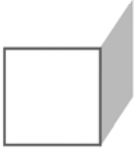
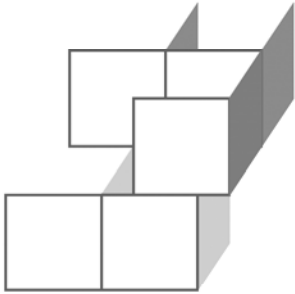
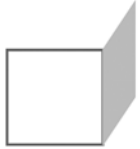
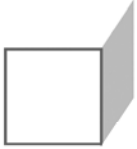




# CHAPTER 6

## WS-SECURITYPOLICY



## OBJECTIVES

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

- Explain the need for published and interoperable policies for web services and clients.
- Explain the need for detailed, interoperable message-security policies.
- Use the WS-Policy and WS-SecurityPolicy languages to express message-security policies for web services, including:
  - Signature
  - Encryption
  - Username/password
  - Timestamps
  - Certificates
- Drive WS-Security implementations from WS-SecurityPolicy documents, using tools such as XWSS and WSIT.

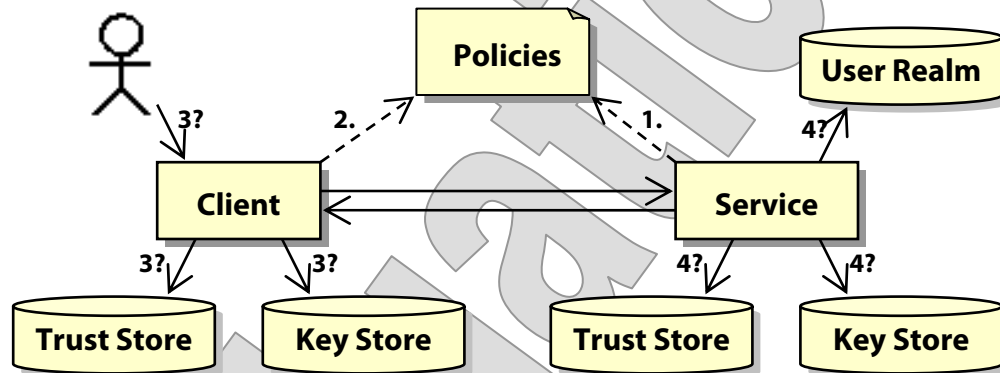
## Two Problems

---

- Two things about our services as we've been building and using them thus far might be frustrating.
- For one thing, while we have solid implementations on both sides, there is no shared, interoperable expression of what is expected in the way of message security.
  - We're accustomed to using **WSDL** to define the service model, and everyone relies on that descriptor to build their code.
  - But, so far, neither party has a similar way to explain its expectations about message security, or to understand what the other party's expectations are so that they can be met.
- For another, we are doing everything at a component level in our implementations.
  - Granted, there are some things about message security that are naturally **fine-grained**, and that therefore require some application code to implement them.
  - But we've been doing a lot of the same stuff from one service to the next, using a generic message-handler class.
  - A lot of message security implementation is **boilerplate**; all that is lacking is a way to configure the right code with fine granularity.
- In this chapter, we'll kill both birds with one stone.
  - We'll look at the interoperability problem first, by way of published security policies.
  - We'll then see that implementation can be driven from these policies, just as basic SOAP-serialization is driven from WSDL.

## Use Case: Sharing Metadata

- So our primary use case for this chapter is fairly general:
  - There are lots of message-security features we might require.
  - The point is that we state these requirements as policies, share them, and then enforce them – all in a consistent manner.



1. The service publishes one or more message-security policies.
  2. When the client is ready to invoke the service (or possibly in advance, such as during a client build) it consults the policies.
  3. It gathers necessary credentials and keys to synthesize a message that carries the required security tokens in the required layout.
  4. The service checks its own policies dynamically as well, and assures that the message satisfies (“complies with”) those policies.
- **With shared policies we get interoperability.**
  - **Generic WSS tools can adopt these policies as their configuration files, as well.**
  - **By the way, the service needn’t be the policy publisher.**
    - As with WSDL, these “contracts” could be authored by the service, the client, or even a third party, such as a standards body.

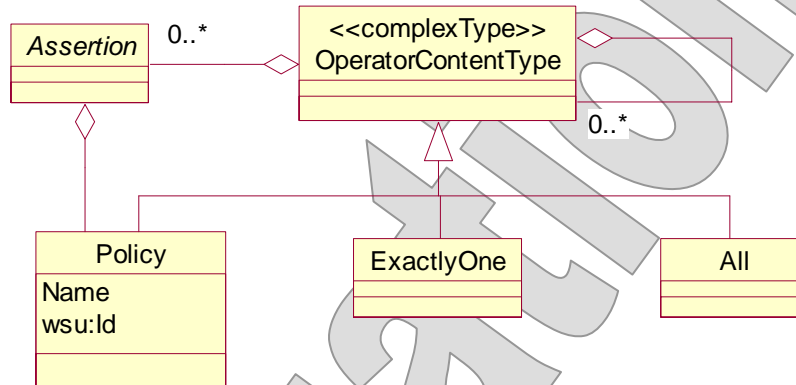
## WS-Policy

---

- OASIS provides the **WS-Policy** specification to define two things:
  - A generic **policy language**
  - A standard for **attaching** policies to their targets
- It defines a standard namespace, usually prefixed **wsp**:  
`http://schemas.xmlsoap.org/ws/2004/09/policy`
- This namespace covers XML vocabulary for both the policy and policy-attachment languages.
- See **c:/Capstone/JavaWSSecurity/Schema/WS-Policy1.2.xsd**.

## Policy Model

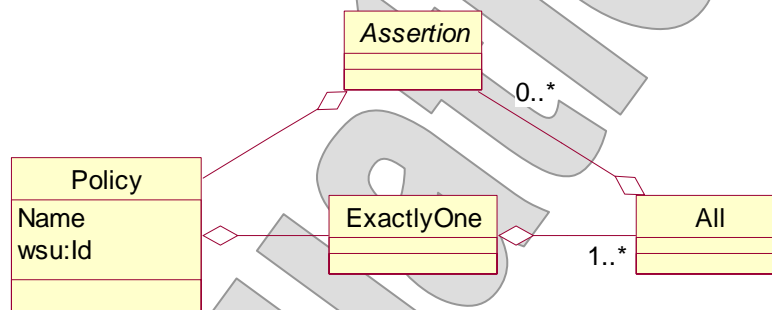
- When we say generic, we mean generic.
  - The policy model is tiny, simple, and very weakly typed:



- You can't really say anything with just WS-Policy.
- Rather, it is a way of collecting and combining **assertions**: all of these, just one of those, either of these and either of those, etc.
- Specific policy languages will extend this one by defining their own types of assertions.
- Plain-English meanings for the element types are:
  - **<ExactlyOne>** means this **OR** that – XML Schema would call this a **disjunction**
  - **<All>** means this **AND** that – that is, a **conjunction**

## Normalized Form

- WS-Policy defines a **normalized form** for policies – this is a subset of the possible expressions allowed under the (intentionally lenient) XML Schema.
  - The formal model, if limited to what's legal in normalized form, would look like this:



- In normalized form, a **policy** is always **exactly one** of some number of sets of **all** of some assertions.
- Any schema-valid policy can be normalized, and there is only one valid normalized form for that policy.

(A OR B) AND C  
 becomes  
 (A AND C) OR (B AND C)

- Thus normalized form is **canonical**, which means two policies that may not have the same XML information set (even when XML-canonicalized) can be considered equivalent if their normalized forms are identical.
- It also means that a policy consumer – or a specification – can set processing rules for certain kinds of policies based on normal form, while readily accepting any schema-valid policy.

## Compact Form

---

- The specification also recognizes a **compact form**, which is actually the one expressed in the original UML model at the beginning of the chapter.
- Under compact form, a few things are legal that are not legal in normalized form:
  - **Optional assertions**, using the **wsp:Optional** attribute.
  - **Nested policies**, whereby an assertion holds a policy of its own as a way of managing a complex set of options; normalized policies are entirely flattened, which can make them just about completely unreadable! but gets us to canonicalization.
- **Compact form is generally preferred where policies are being hand-authored.**

## Simple Constructs

**EXAMPLE**

- In **Examples/Policy/WS-Policy** we have a few simple examples of WS-Policy constructions, each available in compact and normalized forms.
- All the assertions are hypothetical – which will be pretty obvious from the element names ...
- In **All.xml** is a policy that combines three assertions – this means that to adhere to this policy, one must comply with all three assertions:

```
<wsp:Policy ... Name="All" >
  <AssertionA/>
  <AssertionB/>
  <AssertionC/>
</wsp:Policy>
```

- The normalized form for this policy is shown in **All\_Normalized.xml**.

```
<wsp:Policy ... Name="All" >
  <wsp:ExactlyOne>
    <wsp:All>
      <AssertionA/>
      <AssertionB/>
      <AssertionC/>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

## Simple Constructs

**EXAMPLE**

- **Choice.xml** shows a policy that allows for any of three options:

```
<wsp:Policy ... Name="Choice" >
  <wsp:ExactlyOne>
    <AssertionA/>
    <AssertionB/>
    <AssertionC/>
  </wsp:ExactlyOne>
</wsp:Policy>
```

- To comply with this policy, one must comply with one of the assertions – but not with more than one, as that would be just as big a compliance failure as missing one from the **All.xml** policy.

- With this one, the normalized form starts to look a little odd – see **Choice\_Normalized.xml**:

