConfiguration>
```

- Edit **security.properties** to use this configuration:

```
security-config=RequireSignature.xml
```

- Build and deploy the service with the new configuration:

```
asant
```

- Test again from the client side, and see that the request is refused – no surprise, since it's not signed:

```
run
```

```
javax.xml.rpc.soap.SOAPFaultException:
  Message security failure
```

- Change the client side to use **Sign.xml** as its configuration – this file is shown below.

```
<SecurityConfiguration ... >
  <Sign />
</SecurityConfiguration>
```

- Rebuild the client and test again, and it works – the signature header can be seen in SOAPSniffer and is shown on the next page.

Configuring Message Security

EXAMPLE

- See also **SOAP/XWSS_Signature.txt**.

```

<env:Header>
  <wsse:Security env:mustUnderstand="1" >
    <wsse:BinarySecurityToken wsu:Id="...472184233"
      ValueType="...x509-token-profile-1.0#X509v3"
    >MIIC3TCCAkag...</wsse:BinarySecurityToken>
    <ds:Signature >
      <ds:SignedInfo>
        <!-- exc-c14n, rsa-sha1 -->
        <ds:Reference URI="...1902108226" >
          <ds:DigestMethod Algorithm="...sha1" />
          <ds:DigestValue>5m0...</ds:DigestValue>
        </ds:Reference>
        <ds:Reference URI="...2113622996" >
          <ds:DigestMethod Algorithm="...sha1" />
          <ds:DigestValue>ixy...</ds:DigestValue>
        </ds:Reference>
      </ds:SignedInfo>
      <ds:SignatureValue>a...</ds:SignatureValue>
      <ds:KeyInfo>
        <wsse:SecurityTokenReference wsu:Id="..." >
          <wsse:Reference URI="...472184233"
            ValueType="...x509-token-profile..." />
        </wsse:SecurityTokenReference>
      </ds:KeyInfo>
    </ds:Signature>
    <wsu:Timestamp wsu:Id="...2113622996" >
      <wsu:Created>...T21:50:34Z</wsu:Created>
      <wsu:Expires>...T21:50:39Z</wsu:Expires>
    </wsu:Timestamp>
  </wsse:Security>
</env:Header>
<env:Body wsu:Id="...1902108226" >
  ...

```

Configuring Message Security

EXAMPLE

- Notice that this signature header entry does not state the signing key directly.
 - It refers to external key information contained in a separate WSS header entry.
 - This entry is the WSS-standard X.509 certificate entry, which allows the message to carry not just the key information but the full certificate.
 - This gives the service the data it needs to establish not only message integrity but also the authenticity of the signer, should it want to do so.
- The signature refers to two pieces of signed information: the message body, and also a generated timestamp.
 - The timestamp is yet another WSS standard header entry.
 - Signing the timestamp along with the message payload helps to protect against **replay attacks**, because the message content is married to the time at which the message was sent.

Configuring Message Security

EXAMPLE

- Try one last pair of configurations:
 - **RequireEncryptionAndSignature.xml** on the service side.

```
<SecurityConfiguration ... >
  <RequireEncryption />
  <RequireSignature />
</SecurityConfiguration>
```

- **SignAndEncrypt.xml** on the client side.

