

## Chapter - Signing and encrypting SOAP messages

### Objectives

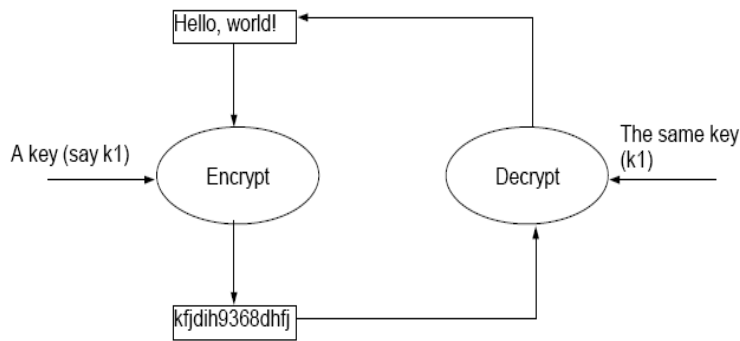
---

- At the end of this chapter you will be able to understand
  - ♦ How to sign and encrypt SOAP messages

## Private key and public key

---

- Usually when we encrypt some text using a key, we need the same key to decrypt it:



3

**seed**  
beyond the obvious

## Private key and public key

---

- This is called "*symmetric encryption*".
- If somebody would like to send something to us in private, then we need to agree on a key.
- If we need to send something private to 100 individuals, then we'll need to negotiate with each such individual to agree on a key (so 100 keys in total).
- This is troublesome.

4

**seed**  
beyond the obvious

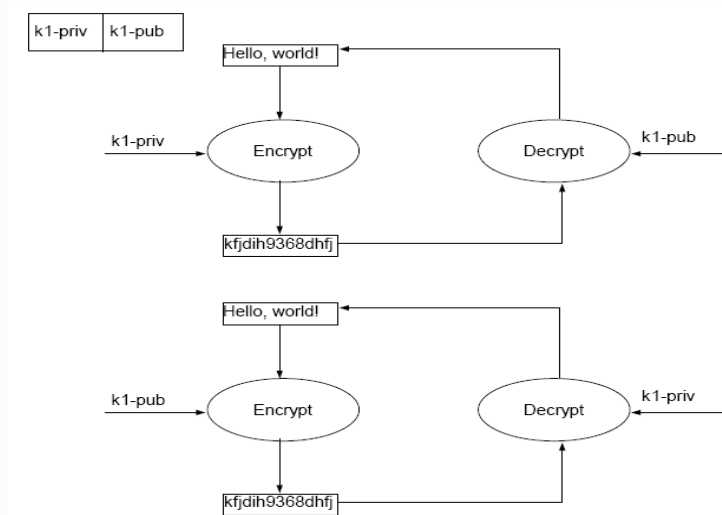
## Private key and public key

- To solve the problem, an individual may use something called a "*private key*" and a "*public key*".
- First, he/she uses some software to generate a pair of keys: One is the private key and the other is the public key.
- There is an interesting relationship between these two keys: If we use the private key to encrypt something, then it can only be decrypted using the public key (using the private key won't work).
- The reverse is also true: If we use the public key to encrypt something, then it can only be decrypted using the private key:

5

**seed**  
beyond the obvious

## Private key and public key



6

**seed**  
beyond the obvious

## Private key and public key

---

- After generating the key pair, he/she will keep the private key really private (won't tell anyone), but he/she will tell everyone his/her public key.
- Can other people find out the private key from the public key??
- It is extremely difficult, so there is no worry about it.