```
<wsp:Policy ... Name="Choice" >
  <wsp:ExactlyOne>
    <wsp:All>
      <AssertionA/>
    </wsp:All>
    <wsp:All>
      <AssertionB/>
    </wsp:All>
    <wsp:All>
      <AssertionC/>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

## Simple Constructs

**EXAMPLE**

- **Optional.xml** makes two assertions mandatory and allows one as an option:

```
<wsp:Policy ... Name="Optional" >
  <AssertionA />
  <AssertionB />
  <AssertionC wsp:Optional="true" />
</wsp:Policy>
```

- To normalize a policy with optional assertions is to duplicate the surrounding conjunction, wrap both in a disjunction, and then make the optional assertion mandatory in one case and erase it from the other.
  - That is, “A and B and maybe C” becomes “A and B and C, or just A and B.”
  - The normal form is shown in **Choice\_Normalized.xml**:

```
<wsp:Policy ... Name="Optional" >
  <wsp:ExactlyOne>
    <wsp:All>
      <AssertionA />
      <AssertionB />
      <AssertionC />
    </wsp:All>
    <wsp:All>
      <AssertionA />
      <AssertionB />
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

## Simple Constructs

**EXAMPLE**

- **Nested.xml** shows a nested assertion: A includes a choice of options A1 and A2:

```
<wsp:Policy ... Name="Nested" >
  <AssertionA>
    <wsp:Policy>
      <wsp:ExactlyOne>
        <AssertionA1/>
        <AssertionA2/>
      </wsp:ExactlyOne>
    </wsp:Policy>
  </AssertionA>
  <AssertionB/>
  <AssertionC/>
</wsp:Policy>
```

- This means “A (with either A1 or A2 but not both) and B and C.”

## Simple Constructs

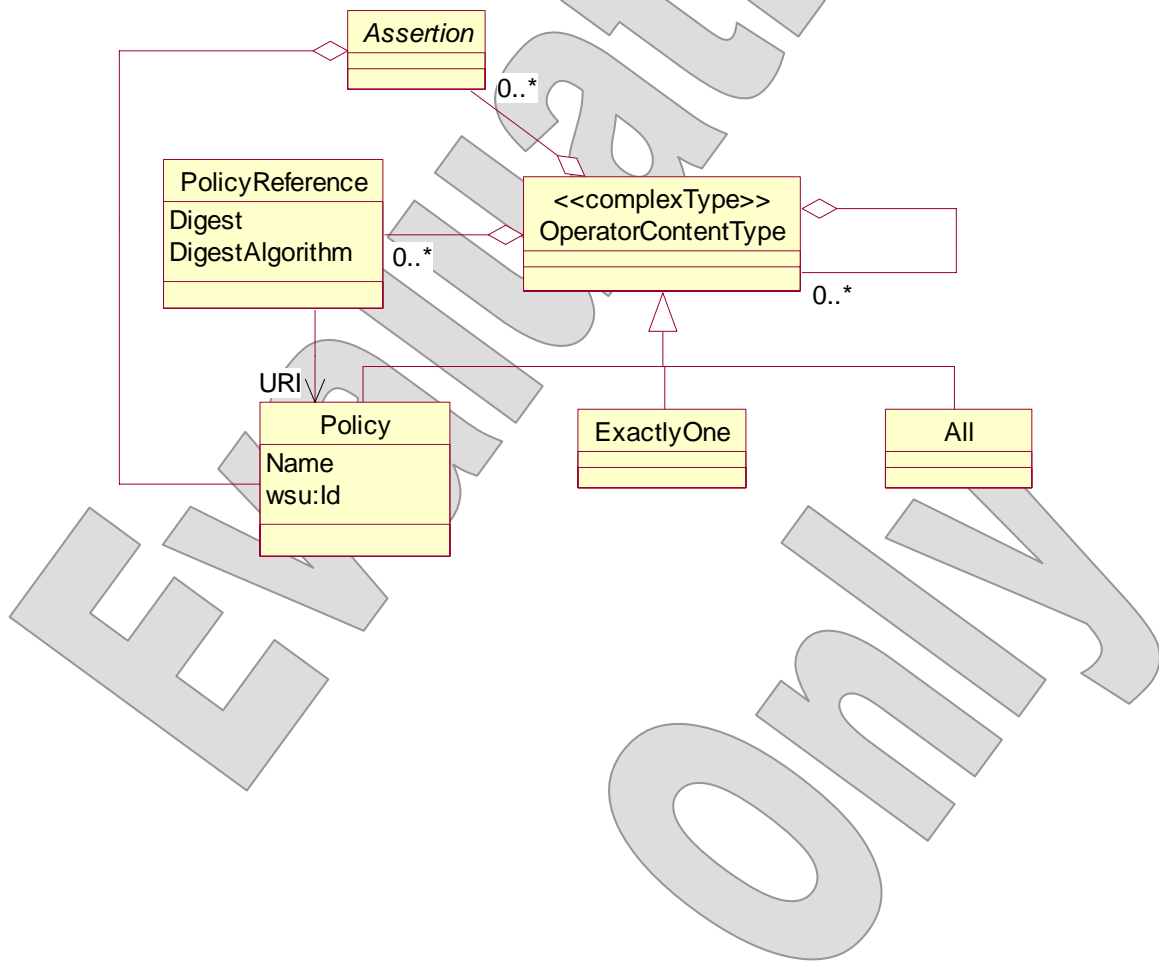
**EXAMPLE**

- Normalizing a nested policy is straightforward unless it has multiple choices within it, as this one does.
- Then, we have to do the same trick we do with optional assertions; the results are not too pretty:

```
<wsp:Policy ... Name="Nested" >
  <wsp:ExactlyOne>
    <wsp:All>
      <AssertionA>
        <wsp:Policy>
          <wsp:ExactlyOne>
            <wsp:All>
              <AssertionA1/>
            </wsp:All>
          </wsp:ExactlyOne>
        </wsp:Policy>
      </AssertionA>
      <AssertionB/>
      <AssertionC/>
    </wsp:All>
    <wsp:All>
      <AssertionA>
        <wsp:Policy>
          <wsp:ExactlyOne>
            <wsp:All>
              <AssertionA2/>
            </wsp:All>
          </wsp:ExactlyOne>
        </wsp:Policy>
      </AssertionA>
      <AssertionB/>
      <AssertionC/>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

## Policy Attachment

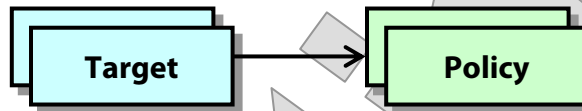
- WS-Policy also defines means by which a policy can be formally associated with a target – by which we mean the thing to which the policy should apply.
- One more puzzle piece is added to the overall model to support references to policies (by assertions, other policies, or by elements outside the WS-Policy namespace):



## Target Points to Policy

---

- In fact it defines two approaches.
- One is called **XML element attachment**: some XML element that represents the policy target identifies the policy.



- It can use the **wsp:PolicyURIs** attribute to do this:

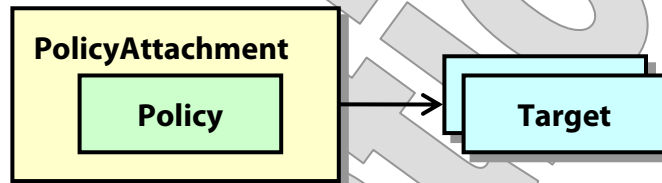
```
<SomeTarget wsp:PolicyURIs=  
  "http://my.com/SorryButThat'sOurPolicy"  
>/>
```

- Or, it can hold a child element **<wsp:PolicyReference>**:

```
<SomeTarget>  
  <wsp:PolicyReference URI="http://my.com/XYZ" />  
</SomeTarget>
```

## Policy/Attachment Points to Target

- In **external policy attachment**, an attachment expression announces that a policy “applies to” some target.
- The attachment element can include the policy, so that really it’s the policy identifying its target(s):

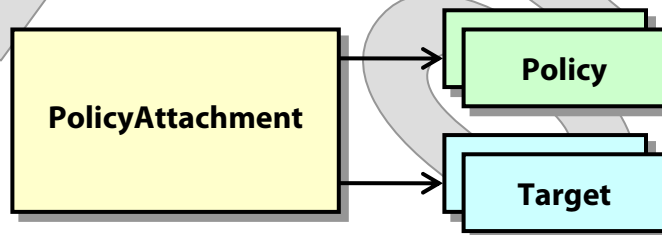


```

<wsp:PolicyAttachment>
  <wsp:AppliesTo>
    <wsa:EndpointReference xmlns:my="..." >
      <wsa:Address>http://my.com/ABC</wsa:Address>
      <wsa:PortType>my:port</wsa:PortType>
      <wsa:ServiceName>my:service</wsa:ServiceName>
    </wsa:EndpointReference>
  </wsp:AppliesTo>
  <wsp:Policy>...</wsp:Policy>
</wsp:PolicyAttachment>

```

- Or, it can refer to an external policy, meaning that it is connecting two external things to each other – neither of which internally identifies (or depends on) the other:



- Instead of a `<Policy>`, then, the `<PolicyAttachment>` would include a `<PolicyReference>`.

## Policy Scopes

---

- For web services, the usual policy targets are the messages traded by those services and their clients – that’s simple enough.
- But there are a few **scopes** at which it’s useful to state policy:
  - **Port** – this is the widest applicable policy scope, and many assertions will be placed here as they’re equally applicable to all messages in all directions
  - **Operation** – we might want to say some things specific to an operation, such as in our earlier example where we wanted encryption only for certain pieces of the **getHistory** operation
  - **Message** – we might distinguish between inbound and outbound policies for a given operation, as in we require signature from the client but not from the service, encrypted responses not requests, etc.
- Many assertion types might reasonably be applied at any scope.
- Some are natural to larger scopes.
  - One clear example of this is the assertion that a service wishes to have a **secure conversation**.
  - This not only affects all operations and messages, but also defines a process that will govern a given client’s interaction with the service as a whole, and encompasses a series of messages in both directions.
  - We’ll consider secure conversations in the following chapter.