```
<SecurityConfiguration ... >
  <Sign/>
  <Encrypt>
    <X509Token certificateAlias="Friend" />
  </Encrypt>
</SecurityConfiguration>
```

- The resulting message adheres to two specifications, separately:
 - The WS-Security X.509 Token Profile, again, is used to convey certificate data: the recipient's **public key** in this case.
 - The XML Encryption header provides the **secret key** with which the content was encrypted – this is why it refers both to the X.509 certificate token and to the body content.
 - The body content is replaced by an **<EncryptedData>** element.
- The resulting message is captured in **SOAP/XWSS_SignatureAndEncryption.txt**.
- It is a bit too complex to show profitably on the printed page; but on the following page is a typical message that has been encrypted, but not signed.

Configuring Message Security

EXAMPLE

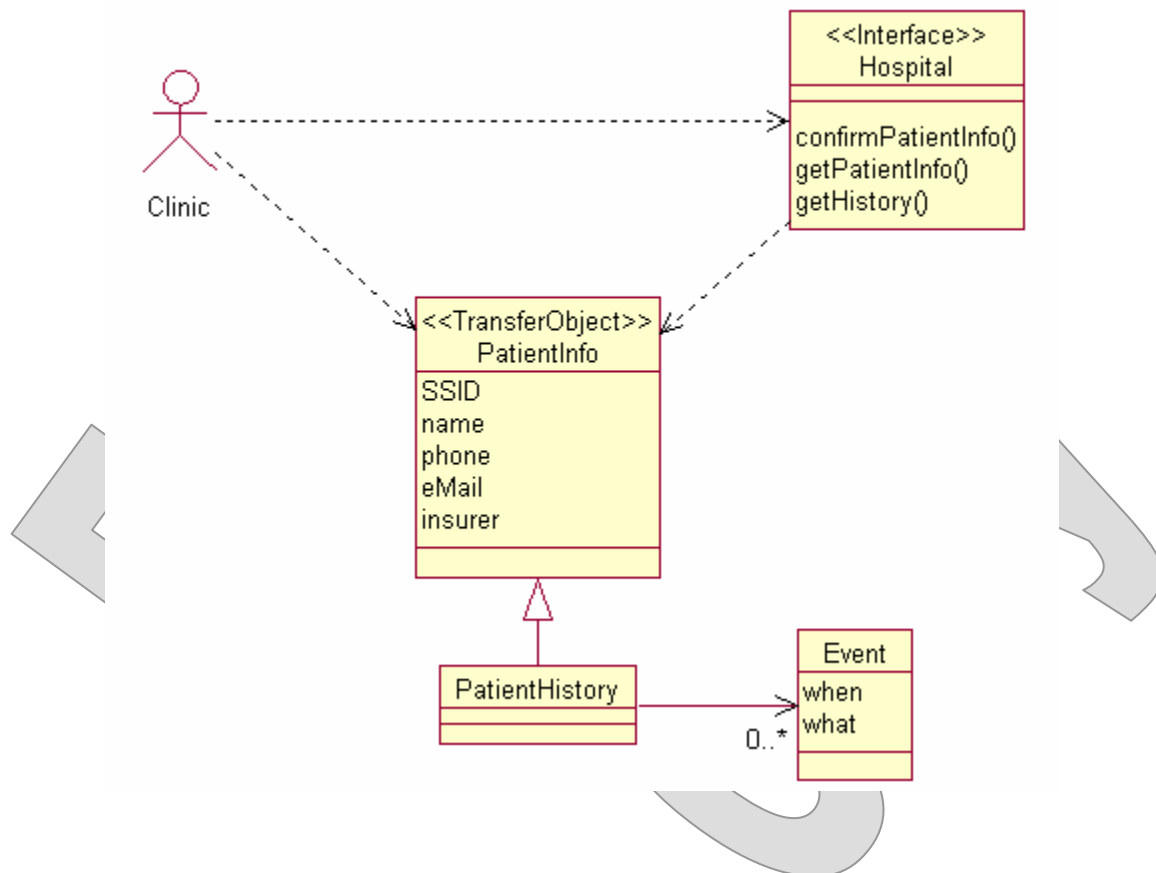
```