7

**seed**  
beyond the obvious

## Private key and public key

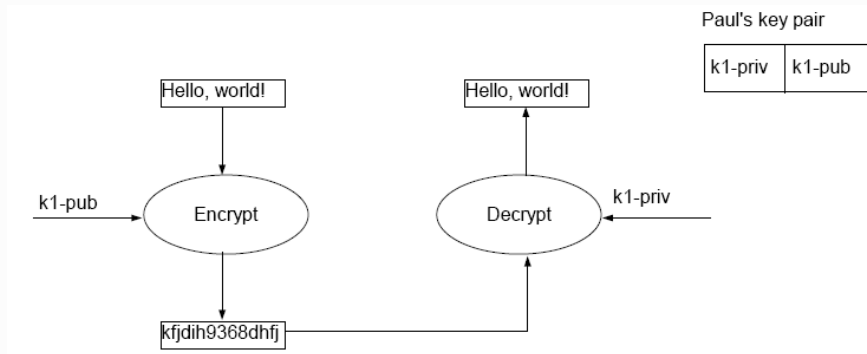
---

- Now, suppose that we'd like to send something confidential to an individual say Paul, we can use his public key to encrypt it.
- Even though other people know his public key, they can't decrypt it (as it is encrypted using the public key, only the private key can decrypt it).
- Only Paul knows the private key and so only he can decrypt it:

8

**seed**  
beyond the obvious

## Private key and public key



- This kind of encryption is called "*asymmetric encryption*".

9

**seed**  
beyond the obvious

## Digital signature

- Suppose that the message we send to Paul is not confidential. However, Paul really needs to be sure that it is really from us.
- How to do that??

10

**seed**  
beyond the obvious

## Digital signature

---

- We need to prove to Paul that the creator of the message (we) knows the private key
- To prove that, we can use our private key to encrypt the message, then send it to Paul. Paul can try to decrypt it using our public key. If it works, then the creator of the message must know the private key and must be we only.

11



## Digital signature

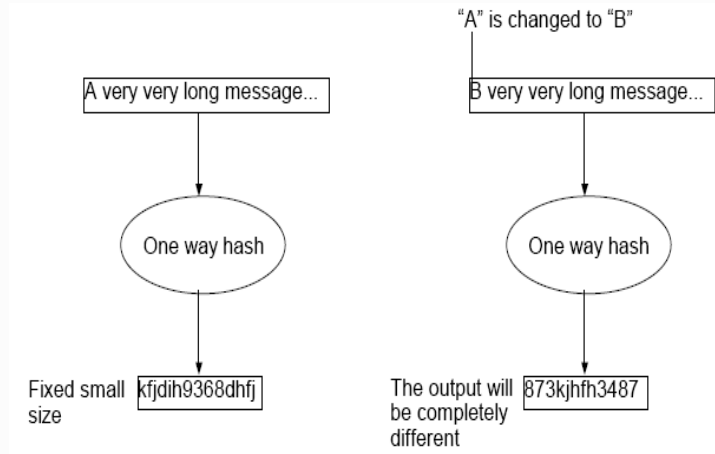
---

- However, this is not a good solution, because if the message is long, the encrypted message may double in size and the encryption takes a lot of time.
- To solve this problem, we can feed the message to a "*one way hash function*". No matter how long the input is, the output from the one way hash function is always the same small size (e.g., 128 bits).
- In addition, if two input messages are different (maybe just a single bit is different), then the output will be completely different. Therefore, the output message can be considered a **small-sized snapshot** of the input message.
- It is therefore called the "*message digest*" of the original message:

12



## Digital signature



13

**seed**  
beyond the obvious

## Digital signature

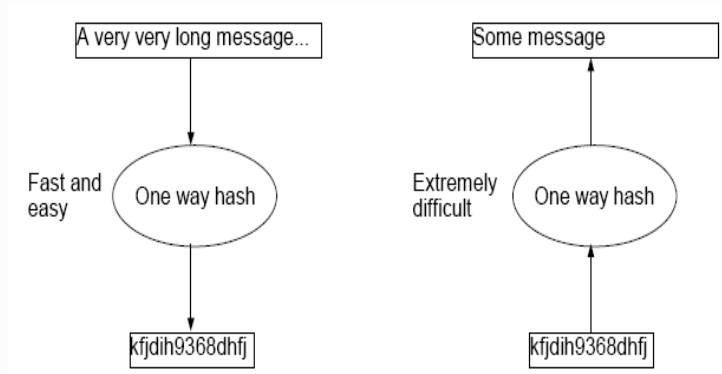
- Another feature of the one way hash function is that it is very fast to calculate the digest of a given message, but it is extremely difficult to calculate a message given a digest.
- Otherwise people would find different messages for a given digest and it is no longer a good snapshot for the message:

14

**seed**  
beyond the obvious

## Digital signature

---



15

**seed**  
beyond the obvious

## Digital signature

---

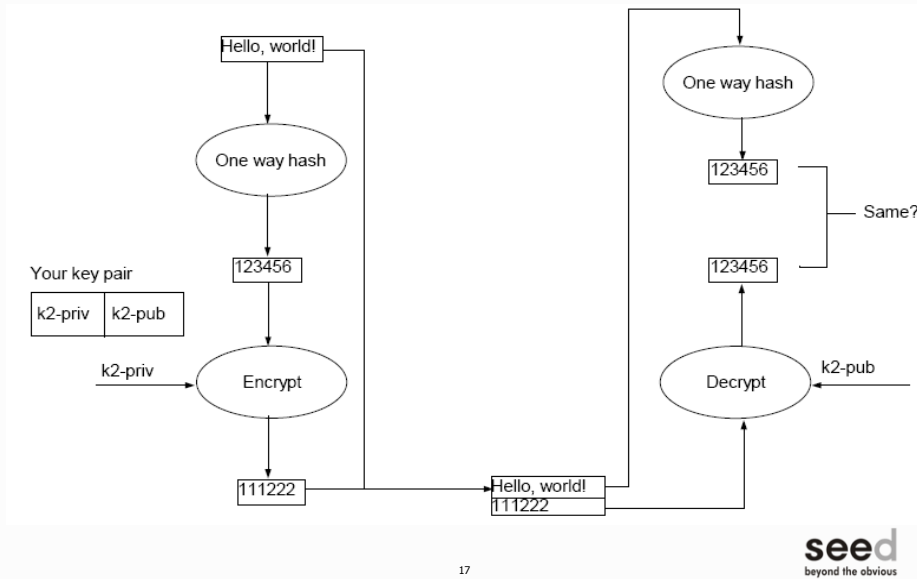
- Now, to prove to Paul that we know our private key, we can use our private key to encrypt the message digest (because the digest is small, the result is also small and the encryption process will be fast), then send both the message and the message digest to Paul.
- He can try to decrypt the digest using our public key. Then he can calculate the digest from the message and compare the two. If the two match, then the person producing the encrypted digest must be we:

16

**seed**  
beyond the obvious



## Digital signature



17

## Digital signature

- The encrypted digest is called the "*digital signature*". The whole process of calculating the digest and then encrypting it is called "*signing the message*".

18

seed  
beyond the obvious

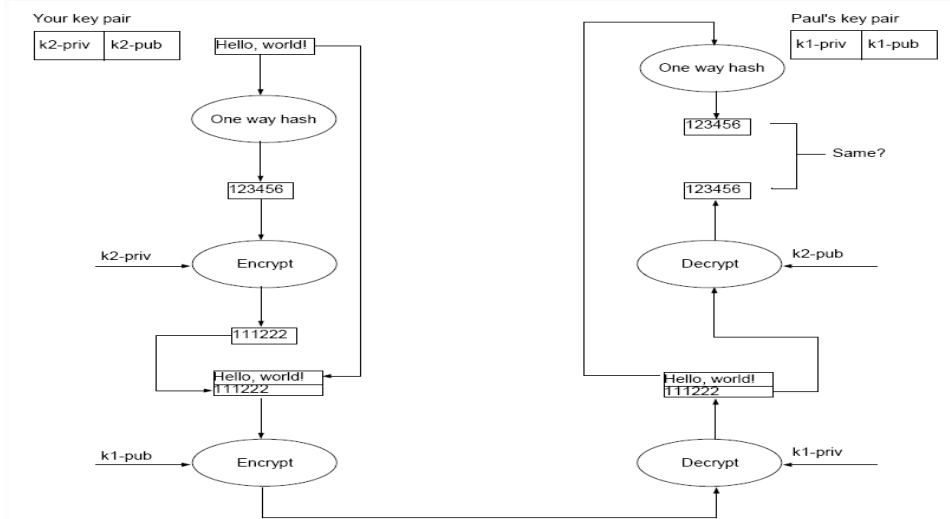
## Signing and encrypting

- What if we'd like to sign the message, while keeping the message available to Paul only??
- Just sign it as usual and then encrypt the message and the digest using Paul's public key.
- When Paul receives it, he uses his private key to decrypt it and then go on to verify the signature as usual:

19

seed  
beyond the obvious

## Signing and encrypting



20

seed  
beyond the obvious

## Certificate and CA

---

- This seems to work very well. However, when we need to say send a confidential message to Paul, we'll need his public key. But how can we find out his public key??

21



## Certificate and CA

---

- We can call him (Paul) on the phone to ask him. But how can we be sure that the person on the phone is really Paul? If he/she is a hacker, he/she will tell us his/her public key.
- When we send the message to Paul using the hacker's public key, the hacker will be able to decrypt it using his/her private key.
- If we need to communicate with many different individuals, this will get even more troublesome.

22



## Certificate and CA

---

- To solve the problem, Paul may go to a government authority, show his ID card and etc and tell the authority his public key.
- Then the authority will generate an electronic message (like an email) stating Paul's public key.
- Finally, it (government authority) signs that message using its own private key:

Name: Paul
Public key: 666888
Signature

23

**seed**  
beyond the obvious

## Certificate and CA

---

- Such a signed message is called a "*certificate*". That authority is called a "*certificate authority (CA)*".
- Then Paul can put his certificate on his personal web site, email it to us directly or put it onto some 3rd party public web site.
- From where we get the certificate is unimportant. What is important is that if we can verify the signature of that CA and we trust what the CA says, then we can trust that public key in the certificate.

24

**seed**  
beyond the obvious

## Certificate and CA

---

- In order to verify the signature, we will need the public key of that CA.
- What?!
- Aren't we back to the origin of the problem?
  - ♦ However, we only need to find out a single public key for a single entity (the CA), not a public key for everyone we need to communicate with.
  - ♦ How to obtain that public key?
  - ♦ Usually it is already configured in our browser or we can download it from a trusted web site, newspaper or other sources that we trust.

25



## Certificate and CA

---

- A CA doesn't really need to be a government authority. It can be well known commercial organizations such as *VeriSign*.
- It means that in order to use asymmetric encryption and digital signature, we need private keys, public keys, a CA and certificates. All these elements combined together is called a "*public key infrastructure (PKI)*" because it provides a platform for us to use public keys.

26



## Distinguished name

---

- If we review the certificate:

Name: Paul
Public key: 666888
Signature

- we will see that it is not that useful because there are probably millions of people named "Paul" in the world.
- Therefore, in a real certificate, usually the country, city and the company of that individual are also included like:

27

**seed**  
beyond the obvious

## Distinguished name

---

	CN means common name	Organization (company)	State	Country
Name:	CN=Paul McNeil, O=Microsoft, ST=WA, C=US			
Public key:	666888			Signature

The whole thing is called  
a "distinguished name  
(DN)"

28

**seed**  
beyond the obvious

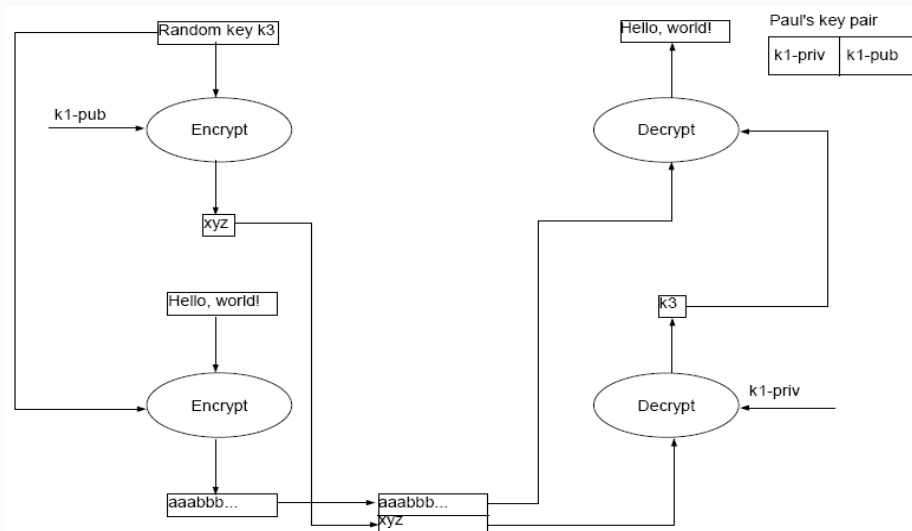
## Performance issue with asymmetric encryption