## WS-SecurityPolicy

---

- WS-SecurityPolicy extends the generic policy language by defining
  - Specific **assertion types**
  - **Bindings**, which are themselves policy assertions but include other assertions in a nested model and set comprehensive rules for content and processing

- The standard namespace for this specification is generally prefixed **sp**:

`http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702`

- See the governing schema document in our top-level **Schema** directory, as **WS-SecurityPolicy1.2.xsd**.

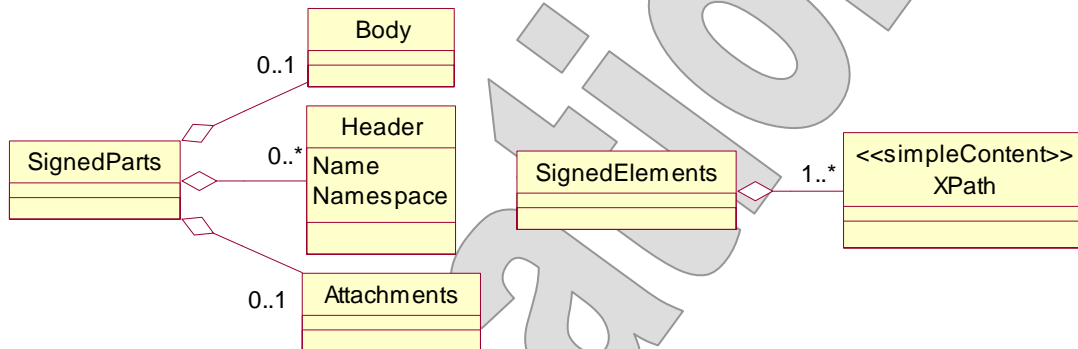
## Assertion Types

---

- The standard defines a large vocabulary of possible message-security assertions, which it breaks into categories like this:
  - **Protection assertions** describe the necessary signature and encryption on a message, along with statements about what sorts of elements are required.
  - **Token assertions** set requirements, not as to specific tokens (as in, you must present the certificate for the Handyman DIY Corporation), but as to token types (as in, you must present some X.509 certificate, which we'll validate by other means).
  - **Supporting and endorsing tokens** are those which are not integral to the protection mechanism but may nevertheless be required as a matter of policy: for example, a username token.
- Other assertion types are either **bindings**, or what the specification calls **binding properties**.
- In fact it is these bindings which are meant to be the **top-level policies at the port scope**.
  - WS-Policy would allow any of the above to be stated as a policy, and be found structurally valid.
  - But WS-SecurityPolicy narrows the options down to these common practices known as bindings, and all the other assertion types either serve to explain the binding details, or support or endorse the resulting messages.
  - Non-binding assertions may form the basis of policies applied at operation and message scopes.

## Protection Assertions

- There are three major types of protection assertion.
- `<SignedParts>` and `<SignedElements>` require signature on specific parts of the message:



- A **part** is identified as the message body, a header element of a certain qualified XML name, or all of the message attachments.
- An **element**, by contrast, can be anything.
- There is a nearly identical model for `<EncryptedParts>`, `<EncryptedElements>`, and the additional concept of `<ContentEncryptedElements>`.
- At this level there's no statement of sign-before-encrypt, or vice-versa, but that does come into play at a higher level, in bindings.
- Finally there is a way to say that there are `<RequiredParts>` and `<RequiredElements>` in the message.
  - XML Schema is usually enough for the service model, but WS-SecurityPolicy is stepping in here to specify header elements.

## Sign and Encrypt

**EXAMPLE**

- In **Examples/WS-SecurityPolicy**, see **SignAndEncrypt.xml** for a set of message-protection assertions:

```
<wsp:Policy ... wsu:Id="MessagePolicy" >
  <sp:EncryptedParts>
    <sp:Body/>
  </sp:EncryptedParts>
  <sp:SignedParts>
    <sp:Body/>
    <sp:Header Name="Action" Namespace="..."/>
    <sp:Header Name="FaultTo" Namespace="..."/>
    <sp:Header Name="From" Namespace="..."/>
    <sp:Header Name="MessageID" Namespace="..."/>
    <sp:Header Name="RelatesTo" Namespace="..."/>
    <sp:Header Name="ReplyTo" Namespace="..."/>
    <sp:Header Name="To" Namespace="..."/>
  </sp:SignedParts>
</wsp:Policy>
```

- The elided **Namespace** value for all the **<sp:Header>** elements is **<http://www.w3.org/2005/08/addressing>**

- This is the namespace for **WS-Addressing**, which is a bit beyond our scope but will often factor into security policies.

- These two specifications intersect as they do because, by stating in XML certain things that might be implicit in the carrying protocol (HTTP), we make it possible to sign them.

- **Note that a signed part is not necessarily a required part.**

- That is, these header entries will be signed if they are found on the message – but it's not necessarily a problem if they are missing.

## Token Assertions

---

- About a dozen assertion types identify token types to be used in certain ways in the messaging process.
- For example, we can say that a message-security binding will rely on an X.509 certificate to spin up a secure exchange.
- Each token type has its own policy model, but some common elements are:
  - We can state a requirement to identify the token **issuer**
  - We can call out the **WS-Security version** used to express the token's information
  - We can set policy as to **how often** the certificate should be sent, with values such as “Once”, “AlwaysToRecipient”, and “Never”.
- We won't delve into every token type, but other common types are shown here, and we'll see examples of a few of these later in the chapter:

```
<UsernameToken>  
<X509Token>  
<KerberosToken>  
<SecurityContextToken>  
<SecureConversationToken>  
<SamlToken>  
<HttpsToken>
```

## Certificate Token

**EXAMPLE**

- In **Examples/WS-SecurityPolicy**, the file **Certificate.xml** expresses the policy that
  - There must be a **certificate**, presented as a WS-Security 1.0 binary security token
  - It must be **sent every time** if there are multiple messages

```
<wsp:Policy wsu:Id="UseACertificateForSomething" >
  <sp:X509Token
    sp:IncludeToken=".../IncludeToken/Always"
  >
    <wsp:Policy>
      <sp:WssX509V3Token10/>
    </wsp:Policy>
  </sp:X509Token>
</wsp:Policy>
```

Evaluation Only

## Supporting and Endorsing Tokens

---

- Tokens that aren't directly involved in the security mechanisms at the message level are called **supporting or endorsing tokens**.
  - **Supporting** tokens are simply additional information.
  - **Endorsing** tokens sign the message signature, adding layers of authenticity assurance.
- **Supporting and endorsing tokens are often signed, or encrypted, or both.**
- **There are separate assertion elements to capture these combinations; each of these holds a <wsp:Policy>, which holds token assertions**

`<SupportingTokens>`

`<SignedSupportingTokens>`

`<EncryptedSupportingTokens>`

`<SignedEncryptedSupportingTokens>`

etc.

## Signed Username Token

**EXAMPLE**

- In **Examples/WS-SecurityPolicy**, the file **Username.xml** expresses the policy that
  - There must be a **username** token, according to WS-Security 1.0
  - It must be **sent once**, in the first message between two parties
  - It must be signed along with the rest of the signed message parts

```
<wsp:Policy ... wsu:Id="SignTheUsername" >
  <sp:SignedSupportingTokens>
    <wsp:Policy>
      <sp:UsernameToken
        sp:IncludeToken=".../IncludeToken/Once"
      >
        <wsp:Policy>
          <sp:WssUsernameToken10/>
        </wsp:Policy>
      </sp:UsernameToken>
    </wsp:Policy>
  </sp:SignedSupportingTokens>
</wsp:Policy>
```

## <AlgorithmSuite>

---

- We're on our way to the larger-scale assertions known as bindings now, and so-called **properties** are a part of that model.
- One property that we'll see in many contexts is the <AlgorithmSuite>.
- This identifies a coherent set of algorithms, as a child assertion:

```
<sp:AlgorithmSuite>  
  <wsp:Policy>  
    <sp:Basic128/>  
  </wsp:Policy>  
</sp:AlgorithmSuite>
```

- WS-SecurityPolicy defines an assertion type for each of these suites, and allows extension to other suites as well.
- The example above, <Basic128>, says
  - We'll hash with **SHA-1**
  - We'll do symmetric encryption with **AES** and 128-bit keys
  - We'll wrap keys using **RSA** in the OAEP mode
  - We'll sign with **RSA/SHA-1**
  - Several other, more arcane statements
- There are too many to list here, but predefined suites exist to support combinations of
  - **128-bit**, **192-bit**, and **256-bit** strength levels
  - **SHA-1** and **SHA-256**
  - **AES** and **3DES**

## <Layout>

---

- Another common property of security-policy bindings is the message layout, using the <Layout> assertion.

```
<sp:Layout>  
  <wsp:Policy>  
    <sp:Strict/>  
  </wsp:Policy>  
</sp:Layout>
```

- There are four possible statements we can make in this way:
  - **Strict** enforces as hard rules the recommendations of WS-Security regarding avoidance of forward-references in the order of security tokens in the SOAP header: keys declared before used, explicit declaration of “reference lists” of encrypted elements found later in the message, and so forth
  - **Lax** allows anything that is legal under WS-Security – that is, a “may” or a “should” from that specification is not a requirement
  - **LaxTimestampFirst** and **LaxTimestampLast** are like **Lax** but with the single requirement that the timestamp element be either first or last in the <wsse:Security> block.