<env:Header>
  <wsse:Security env:mustUnderstand="1" >
    <wsse:BinarySecurityToken
      ValueType="...x509-token-profile-1.0#X509v3"
      wsu:Id="...1752808926"
    >MIICzjCCAjeg...</wsse:BinarySecurityToken>
    <xenc:EncryptedKey>
      <xenc:EncryptionMethod Algorithm=".rsa-1_5" />
      <ds:KeyInfo>
        <wsse:SecurityTokenReference>
          <wsse:Reference URI="...1752808926"
            ValueType="...x509-token-profile-1.0.."
          />
        </wsse:SecurityTokenReference>
      </ds:KeyInfo>
      <xenc:CipherData>
        <xenc:CipherValue>Y2g...</xenc:CipherValue>
      </xenc:CipherData>
      <xenc:ReferenceList>
        <xenc:DataReference URI="...219991348" />
      </xenc:ReferenceList>
    </xenc:EncryptedKey>
  </wsse:Security>
</env:Header>
<env:Body>
  <xenc:EncryptedData Id="...219991348"
    Type="...xmlenc#Content" >
    <xenc:EncryptionMethod
      Algorithm="...xmlenc#tripleDES-cbc" />
    <xenc:CipherData>
      <xenc:CipherValue>QC0Dq...</xenc:CipherValue>
    </xenc:CipherData>
  </xenc:EncryptedData>
</env:Body>

```

The Healthcare Case Study

- For this and the next few chapters, we'll work at securing the messaging involved in a small case study.
- A web service is published by a medical facility – it could be any facility but our service instance will represent a **Hospital** – to make patient data available for transaction with other facilities.
- The SOAP interface published by the Hospital makes various medical records available:



Baseline SOAP Messaging

EXAMPLE

- In **Examples/Healthcare/Step1**, we can build and test out the Hospital and Clinic applications.
- First, let's get everything started, built and deployed:
 - Start your J2EE server.
 - Run **asant** from the **Hospital** subdirectory; this will build and deploy the web service.
 - Run **asant** from the **Clinic** subdirectory to build the Java SE application that includes a JAX-RPC stub for the service.
 - From **Examples/SOAPSniffer**, run the **Sniff** script. In fact you may want to leave the sniffer running for the rest of this chapter, including the upcoming lab.
- Now, from the **Clinic** directory, you can send various canned SOAP messages to the web service.
 - Results are shown on the following pages.
 - Below each command is shown some or all of the resulting SOAP traffic that passes through the SOAPSniffer.

Baseline SOAP Messaging

EXAMPLE

```
run confirm http://localhost:8079/Hospital/Records
Confirming patient info ... confirmed.
```

Messages

(see this also in `SOAP/confirmPatientInfoXXX.txt`)

```
<env:Envelope ... >
  <env:Body>
    <ns0:confirmPatientInfo>
      <info>
        <SSID>123456789</SSID>
        <name>Jane Q. Private</name>
        <phone>564-564-5646</phone>
        <eMail>jane@privacy.us</eMail>
        <insurer>Insurance-R-Us</insurer>
      </info>
    </ns0:confirmPatientInfo>
  </env:Body>
</env:Envelope>

<env:Envelope ... >
  <env:Body>
    <ns0:confirmPatientInfoResponse>
      <result>>true</result>
    </ns0:confirmPatientInfoResponse>
  </env:Body>
</env:Envelope>
```

Baseline SOAP Messaging

EXAMPLE

```
run getInfo http://localhost:8079/Hospital/Records
```

```
Getting patient info ...
```

```
SSID      : 011235813
Name      : Leo P. Fibonacci
Phone     : 213-455-8914x4233
EMail     : leo@fibonacci.it
Insurer  : Imaginary-I
```

Messages

(see this also in **SOAP/getPatientInfoXXX.txt**)