- Suppose that we'd like to send an encrypted message to Paul. We can use Paul's public key to do that. However, in practice few people would do it this way, because asymmetric encryption is very slow. In contrast, symmetric encryption is a lot faster.
- To solve this problem, we can generate a random symmetric key, use it to encrypt the message, then use Paul's public key to encrypt that symmetric key and send it to Paul along with the encrypted message.
- Paul can use his private key to get back the symmetric key and then use it to decrypt the message:

29

seed  
beyond the obvious

## Performance issue with asymmetric encryption



30

seed  
beyond the obvious

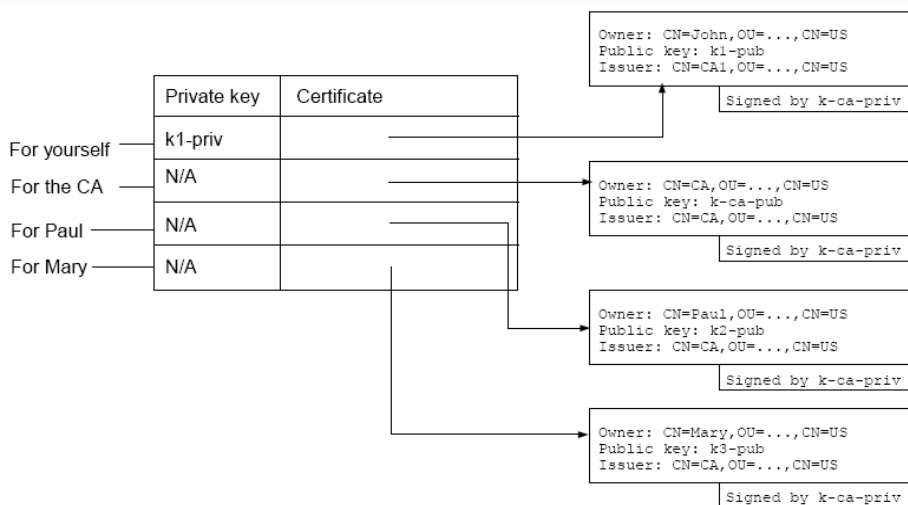
## Keeping key pair and certificates in Java

- In order to use PKI, typically we should have
  - ♦ A private key for our self
  - ♦ A certificate for our self so that we can send to others
  - ♦ A certificate for each person that we need to send something confidential to (e.g. Paul and Mary)
  - ♦ The public keys of the CA's that we trust.
- For the public key of the CA, we don't directly store its public key. Instead, we store its certificate which contains its public key. But who issued that certificate to it? It was issued by itself (signed by its own private key):

31

**seed**  
beyond the obvious

## Keeping key pair and certificates in Java



32

**seed**  
beyond the obvious



## Keeping key pair and certificates in Java

- Such a table is called a "keystore" in Java.
- A *keystore* is stored in a file. In addition, each entry in the table has a name called the "*alias*" of the entry.
- This way we can, e.g., tell the software to sign a particular message using the private key in the "john" entry (our self), or encrypt the message using the public key in "paul" entry.
- Without the alias we will have to use the DN to refer to an entry:

33

**seed**  
beyond the obvious

## Keeping key pair and certificates in Java

keystore

Alias	Private key	Certificate
john	k1-priv	_____→
CA	N/A	_____→
paul	N/A	_____→
mary	N/A	_____→