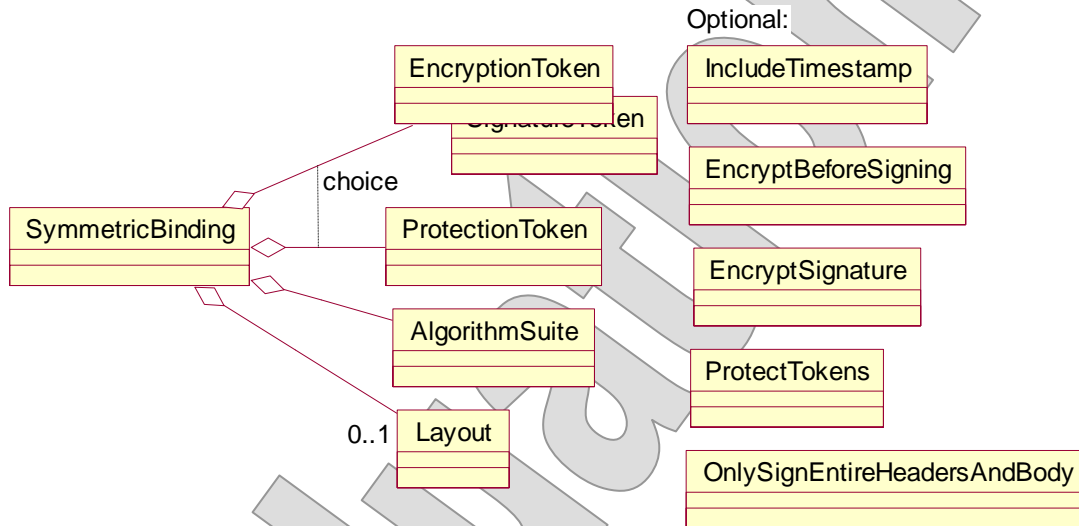
## Bindings

---

- **WS-Policy is a wide-open model that supports arbitrary folding and nesting, optional assertions, etc.**
- **Message security at some point demands a firmer hand.**
- **WS-SecurityPolicy could just lay out a lot of assertion types and leave it to the policy author to figure out sensible combinations.**
  - This would leave most of us scrambling for answers.
  - And, plenty of “legal” policies would nevertheless be bad ideas: requirements to sign with no corresponding token requirements, and so forth.
- **In fact, WS-SecurityPolicy lays out a handful of much more strict models, each of which represents a common current practice:**
  - **<TransportBinding>** sets requirements for use of HTTPS as the security mechanism for a service. (This is a perfectly useful policy technique, but not interesting in the scope of this course as we’re working with message-level security.)
  - **<SymmetricBinding>** sets up message-level security in which the tokens used to encrypt and to sign are the same in both directions.
  - **<AsymmetricBinding>** sets up message-level security in which different token sets may be used in different directions: for example a key pair used by the client to sign requests and a different key pair used by the service to sign responses.

## <SymmetricBinding>

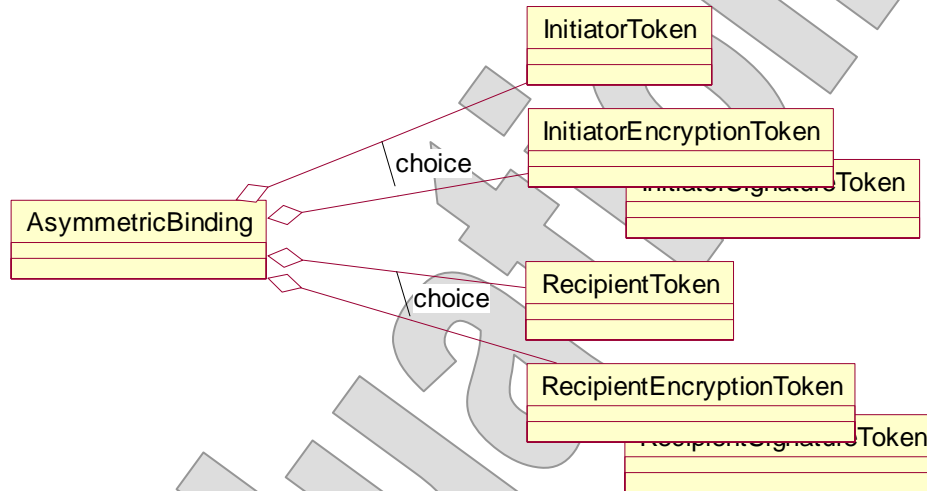
- To assert a symmetric message-security policy, we have to define a few things:



- The **token(s)** that will be used to protect the messages
- An **algorithm suite**
- Then we can also call out layout policy, timestamps, sign/encrypt ordering, and so on.
- Note that symmetric binding does not say that we'll use symmetric encryption!
  - Often we do, and that's how <ProtectionToken> is used: it will identify a certificate which is used to wrap a generated, symmetric key – and there's the SSL model once again.
  - But we can also identify the static <EncryptionToken> and <SignatureToken> that will be used: this indicates asymmetric encryption and signature

## <AsymmetricBinding>

- With <AsymmetricBinding>, we state as many as four distinct tokens:



- The **initiator** might have one key pair for encryption and one for signature, or use the same key for both.
- The **recipient** has the same options.
- The rest of the binding model is the same as it is for <SymmetricBinding>, including algorithm suite, layout, timestamp, etc.

## Symmetric Binding with RSA and AES

**EXAMPLE**

- In `Examples/Policy/WS-SecurityPolicy, Symmetric.xml` expresses a coherent policy calling for symmetric protections over a SOAP conversation.
- Let's look at this piece-by-piece:

```
<wsp:Policy ... wsu:Id="Symmetric" >
  <sp:SymmetricBinding>
    <wsp:Policy>
```

- We say that we will use an X.509 certificate to present the public key that is used to wrap a generated secret key – and that we'll not transmit that key with the messages, but refer to it externally:

```
  <sp:ProtectionToken>
    <wsp:Policy>
      <sp:X509Token sp:IncludeToken=
        ".../IncludeToken/Never"
      >
        <wsp:Policy>
          <sp:WssX509V3Token10/>
        </wsp:Policy>
      </sp:X509Token>
    </wsp:Policy>
  </sp:ProtectionToken>
```

- The message layout policy is strict:

```
  <sp:Layout>
    <wsp:Policy>
      <sp:Strict/>
    </wsp:Policy>
  </sp:Layout>
```

## Symmetric Binding with RSA and AES

**EXAMPLE**

- Messages will be time-stamped:

```
<sp:IncludeTimestamp/>
```

- We will allow signature encryption only over entire units: the message body or individual security tokens. (This may seem obvious but it would otherwise be possible for a policy at a narrower scope to call for signature of just some small piece; with this assertion that is forbidden.)

```
<sp:OnlySignEntireHeadersAndBody/>
```

- We'll use the "basic 128" suite, meaning RSA-OAEP to wrap an AES encryption key:

```
<sp:AlgorithmSuite>  
  <wsp:Policy>  
    <sp:Basic128/>  
  </wsp:Policy>  
</sp:AlgorithmSuite>  
</wsp:Policy>  
</sp:SymmetricBinding>  
</wsp:Policy>
```

## Attaching a Policy in WSDL

**EXAMPLE**

- In **Examples/Pizza/WS-SecurityPolicy**, there are several examples of security policies for the Pizza service.
- We'll explore these individually in a moment, but for now let's look at how these policies are incorporated into the service WSDL: consider **Delivery/WSDL/Symmetric.wsdl**:
  - WSDL allows broad extensibility under its content model, and so we can include policies as top-level elements in the descriptor:

```
<definitions ... >
  <wsp:Policy wsu:Id="PizzaPolicy" >
    <sp:SymmetricBinding>
      ...
```

- Notice that we have a general policy and then policies for inbound and outbound messages:

```
<wsp:Policy wsu:Id="InboundPolicy">
  <sp:SignedParts>
    <sp:Body/>
  </sp:SignedParts>
</wsp:Policy>

<wsp:Policy wsu:Id="OutboundPolicy">
  <sp:SignedParts>
    <sp:Body/>
  </sp:SignedParts>
</wsp:Policy>
```

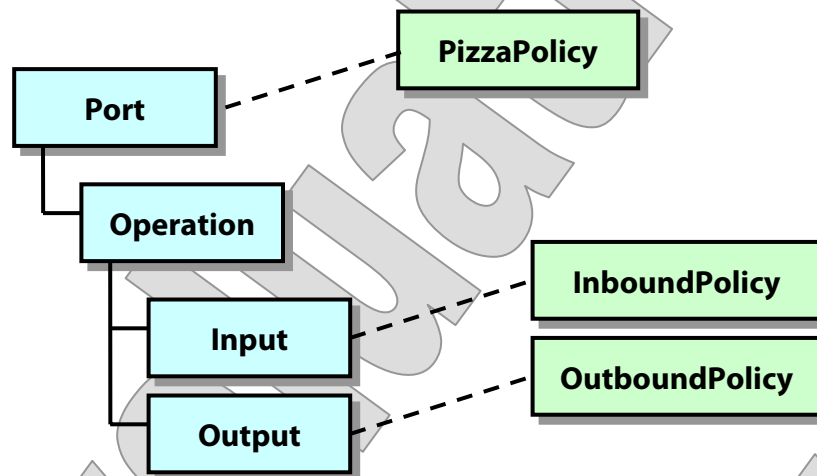
## Attaching a Policy in WSDL

**EXAMPLE**

- Roll down to the `<binding>` element, and see that it is the attachment point for the primary policy:

```
<binding name="DeliveryBinding" type="p:Delivery">
  <wsp:PolicyReference URI="#PizzaPolicy" />
  <wsp:PolicyReference URI="#ServiceCredentials" />
</binding>
```

- We'll come back to that second policy a bit later.