```
<env:Envelope ... >
```

```
<env:Body>
```

```
<ns0:getPatientInfo>
```

```
<SSID>011235813</SSID>
```

```
</ns0:getPatientInfo>
```

```
</env:Body>
```

```
</env:Envelope>
```

```
<env:Envelope ... >
```

```
<env:Body>
```

```
<ns0:getPatientInfoResponse>
```

```
<info>
```

```
<SSID>011235813</SSID>
```

```
<name>Leo P. Fibonacci</name>
```

```
<phone>213-455-8914x4233</phone>
```

```
<eMail>leo@fibonacci.it</eMail>
```

```
<insurer>Imaginary-I</insurer>
```

```
</info>
```

```
</ns0:getPatientInfoResponse>
```

```
</env:Body>
```

```
</env:Envelope>
```

Baseline SOAP Messaging

EXAMPLE

```
run getHistory
```

```
http://localhost:8079/Hospital/Records
```

```
Getting patient history ...
```

```
SSID      : 987654321
Name      : Michael Prim
Phone     : 789-789-7897
EMail     : mprim@that90scompany.com
Insurer  : BigAndBlueish
History:
  May      4, 2002 -- Pulled hamstring
  September 11, 2004 -- Severe headaches
  March    9, 2005 -- Dislocated shoulder
  March    9, 2005 -- Deep cuts to left hand,
                    minor nerve damage
  December 31, 2006 -- Exccema
```

Messages

(see this also in `SOAP/getHistoryXXX.txt`)