- Then, each of the two message types for our only operation has a policy attached to it as well:

```
...
<operation name="orderPizza">
  <input>
    <wsp:PolicyReference URI="#InboundPolicy"/>
    ...
  </input>
  <output>
    <wsp:PolicyReference URI="#OutboundPolicy"/>
    ...
  </output>
</operation>
</binding>
```

## WSIT and Project Metro

---

- For Java applications, the **Web Services Interoperability Technology** standard, or **WSIT**, provides for container-based message security.
  - WSIT is primarily geared toward Java-.NET interoperability.
  - Part of that is interoperable message security, and for this it processes WS-SecurityPolicy policies as configuration.
- A broader open-source project called **Metro** implements WSIT (using XWSS, in fact), and is in turn embedded in GlassFish.
- The upshot of all this is that we can build web services and clients from WSDL descriptors that embed WS-SecurityPolicy information, and
  - The server will implement those policies!
  - The client-side stub will implement them, too! Dynamically!
- This is pretty good stuff, as you'll see that we don't need nearly as much application code to get the same security features.
- And Metro supports more features than we could ever have implemented piecemeal if we'd continued to pursue the approach of the previous chapter.
  - So, hopefully that was illustrative, but we're about to get a much bigger bang for our coding buck.
  - We'll start to look at our SOAP results at a higher level as well.

## Private “Policies”

---

- Remember that WS-SecurityPolicy provides a way to call for certain token types – usernames, certificates, etc. – but does not specify what they will be.
  - The policy is a **contract**.
  - We need some other way to meet our **responsibilities** under that contract: i.e. what usernames will be provided in a request, what trust stores will be used to validate certificates on message receipt?
- Metro takes an interesting approach to this, which echoes XWSS’ blend of declarative and programmatic solutions.
- WSIT defines WS-Policy assertion types for keystore, trust store, user names, and so on.
  - These assertions can then be set into the application on the service side or the client side.
  - They can also be included in the WSDL descriptor, for convenience, by way of a **WS-Policy**.
  - In this case they are marked with an attribute **wspp:visibility**, set to “private” – which tells Metro not to propagate them to the published WSDL, but rather to consume them.
- Metro still uses JAAS callbacks for certain information, and offers similar options to use either configuration or Java programming for things like usernames and passwords.

## Security Policies in Play

EXAMPLE

- So, the WSDL we just studied will be implemented by Metro, you say? On both the service and the client sides? Let's try it!
- There are various WSDL options in this variant of the Pizza application, and so we build a little differently on the server side.
- Build the simple symmetric case now, using the **build** script from the **Delivery** directory:

```
build Symmetric
```

- This just copies **Symmetric.wsdl** to **PizzaDelivery.wsdl** (overwriting if necessary), and then runs **ant** normally.
- Now run **ant** normally from the **Caller** directory.
  - One interesting note on this: see that the client application doesn't need a corresponding WSDL for each service variant.
  - Since policies are published, we can refer to them – see **Caller/WSDL/PizzaDelivery.wsdl**:

```
<definitions>
  <wsp:Policy wsu:Id="ClientCredentials" >
    ...
  </wsp:Policy>
  ...
  <binding name="DeliveryBinding" type="p:Delivery">
    <wsp:PolicyReference
      URI="http://localhost:8080/Pizza
          /Delivery?WSDL#PizzaPolicy" />
    <wsp:PolicyReference URI="#ClientCredentials"/>
  </binding>
</definitions>
```

- So the client will adapt to whatever policy the server has lately published, each time it runs – no need even to rebuild.

## Security Policies in Play

**EXAMPLE**

- Test, and run the traffic through the SOAPSniffer.
  - Notice that the client application has been enhanced a bit: it now makes two calls for a total of three pizzas:

```
run http://localhost:8079/Pizza/Delivery
Ordering pizza ...
  2 pizzas ordered.
  1 more pizza ordered.
```

- The resulting SOAP traffic is downright voluminous; you can see a full capture of the conversation in **SOAP/Symmetric.txt**.
- We won't go through each token and relationship in the message, as we've done a good bit of that already in the course.
- But, in **SOAPSummary/Symmetric.txt** is a much-abbreviated version of the same traffic, leaving just minimal representations of the primary building blocks of the conversation.
  - Most of the innards of signatures, tokens, timestamps, etc., have been removed completely.
  - IDs and references have been retained, and their not-so-readable values have been replaced with meaningful names.
- The first request and response from our test a moment ago are represented in this fashion on the following pages.

## Security Policies in Play

**EXAMPLE**

POST /Pizza/Delivery HTTP/1.1

```

<S:Envelope>
  <S:Header>
    <wsse:Security>
      <wsu:Timestamp wsu:Id="Timestamp" />
      <wsse:BinarySecurityToken
        wsu:Id="PublicKey"
      />
      <xenc:EncryptedKey Id="SecretKey" >
        <ds:KeyInfo>
          <wsse:SecurityTokenReference>
            <wsse:Reference URI="#PublicKey" />
          </wsse:SecurityTokenReference>
        </ds:KeyInfo>
      </xenc:EncryptedKey>
      <ds:Signature Id="Signature" >
        <ds:SignedInfo>
          <ds:Reference URI="#Body" />
          <ds:Reference URI="#Timestamp" />
        </ds:SignedInfo>
        <ds:KeyInfo>
          <wsse:SecurityTokenReference>
            <wsse:Reference URI="#SecretKey" />
          </wsse:SecurityTokenReference>
        </ds:KeyInfo>
      </ds:Signature>
    </wsse:Security>
  </S:Header>
  <S:Body wsu:Id="Body" />
</S:Envelope>

```

## Security Policies in Play

**EXAMPLE**

HTTP/1.1 200 OK

```
<S:Envelope>
  <S:Header>
    <wsse:Security>
      <wsu:Timestamp wsu:Id="Timestamp" />
      <ds:Signature Id="Signature" >
        <ds:SignedInfo>
          <ds:Reference URI="#Body" >
            <ds:Reference URI="#Timestamp" />
          </ds:SignedInfo>
          <ds:KeyInfo/> <!-- embeds wrapped key -->
        </ds:Signature>
      </wsse:Security>
    </S:Header>
    <S:Body wsu:Id="Body" />
  </S:Envelope>
```

## Security Policies in Play

**EXAMPLE**

- Let's take a moment to understand how each side of the conversation is managing this trick.
  - Each entity is performing some digital cryptography.
  - Where are they getting their key and certificate information?
- The service automatically uses the certificate of the application server, and relies on a pre-configured trust store as necessary.
- Both files are found in **%JAVA\_EE\_HOME%/domains/domain1/config:**
  - The keystore is **keystore.jks**, and the key alias is **s1as** (old term: “Sun One Application Server”).
  - The trust store is **cacerts.jks**.
- The client configures its keystore and trust store choices explicitly – see **Caller/WSDL/PizzaDelivery.wsdl** again:

```
<wsp:Policy wsu:Id="ClientCredentials">
  ...
  <cc:KeyStore location=
    "C:\Sun\SDK\domains\domain1\config\keystore.jks"
  ... />
  <cc:TrustStore location=
    "C:\Sun\SDK\domains\domain1\config\cacerts.jks"
  ... />
</wsp:Policy>
```

- Yes, we're using the server's stores directly – not practical at all.
- For purposes of studying WS-SecurityPolicy, it's good enough.
- But the Healthcare example will take a more robust approach.

## Security Policies in Play

**EXAMPLE**

- Now try building and testing variants of this policy: for each, rebuild the server with the specified WSDL and test with the **run** script from the client side.
- **Delivery/WSDL/Symmetric\_Username.xml** adds a signed username token:

```
<sp:SignedSupportingTokens>
  <wsp:Policy>
    <sp:UsernameToken sp:IncludeToken=".../Once" >
      <wsp:Policy>
        <sp:WssUsernameToken10/>
      </wsp:Policy>
    </sp:UsernameToken>
  </wsp:Policy>
</sp:SignedSupportingTokens>
```

- Build this variant:

```
build Symmetric_Username
```

- Then build from the client side and test it out. You'll see a failure:

```
ant
run
javax.xml.ws.soap.SOAPFaultException:
  Authentication of Username Password Token Failed
```

## Security Policies in Play

**EXAMPLE**

- Thinking about it a bit, this isn't actually much of a surprise.
  - Yes, we're supplying a username and password – the **ClientCredentials** “policy” sets the login **footballfan/hungry**.

```
<wsp:Policy wsu:Id="ClientCredentials">
  <cc:CallbackHandlerConfiguration ... >
    <cc:CallbackHandler name="usernameHandler"
      default="footballfan" />
    <cc:CallbackHandler name="passwordHandler"
      default="hungry" />
    ...
  </cc:CallbackHandlerConfiguration >
</wsp:Policy>
```

- But how are we expecting this user to be authenticated?
- In the Healthcare example, we have **cc.wss.LoginHandler** making authentication decisions for the Hospital.
- Who's making them for the Pizza Delivery service?
- This is actually a significant benefit of container-based message security: the container is also performing authentication for us.
- Or, it's trying to – but we need to give it some clues about valid usernames!
- Create the necessary user account in GlassFish using a prepared Ant target:

```
ant -Duser=footballfan -Dpassword=hungry
  -Dgroups=caller create-user
Command create-file-user executed successfully.
```

## Security Policies in Play

**EXAMPLE**

- Try the client again, and get a clean round-trip.
- Oddly, the difference in the resulting SOAP is mostly in the presence of an `<xenc:ReferenceList>` element and some `<xenc:EncryptedData>`.
  - See `SOAPSSummary/Symmetric_Username.txt`:

```

</xenc:EncryptedKey>
<xenc:ReferenceList>
  <xenc:DataReference URI="#UsernameToken" />
</xenc:ReferenceList>
<xenc:EncryptedData Id="UsernameToken_Encrypted" >
  <ds:KeyInfo>
    <wsse:SecurityTokenReference>
      <wsse:Reference URI="#SecretKey" />
    </wsse:SecurityTokenReference>
  </ds:KeyInfo>
</xenc:EncryptedData>

```

- WSIT has taken “signed” to mean “signed and encrypted.”
- This is arguably a WS-SecurityPolicy compliance failure, but the reasoning is that we don’t want passwords traveling in the clear.
- Note that the policy doesn’t say to hash the password – and WSIT doesn’t yet support authenticating hashed username tokens.
- The token is indeed signed, but it’s also encrypted, so that the ID reference can’t easily be resolved by eye.

```

<ds:SignedInfo>
  <ds:Reference URI="#Body" />
  <ds:Reference URI="#Timestamp" />
  <ds:Reference URI="#UsernameToken_Plain" />
</ds:SignedInfo>

```

## Security Policies in Play

**EXAMPLE**

- In **Delivery/WSDL/Symmetric\_Encrypted.xml**, the username token is gone, and we've added body encryption to each of the inbound and outbound message policies:

```
<wsp:Policy wsu:Id="InboundPolicy">
  <sp:EncryptedParts>
    <sp:Body/>
  </sp:EncryptedParts>
  <sp:SignedParts>
    <sp:Body/>
  </sp:SignedParts>
</wsp:Policy>
```

- **Build and test this variant ...**

```
build Symmetric_Encrypted
```

- ... and see the results – nothing we couldn't predict and haven't seen with XWSS:

```
<xenc:ReferenceList>
  <xenc:DataReference URI="#Body_Encrypted" />
</xenc:ReferenceList>
...
<S:Body wsu:Id="Body" >
  <xenc:EncryptedData Id="Body_Encrypted" >
    <ds:KeyInfo>
      <wsse:SecurityTokenReference>
        <wsse:Reference URI="#SecretKey" />
      </wsse:SecurityTokenReference>
    </ds:KeyInfo>
  </xenc:EncryptedData>
</S:Body>
```

## Security Policies in Play

**EXAMPLE**

- **Delivery/WSDL/Symmetric\_Addressing.wsdl** brings in the use of WS-Addressing, with this single assertion, right at the bottom of the primary policy:

```
<wsaws:UsingAddressing/>
```

- The message policies have been enhanced to assure that all the WS-Addressing headers are signed:

```
<wsp:Policy ... wsu:Id="InboundPolicy" >
  <sp:SignedParts>
    <sp:Body/>
    <sp:Header Name="Action" Namespace="..." />
    <sp:Header Name="FaultTo" Namespace="..." />
    <sp:Header Name="From" Namespace="..." />
    <sp:Header Name="MessageID" Namespace="..." />
    <sp:Header Name="RelatesTo" Namespace="..." />
    <sp:Header Name="ReplyTo" Namespace="..." />
    <sp:Header Name="To" Namespace="..." />
  </sp:SignedParts>
</wsp:Policy>
```

- When you test this through the SOAPSniffer, you see the additional header entries called for by of WS-Addressing:

```
<S:Envelope>
  <S:Header>
    <wsa:To wsu:Id="To" />
    <wsa:Action wsu:Id="Action" />
    <wsa:ReplyTo wsu:Id="ReplyTo" />
    <wsa:MessageID wsu:Id="MessageID" />
    <wsse:Security>
      ...
```

## Security Policies in Play

**EXAMPLE**

- These then are digitally signed, according to the message policy:

```
<ds:Signature Id="Signature" >
  <ds:SignedInfo>
    <ds:Reference URI="#MessageID" />
    <ds:Reference URI="#ReplyTo" />
    <ds:Reference URI="#To" />
    <ds:Reference URI="#Action" />
    <ds:Reference URI="#Body" />
    <ds:Reference URI="#Timestamp" />
  </ds:SignedInfo>
</ds:Signature>
```

- The advantage of this assertion, then, is that messages governed by this policy will be resistant to redirection/misdirection attacks.
- A final variant combines all three of these additional features – username token, body encryption, and addressing.

`build Symmetric AllTogether`

- The resulting SOAP messages are pretty much the sum of the parts that we've seen from the previous policy variants, but are probably worth a final review.

## Security Policies in Play

**EXAMPLE**

- One last policy departs completely from what we've seen so far, providing for an asymmetric binding.
- See **Delivery/WSDL/Asymmetric.wsdl**:

```
<sp:AsymmetricBinding>
  <wsp:Policy>
    <sp:RecipientToken>
      <wsp:Policy>
        <sp:X509Token ...>...</sp:X509Token>
      </wsp:Policy>
    </sp:RecipientToken>
    <sp:InitiatorToken>
      <wsp:Policy>
        <sp:X509Token ...>...</sp:X509Token>
      </wsp:Policy>
    </sp:InitiatorToken>
  </wsp:Policy>
</sp:AsymmetricBinding>
```

- So here we identify **different keys** and certificates to be used on each side of the exchange.
- Again, in our example this has not much impact since both parties use the same keystores.
- But it will change the way those keys are used: we'll see **asymmetric signature and encryption** directly on message content, instead of an SSL-style wrapping of a symmetric key.
- Most of the rest of the policy is as we've seen it so far: same layout policy, same algorithm suite, etc.

## Security Policies in Play

**EXAMPLE**

- Test this one out, and see **SOAPSummary/Asymmetric.txt** or your sniffer console for the results:

```
POST /Pizza/Delivery HTTP/1.1
```

```
<S:Envelope>
  <S:Header>
    <wsse:Security>
      <wsu:Timestamp wsu:Id="Timestamp" />
      <wsse:BinarySecurityToken wsu:Id="SigningKey" />
      <ds:Signature Id="Signature" >
        <ds:SignedInfo>
          <ds:Reference URI="#Body" />
          <ds:Reference URI="#Timestamp" />
        </ds:SignedInfo>
        <ds:KeyInfo>
          <wsse:SecurityTokenReference>
            <wsse:Reference URI="#SigningKey" />
          </wsse:SecurityTokenReference>
        </ds:KeyInfo>
      </ds:Signature>
    </wsse:Security>
  </S:Header>
  <S:Body wsu:Id="Body" />
</S:Envelope>
```

## Security Policies in Play

**EXAMPLE**

HTTP/1.1 200 OK

```
<S:Envelope>
  <S:Header>
    <wsse:Security>
      <wsu:Timestamp wsu:Id="Timestamp" />
      <ds:Signature Id="Signature" >
        <ds:SignedInfo>
          <ds:Reference URI="#Body" />
          <ds:Reference URI="#Timestamp" />
        </ds:SignedInfo>
        <ds:KeyInfo>
          <wsse:SecurityTokenReference>
            <wsse:KeyIdentifier>SigningKey
          </wsse:KeyIdentifier>
          </wsse:SecurityTokenReference>
        </ds:KeyInfo>
      </ds:Signature>
    </wsse:Security>
  </S:Header>
  <S:Body wsu:Id="Body" />
</S:Envelope>
```

## Implementing Security Policies

---

- So WS-SecurityPolicy defines a means of sharing policies.
- WSIT is one good way for Java applications to implement those policies, and it takes care of almost all of the details for you.
- The application must in some cases provide its own logic, or connect WSIT to other security systems.
- There are two main areas in which these choices must be made:
  - User login: client supplies **credentials**, and service **authenticates**
  - PKI: client and server must define **keystores** and **trust stores**
- On the client side, such things must always be provided explicitly, first by configuration.
- The service can rely on default behaviors from the hosting application server, or in some cases can plug in its own logic.

## Managing Usernames and Passwords

---

- A client must supply username and password whenever the policy calls for a username token.
- Use the `<cc:UsernameCallback>` for one, and the `<cc>PasswordCallback>` for the other.
  - You can either set values in the configuration – this is only appropriate for early-stage development and testing:

```
<cc:CallbackHandlerConfiguration ... >  
  <cc:CallbackHandler name="usernameHandler"  
    default="footballfan" />  
  <cc:CallbackHandler name="passwordHandler"  
    default="hungry" />  
</cc:CallbackHandlerConfiguration>
```

- More robust is to plug in a JAAS handler class:

```
<cc:CallbackHandlerConfiguration ... >  
  <cc:CallbackHandler name="usernameHandler"  
    classname="com.me.MyUsernameHandler" />  
  <cc:CallbackHandler name="passwordHandler"  
    classname="com.me.MyPasswordHandler" />  
</cc:CallbackHandlerConfiguration>
```

- On the service side, WSIT will automatically authenticate against the default realm for the application server, as we've seen.
  - Or, you can configure your own handler class:

```
<sc:ValidatorConfiguration ... >  
  <sc:Validator name="usernameValidator"  
    classname="cc.wss.LoginValidator" />  
</sc:ValidatorConfiguration>
```

## Managing Certificates

---

- Most policies will call for at least some sort of cryptography, and this will usually begin with asymmetric encryption, signature, or key wrapping.
- These processes, in turn, require public and private keys.
- On the client side, one must configure a keystore and/or a trust store (depending on the processes required by the policy).
- On the server side, WSIT will rely on the key store and trust store of the application server – and only these.
  - Any `<sc:KeyStore>` or `<sc:TrustStore>` assertions will be ignored.
  - Expect this to be opened up to more configuration in Metro 2.0.
- WSIT also has a quirk by which it requires certain non-standard properties to be set for certificates that it uses.
  - The details aren't really worth the digging, but the upshot is that **keytool** is insufficient to create certificates that will satisfy WSIT.
  - One must use another tool, such as **opensso**.
- For our exercises, we work around this in one of two ways:
  - Configure the client to use the server's own key and trust stores. The Pizza client application does this.
  - Provide a key/trust store for the client that has the necessary extensions to the certificate. Hence **Clinic.jks** (which, shh, is actually a GlassFish certificate, swiped from another installation of the server!).