```
<env:Envelope ... >
  <env:Body>
    <ns0:getHistory>
      <SSID>987654321</SSID>
    </ns0:getHistory>
  </env:Body>
</env:Envelope>
```

Baseline SOAP Messaging

EXAMPLE

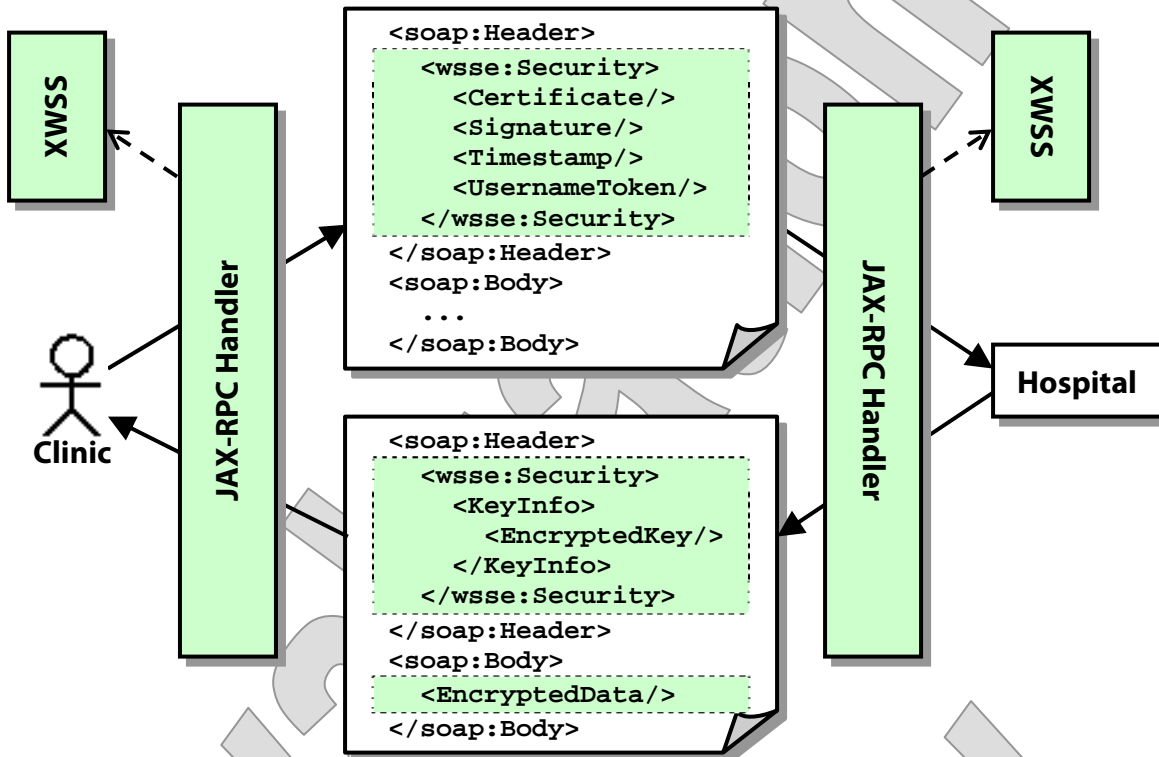
```
<env:Envelope ... >
  <env:Body>
    <ns0:getHistoryResponse>
      <history
        xsi:type="ns0:History"
      >
        <SSID>987654321</SSID>
        <name>Michael Prim</name>
        <phone>789-789-7897</phone>
        <eMail>mprim@that90scompany.com</eMail>
        <insurer>BigAndBlueish</insurer>
        <event>
          <when>2002-05-04</when>
          <what>Pulled hamstring</what>
        </event>
        <event>
          <when>2004-09-11</when>
          <what>Severe headaches</what>
        </event>
        <event>
          <when>2005-03-09</when>
          <what>Dislocated shoulder</what>
        </event>
        <event>
          <when>2005-03-09</when>
          <what>Deep cuts to left hand, minor nerve
damage</what>
        </event>
        <event>
          <when>2006-12-31</when>
          <what>Excema</what>
        </event>
      </history>
    </ns0:getHistoryResponse>
  </env:Body>
</env:Envelope>
```

Security Issues

- **To say the least, there's a need for security here!**
 - Patient information ranges from the relatively public – phone number and email address – to obviously sensitive stuff such as the patient's full medical history.
 - If the hospital puts this out on a public web server with no security measures, they're fairly certain to be sued (and successfully).
- **Our actual service client is in fact trustworthy: a healthcare Clinic with perfectly legitimate needs for patient data.**
- **We'll gradually add security features to the service side of the case study – while improving the client side to keep up.**
 - We want the caller to authenticate with a **username token**.
 - We'll want **message signatures** to assure that the token and other information in the message really came from the alleged client.
 - We'll insist on **timestamps** to guard against replay attacks and other varieties of treachery.
 - We'll **encrypt** our responses, so that outside parties to the conversation can't snoop out the patient data.

Healthcare Message Security

- Here's a master diagram of the interactions we want to see:



Message Security for the Hospital

LAB 5A

Suggested time: 30 minutes

In this lab you will begin to implement message security for the healthcare case study. You'll configure a JAX-RPC handler for the Hospital service and configure it to require username tokens, as a starting point. You'll test your service using canned messages sent from the SOAPPad, and in the following lab you'll bring the Clinic up to speed with its own JAX-RPC/XWSS handler.

Detailed instructions are found at the end of the chapter.

Evaluated Only

Message Security for the Clinic

LAB 5B

Suggested time: 45 minutes

In this lab you will integrate XWSS support into the Clinic client application, and re-establish communications with the Hospital. You'll then be able to tune up the configurations on both sides, until you have all the desired security features in place.

Detailed instructions are found at the end of the chapter.

Evaluation
Only

WSS4J

- Apache offers an alternative solution called **WSS4J**.
- WSS4J is portable to any JAX-RPC or SAAJ framework, but it is primarily dedicated to Axis web services.
- It offers a similar range of features to XWSS, but
 - There is **no inversion of control** as with XWSS and JAX-RPC: one must write the handler code to trigger signature, encryption, and the generation of security headers and tokens.
 - That is, WSS4J is more in the tradition of a **pure object library**, and not a runtime that carries out tasks on behalf of configured components.

SUMMARY

- **WS-Security provides a simple framework for including security tokens with a SOAP message.**
- **The token profiles then give specifics of various token types.**
- **Digital signature and encryption content can be placed as part of a WS-Security header as well:**
 - We observed the `<ds:Signature>` and `<enc:EncryptedKey>` elements in message content generated by XWSS.
- **XWSS offers a nice range of features for WS-Security:**
 - Signature, encryption, and several token profiles
 - An XML configuration model
 - Fine-grained selection of signed and encrypted content
 - JAAS-based callbacks to provide keys and other specific data
 - Injection of message security into existing JAX-RPC systems, with no recoding necessary for service endpoints, and minimal handler coding required

Message Security for the Hospital

LAB 5A

In this lab you will begin to implement message security for the healthcare case study. You'll configure a JAX-RPC handler for the Hospital service and configure it to require username tokens, as a starting point. You'll test your service using canned messages sent from the SOAPPad, and in the following lab you'll bring the Clinic up to speed with its own JAX-RPC/XWSS handler.

Lab workspace:	Labs/Lab5A
Backup of starter code:	Examples/Healthcare/Step1
Answer folder(s):	Examples/Healthcare/Step2
Files:	Hospital/ docroot/WEB-INF/webservices.xml Hospital/security.properties Hospital/RequireUsername.xml (to be created) SOAP/UsernameToken.txt

Instructions:

1. If you haven't yet, build and test the starter version of the Hospital service and Clinic client, and see the requests and responses for the three operations it supports by way of the SOAPSniffer.
2. Open **webservices.xml** and add a handler configuration, to bring **WSSServiceHandler** into play:

```

<service-impl-bean>
  <servlet-link>MedicalRecords</servlet-link>
</service-impl-bean>
<handler>
  <handler-name>XWSSInterceptor</handler-name>
  <handler-class>cc.wss.WSSServiceHandler</handler-class>
</handler>
</port-component>
</webservice-description>
</webservices>

```

3. The handler looks to a file named **security.properties** for information including the keystore to use for signature and encryption, and the XWSS configuration file to load. (This file, and the keystore, and the XWSS configuration file, are copied into the necessary locations by the Ant build process.) Currently it identifies the file **NoSecurity.xml**; change this now:

```

security-config=RequireUsername.xml
keystore=Clinic.jks
...

```

Message Security for the Hospital**LAB 5A**

4. Create a new file **RequireUsername.xml** in the **Hospital** directory, with the contents below. You can use **NoSecurity.xml** as a template.

```
<SecurityConfiguration xmlns="http://java.sun.com/xml/ns/xwss/config" >
  <RequireUsernameToken />
</SecurityConfiguration>
```

5. Build and deploy the service.
6. Try running any of the methods from the Clinic and you'll find the requests are refused:

run confirm

```
javax.xml.rpc.soap.SOAPFaultException: Message security failure
```

7. Test the service from SOAPPad using the prepared message **UsernameToken.txt**. You'll find that you get a response – the first time! A second request will be refused, because the XWSS processor inspects the nonce included in the username token and sees that it's been given once already – this defeats replay attacks.

```
<faultstring>Message security failure</faultstring>
<detail>
  <stackTrace>com.sun.xml.wss.XWSSecurityException:
com.sun.xml.wss.impl.WssSoapFaultException: Invalid/Repeated Nonce
value for Username Token
```

Message Security for the Clinic

LAB 5B

In this lab you will integrate XWSS support into the Clinic client application, and re-establish communications with the Hospital. You'll then be able to tune up the configurations on both sides, until you have all the desired security features in place.

Lab workspace:	Labs/Lab5B
Backup of starter code:	Examples/Healthcare/Step2
Answer folder(s):	Examples/Healthcare/Step3
Files:	Clinic/src/cc/med/Clinic.java Clinic/security.properties Client/Username.xml (to be created) Client/Sign.xml (to be created) Client/Final.xml (to be created) Client/WrongPassword.xml (to be created) Hospital/security.properties Hospital/RequireSignature.xml (to be created) Hospital/Encrypt.xml (to be created) Hospital/Final.xml (to be created)

Instructions:

1. Open **Clinic.java** and find the **initializeService** method. See that so far it just creates a service proxy and uses that to create a stub. You'll add code between these two steps so that a handler chain is configured before the stub is created. (For the next few steps, you can use your code from Lab 3 as a basis.)
2. Create a new **QName** reference **portName**, passing two arguments to the constructor: the namespace URI "http://www.capstonecourseware.com/WS/Hospital" and the port name "Records".
3. Get a reference to the handler registry by calling **service.getHandlerRegistry**.
4. Create a **List** of **HandlerInfo** references, and to it add one **HandlerInfo** that you create with constructor arguments **WSSClientHandler.class**, **null**, and **null**.
5. Call **setHandlerChain** on the handler registry, passing **portName** and your list of **HandlerInfo** references.

Now the service proxy is configured to attach this handler chain to any stub it creates.

Message Security for the Clinic**LAB 5B**

6. **WSSClientHandler** reads **security.properties** too. Edit this to use a new XWSS configuration file:

```
security-config=Username.xml
keystore=Clinic.jks
...
```

7. Create the new configuration as follows – as with all the configuration files in this lab, you'll probably find it simplest to use an existing file (**NoSecurity.xml** in this case) as a template.

```
<SecurityConfiguration xmlns="http://java.sun.com/xml/ns/xwss/config" >
  <UsernameToken name="the_clinic" password="shminic" />
</SecurityConfiguration>
```

8. Build the application and test it out. You should see that you're again able to request information from the Hospital; it authenticates your requests (which, if you inspect them via SOAPSniffer, you'll see are similar to the canned request you used for testing at the end of the previous lab).

```
asant
run confirm
Confirming patient info ... confirmed.
```

9. So, we have handlers in place on both sides of the conversation; from here, it's just a matter of tuning up the configuration files. Try requiring a signature on the request with a configuration **RequireSignature.xml** on the **Hospital** side:

```
<SecurityConfiguration xmlns="http://java.sun.com/xml/ns/xwss/config" >
  <RequireSignature />
</SecurityConfiguration>
```

... and **Sign.xml** on the **Clinic** side:

```
<SecurityConfiguration xmlns="http://java.sun.com/xml/ns/xwss/config" >
  <Sign/>
</SecurityConfiguration>
```

Note that you must rebuild both projects before testing, because the configuration files have to be copied into the EAR file (for the service) and the class path (for the client) by the build process.

Message Security for the Clinic**LAB 5B**

10. On the service side, try encrypting the response message – this should assure patient confidentiality:

```
<SecurityConfiguration xmlns="http://java.sun.com/xml/ns/xwss/config" >
  <RequireSignature />
  <Encrypt>
    <X509Token certificateAlias="Clinic" />
  </Encrypt>
</SecurityConfiguration>
```

(Notice an idiosyncrasy of XWSS here: we could sign without specifying the key, because the JAAS handler would provide it; with encryption we have to provide the key in the config file.)

Rebuild the service and test from the client side. You'll see an encrypted response body, but the client decrypts the information (automatically – no special configuration is needed) and shows it in the console as usual. See **EncryptedResponse.txt**.

11. Not every piece of every response from this service really merits encryption. If you wanted to encrypt only the medical history, you could try a specific encryption target:

```
<Encrypt>
  <X509Token certificateAlias="Clinic" />
  <EncryptionTarget type="xpath" value="//event" contentOnly="false" />
</Encrypt>
```

This too will work – see **EncryptedHistory.txt** – but if you try a request other than **getHistory**, the service will crash. XWSS isn't very graceful about failing to find a configured encryption target, so if the history **<event>** elements aren't there, things get ugly. We'll come back to this issue in a later lab.

12. The answer code in **Step3** uses a **Final.xml** file on each side – the service:

```
<SecurityConfiguration xmlns="http://java.sun.com/xml/ns/xwss/config" >
  <RequireUsernameToken />
  <RequireSignature/>
  <Encrypt>
    <X509Token certificateAlias="Clinic" />
  </Encrypt>
</SecurityConfiguration>
```

... and the client:

```
<SecurityConfiguration xmlns="http://java.sun.com/xml/ns/xwss/config" >
  <UsernameToken name="the_clinic" password="shminic" />
  <Sign/>
</SecurityConfiguration>
```