## Policies for the Healthcare System

**LAB 6**

**Suggested time: 30-60 minutes**

In this lab you will complete a WS-SecurityPolicy for the Healthcare system, assuring that all records requests are signed and that medical history responses are encrypted.

Optionally, you will re-introduce the username-token requirements from the previous (XWSS) version of the application. This will involve some additional policy authoring, and that you refactor the Clinic's **LoginHandler** and write a new **LoginValidator** for the Hospital.

Detailed instructions are found at the end of the chapter.

## Server Authentication

**EXAMPLE**

- In the previous lab, we plugged in our own user authentication validator, which allowed the Hospital to continue to use its own primitive **users.properties** realm.
- Earlier, we saw that WSIT is also happy to let the application server authenticate based on its own realm.
- In fact, we can apply role-based authorization based on user accounts in the application server, as well.
- We see this in a third version of the Housing application – see **Examples/Housing/Step3**.
  - This is not standardized yet – look to JSR 196 and Java EE 6.
  - But Metro provides a nonstandard solution by publishing the Java **Subject** – complete with both user and group **Principals** – as attributes in the web-service context object.
  - So we can read those and check for server-administered group affiliations, treating those as authorization roles.
  - In other words, where we'd like to say, simply:  
`context.isUserInRole ("importantRole")`
  - ... we instead have to do some non-standard coding, as shown in this example.

## Server Authentication

**EXAMPLE**

- This version of the application drops the HTTP BASIC authentication put in place for **Step2** and, having taken that step backward, steps forward in a different direction, which is message-level security with username tokens.
  - The **policy** calls for a symmetric binding with a supporting username token – very close to our Pizza example of username tokens, and also cheating on the PKI by using the server’s keystore.
  - On **client** side, we now provide username and password through a JAAS handler, very much like what the Clinic does.
  - But on the **service** side, we have no login validator; we’ll rely on the application server for **authentication**.
  - For authorization, we have a new helper method in the service implementation class that will read the **Subject** to discover the group(s) of the authenticated caller.
- See **Service/src/cc/housing/HousingSearchImpl.java**.
  - Two of the three web-service methods now call a helper method before proceeding; so we’re saying that only certain users will be allowed to see listing details, while anyone can get summaries:

```
public HousingUnitDetailList searchForHousing (...)  
{  
    checkAccessToDetails ();  
    return query (...);  
}
```

## Server Authentication

**EXAMPLE**

- The helper method first tries the standard methods for programmatic authorization – which would work for HTTP BASIC or DIGEST but are not yet required to tie in to message-level security mechanisms such as username tokens:

```
private void checkAccessToDetails ()
{
    System.out.println ("Username: " +
        context.getUserPrincipal ().getName ());
    System.out.println ("Customer? " +
        context.isUserInRole ("customer"));
}
```

- Then it gets down to business: it gets an attribute from the web-service context whose name is the full name of the **Subject** class. Then it reads through all the **Principals**:

```
List<String> groups = new ArrayList<String> ();
for (Principal candidate :
    ((Subject) context.getMessageContext ()
        .get (Subject.class.getName ()))
        .getPrincipals ())
```

- For each, it looks for the GlassFish Group principal type:

```
if (candidate.getClass ().getName ().equals
    ("com.sun.enterprise.deployment.Group"))
{
    System.out.println ("Group: " +
        candidate.getName ());
    groups.add (candidate.getName ());
}
```

## Server Authentication

**EXAMPLE**

- Finally it asserts that the name “customer” be found in the list of groups that it has derived:

```
if (!groups.contains ("customer"))
    throw new SecurityException
        ("Not authorized to view unit details.");
}
```

- The application deployment descriptor has been enhanced to map each of two role names to groups in the server.

- See `Service/META-INF/sun-application.xml`:

```
<security-role-mapping>
  <role-name>friend</role-name>
  <group-name>friend</group-name>
</security-role-mapping>

<security-role-mapping>
  <role-name>customer</role-name>
  <group-name>customer</group-name>
</security-role-mapping>
```

- So we’ll test this with two users:
  - **research**, who already has an account with the server from our earlier demonstration, and is in the **friend** group
  - **agency**, who we will create during our testing and who will be in the **customer** group

## Server Authentication

**EXAMPLE**

- Build and deploy the service with **ant**, and build and test the client as we did before:

**ant**

```
run 1000 1 true research barbaloot
javax.xml.ws.soap.SOAPFaultException:
  Not authorized to view unit details.
```

- With the new authorization policy in place, the “research” user is not allowed to get detailed listings – and would have to be refactored to call **getSummaries** to acquire data, or start paying us for access to details.
- In the server console, we see:

```
Username: research (spot-on)
Customer? false (true, but just a lucky guess)
Group: friend
```

## Server Authentication

**EXAMPLE**

- Add a second user to the server, which will be our first customer:

```
ant -Duser=agency -Dpassword=agency123
-Dgroups=customer create-user
Command create-file-user executed successfully.
```

- Test the client again:

```
run 1000 1 true agency agency123
501 Centre Street #1F -- $800/month
26 Hanover Lane -- $650/month
75 Clarendon Street #1A -- $980/month
22 Shirley Road -- $600/month
```

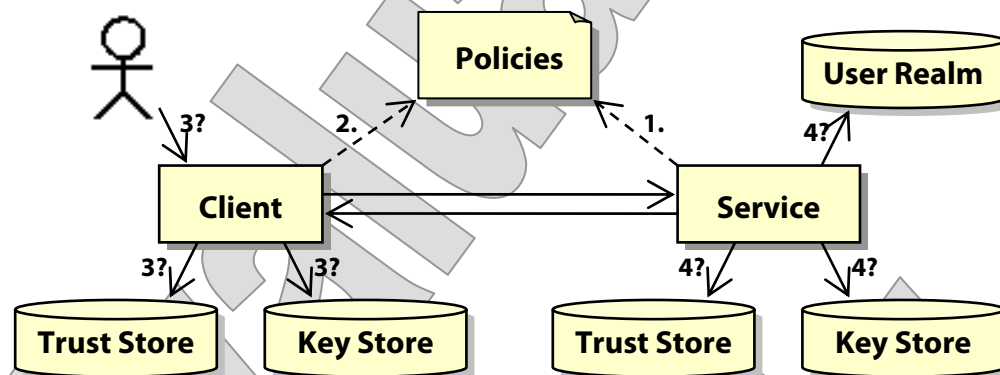
- The server console shows different traces now:

```
Username: agency
Customer? false (oops!)
Group: customer
```

- This shows the difference between using the standard **isUserInRole**, and the **Subject/Principal** interfaces.
  - **isUserInRole** is a standard, but as of Java EE 5 it is not required to reflect bindings based on WS-Security username tokens. So it returns **false** every time – not helpful.
  - The **Subject** information provided by Metro is non-standard, both in the context attribute name used to derive it and the principal type that we seek, which is unique to GlassFish.
  - But it gets the job done!

## Integrating Security Frameworks

- Many software enterprises have one or more pre-existing systems in place for user management, authentication, authorization, etc.
  - Kerberos
  - LDAP
  - Etc.
- Ideally, web services and SOAs will get full leverage from those other systems – but how, exactly?



- It's the system of pluggable handlers and validators that's meant to act as a bridge to these other tools and frameworks.
- We've kept it to username/password logins and X.509 certificates, and will continue to do so in the rest of the course.
- But solutions exist for Kerberos, LDAP, and many other externalities, and where they can't be found, they can be built.

## SUMMARY

- **WS-Policy and WS-SecurityPolicy provide a critical piece of the message-level-security puzzle: a way to share metadata.**
- **Policies can express all the detail that services and clients will need in order to interoperate.**
- **In this way it plays a role quite similar to WSDL for SOAP message bodies.**
- **And the relationship between WSDL and WS-SecurityPolicy is naturally going to be a close one, as policies will be attached to WSDL descriptors.**
  - This establishes relationships between policies and their targets.
  - It also provides a transport for the policies: since client and service development will naturally share WSDL descriptors, we can have our policies “ride” the WSDL and so don’t have to create a separate infrastructure for policy publishing and subscription.
- **WSIT and Metro offer a great boost in productivity and maintainability over XWSS for WS-Security implementations, largely because they are driven by WS-SecurityPolicy.**

## Policies for the Healthcare System

**LAB 6**

In this lab you will complete a WS-SecurityPolicy for the Healthcare system, assuring that all records requests are signed and that medical history responses are encrypted.

Optionally, you will re-introduce the username-token requirements from the previous (XWSS) version of the application. This will involve some additional policy authoring, and that you refactor the Clinic's **LoginHandler** and write a new **LoginValidator** for the Hospital.

<b>Lab workspace:</b>	<b>Labs/Lab06</b>
<b>Backup of starter code:</b>	<b>Examples/Healthcare/Step5</b>
<b>Answer folder(s):</b>	<b>Examples/Healthcare/Step6</b> (intermediate) <b>Examples/Healthcare/Step7</b> (final)
<b>Files:</b>	<b>Hospital/WSDL/Hospital.wsdl</b> <b>Clinic/WSDL/Clinic.wsdl</b> <b>Clinic/establishTrust.bat</b> <b>Hospital/src/cc/wss/LoginValidator.java</b> <b>Clinic/src/cc/wss/LoginHandler.java</b> <b>Clinic/src/cc/med/Clinic.java</b>

### Instructions:

1. Review the starter code for the Hospital and Clinic applications. Notice that the XWSS message handlers have been removed from both sides: the **Clinic** class no longer configures a handler chain at all, and the Hospital's **webservices.xml** similarly has removed the handler configuration.  
So we're back to square one – or not quite, because you can also see security policies laid out in **Hospital.wsdl**. These specify a symmetric binding – very much like the one we just explored in the Pizza case study – and a global message policy calling for body signatures.
2. If you like, try building and testing the system as a baseline. You'll see that it runs more or less as the very first step of the case study did, which is to say with no message security. The policies you see in the WSDL haven't been attached to the service in any way, and the XWSS code from the previous step has been disconnected as well.

## Policies for the Healthcare System

## LAB 6

3. First, attach these policies to the WSDL port:

```
<binding name="MedicalRecordsBinding" type="h:MedicalRecords">
  <wsp:PolicyReference URI="#HospitalPolicy" />
  <wsp:PolicyReference URI="#MessagePolicy" />
</soap:binding>
```

4. Build and deploy the Hospital. If you test the Clinic application now, none of the commands will work, because the service's security requirements go unmet.

5. Open **Clinic.wsdl** and make similar attachments – this time using remote URLs:

```
<binding name="MedicalRecordsBinding" type="h:MedicalRecords">
  <wsp:PolicyReference
    URI="http://localhost:8080/Hospital/Records?WSDL#HospitalPolicy"
  />
  <wsp:PolicyReference
    URI="http://localhost:8080/Hospital/Records?WSDL#MessagePolicy"
  />
</soap:binding>
```

6. Build and test the Clinic – we're not quite there yet:

```
ant
```

```
run confirm markinson schmarkinson
```

```
SEVERE: Could not locate TrustStore, check truststore assertion in WSIT
configuration
```

WSIT is running in the client process, and when it sees the policy requirements, it goes looking for a trust store and a "peer alias," which is the alias in the trust store of the certificate of the service, which it will need in order to wrap a generated key. But, so far, we've not told WSIT where to look for that trust store, or what that alias is.

7. You've probably already noticed the one policy in **Clinic.wsdl**:

```
<wsp:Policy wsu:Id="Credentials" >
  <cc:TrustStore
    wssp:visibility="private"
    location="build\classes\cc\wss\Clinic.jks"
    storepass="holdthemayo"
    peeralias="hospital"
  />
</wsp:Policy>
```

Simply add this to the list of policies attached to the port – for the client's purposes only:

```
<wsp:PolicyReference
  URI="http://localhost:8080/Hospital/Records?WSDL#MessagePolicy"
/>
<wsp:PolicyReference URI="#Credentials" />
</soap:binding>
```

## Policies for the Healthcare System

## LAB 6

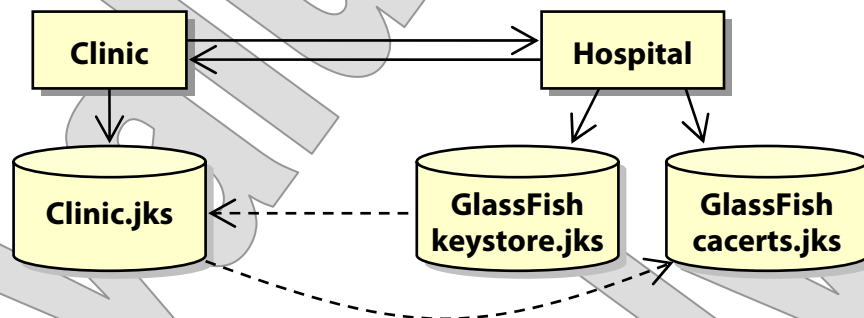
8. Save this file and test again. Note that you don't need to rebuild in this case; the generated JAX-WS proxy encodes the location of the original WSDL, and so your changes will be picked up automatically each time the proxy is created.

But, still no joy:

```
run confirm markinson schmarkinson
```

```
SEVERE: WSS0221: Unable to locate matching certificate for Key Encryption using Callback Handler.
```

Now we have a trickier problem, and one that matches the real world a little more closely than the simplistic keystore-sharing approach in the earlier examples. We're using a pre-built **Clinic.jks**, and of course it has no entry that would match the GlassFish certificate as installed on your machine. This is a miniature version of the general problem of key infrastructure, and of developing trust relationships between entities and security domains. To establish trust between the Clinic and the Hospital (which really means between the Clinic and the Hospital's application server), we need to trade certificates: the GlassFish certificate must be imported into **Clinic.jks**, and the Clinic certificate must be imported into the server's **cacerts.jks**.



9. Happily – oh, so happily, as this is not a fun task to carry out manually – we have a batch file prepared for this purpose. Simply run this batch file ...

```
establishTrust
```

```
...
Certificate stored in file <clinic.crt>
...
Certificate was added to keystore
...
Certificate stored in file <GF.crt>
...
Certificate was added to keystore
```

10. Build – you do have to build this time, to carry the altered **Clinic.jks** into your application's class path – and test again. Again you see a clean round-trip:

```
run confirm markinson schmarkinson
```

```
Confirming patient info ... confirmed.
```

## Policies for the Healthcare System

## LAB 6

11. Now add a policy to **Hospital.wsdl** – this will re-establish the requirement that patients' medical histories be encrypted in transit back to the requester:

```
<wsp:Policy wsu:Id="MessagePolicy_MedicalHistory">
  <sp:EncryptedParts>
    <sp:Body/>
  </sp:EncryptedParts>
</wsp:Policy>
```

12. Attach this policy, not to the port, but to the **getHistory** response:

```
<operation name="getHistory">
  <input>
    <soap:body ... />
  </input>
  <output>
    <wsp:PolicyReference URI="#MessagePolicy_MedicalHistory" />
    <soap:body ... />
  </output>
</operation>
```

13. Build and deploy the Hospital.

14. Test the Clinic against this new policy, running the traffic through the SOAPSniffer, and see that the response is indeed confidential.

```
run getHistory markinson schmarkinson (all on one line)
http://localhost:8079/Hospital/Records
Getting patient history ...
SSID    : 987654321
Name    : Michael Prim
...
```

The **getHistory.txt** in **Examples/Healthcare/Step6/SOAP** shows this as well.

15. Finally, notice that the username and password don't matter. Try any values here. You can't omit them, because the Clinic does use them internally, but they play no role in messaging at the moment:

```
run confirm x y
Confirming patient info ... confirmed.
```

This is the intermediate answer in **Step6**.

## Policies for the Healthcare System

## LAB 6

### Optional Steps

16. Add a username-token requirement to the Hospital policy. You can use the **Symmetric\_UsernameToken.wsdl** in **Examples/Pizza/WS-SecurityPolicy/Delivery/WSDL** as an example: in fact you can copy the `<sp:SignedSupportingTokens>` tree verbatim and place it at the bottom of the first policy in **Hospital.wsdl**.
17. Now, how will usernames and passwords be generated and authenticated? First, connect a pre-built validator on the service side. To do this you need to define an additional, “private” policy, and include a callback configuration in it:

```
<wsp:Policy wsu:Id="Private">
  <sc:ValidatorConfiguration wsp:visibility="private" >
    <sc:Validator name="usernameValidator"
      classname="cc.wss.LoginValidator" />
  </sc:ValidatorConfiguration>
</wsp:Policy>
```

You may want to review the code for this validator class. Mostly it’s ugly JAAS code, and some of it is boilerplate to support something that WSIT itself doesn’t support yet, which is hashed passwords. But the key bit is that it takes the username and password and checks them against **users.properties**, just as the XWSS version of the service did before. In fact, this is exactly the handler that was used in the earlier version; it was just serving as a delegate of **WSSMessageHandler** for certain callback types, where now it’s being configured directly as a WSIT handler in its own right.

18. Attach this policy along with the other two to the **MedicalRecordsBinding**.
19. Build and deploy the Hospital.
20. On the client side, we already have a **LoginHandler** – though in fact it’s not being used right now. We have to do two things:
  - Refactor it to respond to JAAS name/password callbacks, as WSIT will send these instead of the XWSS-specific ones.
  - Make the **username** and **password** fields static, since we don’t have access to the actual handler instance anymore.

**Policies for the Healthcare System****LAB 6**

21. First, replace **com.sun.xml.wss.impl.callback.UsernameCallback** with **javax.security.auth.callback.NameCallback** throughout the source file.
22. Do the same with **PasswordCallback** – here the class name is the same and only the package name changes.
23. When calling **setPassword**, you must now pass a character array, where a simple string was okay before: make the argument **password.toCharArray()**.
24. Make both fields **public** and **static**.
25. Remove the constructor: it was there to allow initialization of the fields, but we'll go a different way now.
26. Now open **Clinic.jks** and initialize the handler. Before, we created a new instance and passed values to the constructor; instead, in **initializeService**, set the static fields directly:

```
LoginHandler.username = username;  
LoginHandler.password = password;
```

27. Build and test the Clinic. You should now find that the username and password are again significant, and if you look at the SOAP traffic you'll see the username token is included in the message – though it's not so easy to spot, because WSIT will encrypt it, as we've seen in earlier examples.

This is the final answer in **Step7**.