34

**seed**  
beyond the obvious

## Installing Rampart

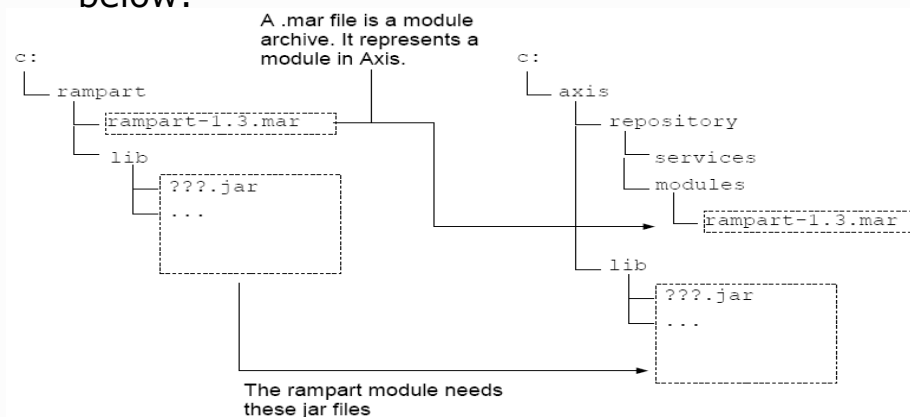
- In order to perform signing or encryption, we need an Axis module called "Rampart".
- Suppose that it is rampart-1.3.zip. Unzip it into say c:\rampart.
- Rampart needs another library xalan 2.7.0. If we're using JDK 5 or earlier, we probably have only an old version. So, in that case, download xalan-2.7.0.jar and put it into c:\rampart\lib.

35

**seed**  
beyond the obvious

## Installing Rampart - On server side

- To make rampart available to our web services at runtime, copy all the files shown below:



36

**seed**  
beyond the obvious

## Installing Rampart – On client side

---

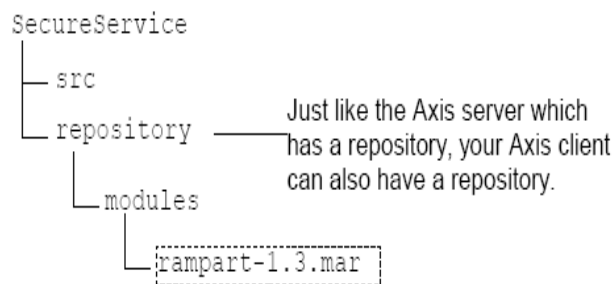
- To make the rampart module available to our client code, add the jar files in c:\rampart\lib to the build path of our project and copy rampart-1.3.mar into our project in such a folder structure:

37



## Installing Rampart – On client side

---



38



## Updated build.xml

```
<project basedir="." default="jar.server">
  ...
  <property name="name" value="SecureService" />
  ...
  <target name="generate-service">
    <wsdl2code
      wsdlfilename="${name}.wsdl"
      serverside="true"
      generateservicexml="true"
      skipbuildxml="true"
      serversideinterface="true"
      namespacepackages="http://ttdev.com/ss=com.ttdev.secure"
      targetsourcefolderlocation="src"
      targetresourcefolderlocation="src/META-INF"
      overwrite="true"
      unwrap="true" />
    <replaceregexp
      file="src/META-INF/services.xml"
      match="${name}Skeleton"
      replace="${name}Impl" />
  </target>
  <target name="generate-client">
    <wsdl2code
      wsdlfilename="${name}.wsdl"
      skipbuildxml="true"
      namespacepackages="http://ttdev.com/ss=com.ttdev.secure.client"
      targetsourcefolderlocation="src"
      overwrite="true"
      unwrap="true" />
  </target>
</project>
```

39



## Signing SOAP messages

- In order to sign the SOAP messages, modify the WSDL file:

40



# Signing SOAP messages

It belongs to the web service policy namespace

```
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://ttdev.com/ss"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy"
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wsp1="http://docs.oasis-open.org/ws-sx/2004/01/oasis-200401-ws-
  wsssecurity-utility-1.0.xsd"
  name="SecureService">
  <wsdl:policy wsu:id="p1" />
  <sp:SignedParts>
    <sp:Body />
  </sp:SignedParts>
  <wsp:Policy>
    <wsdl:types>
      <wsdl:message name="concatRequest">
        <wsdl:sequence base="xsd:string" minOccurs="1" maxOccurs="1" />
      </wsdl:message>
      <wsdl:message name="concatResponse">
        <wsdl:sequence base="xsd:string" minOccurs="1" maxOccurs="1" />
      </wsdl:message>
      <wsdl:portType name="SecureService">
        <wsdl:operation name="concat">
          <wsdl:input message="tns:concatRequest" />
          <wsdl:output message="tns:concatResponse" />
        </wsdl:operation>
      </wsdl:portType>
      <wsdl:binding name="SecureServiceSOAP" type="tns:SecureService">
        <soap:binding style="document" />
        <wsdl:operation name="concat">
          <wsdl:input />
          <wsdl:output />
        </wsdl:operation>
      </wsdl:binding>
      <wsdl:service name="SecureService">
        <wsdl:port binding="tns:SecureServiceSOAP" />
      </wsdl:service>
    </wsdl:definitions>
```

This is a "policy". A policy specifies non-functional requirements of the web service (e.g., security, quality of service). The syntax of specifying a policy is governed by the WS-Policy standard.

This is a "policy assertion". It requires certain parts of the SOAP message be signed.

The parts should be signed and are listed here. Here, only the -Body- of the SOAP message should be signed.

Apply the policy "p1" to the SOAP binding of the concat operation. It means the -Body- of all the messages for the concat operation must be signed as long as they're using SOAP over HTTP. Without this the policy would be sitting there idle and would have no effect.

As the -PolicyReference- element belongs to a foreign namespace (wsp), there is no guarantee that the program processing the WSDL file (e.g. -wsdl2code-) understands it. This attribute requires that the program understand it, otherwise it should abort the processing.

If you had multiple operations in the port type and they all required signed messages, you would move the -PolicyReference- to there so that it would apply to the SOAP binding of the SecureService port type.

41



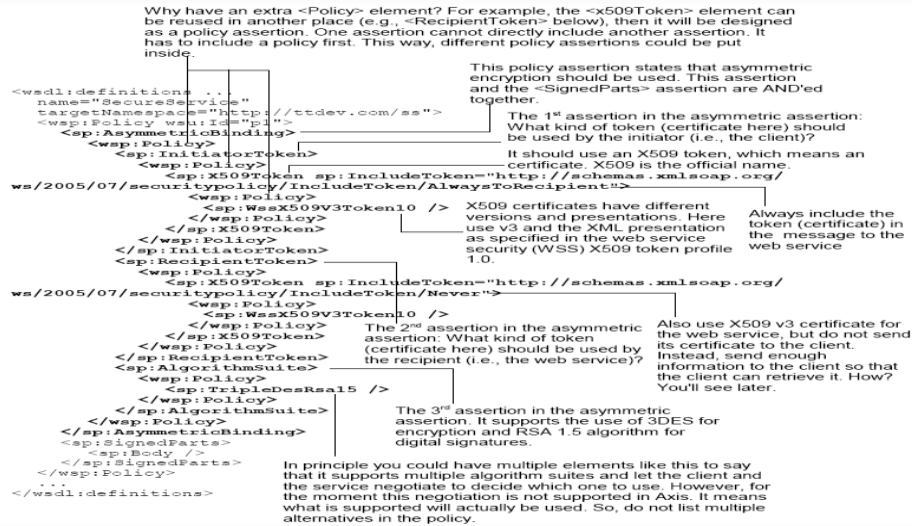
# Signing SOAP messages

- Saying that the <Body> should be signed is not enough. We still need to specify that asymmetric encryption should be used and what signature algorithms are supported and etc.:

42



# Signing SOAP messages



43

**seed**  
beyond the obvious

# Signing SOAP messages

- Finally, we still need to say that it supports the Web Service Security (WSS) standard v1.0:

44

**seed**  
beyond the obvious

## Signing SOAP messages

```
<wsdl:definitions ...
  name="SecureService"
  targetNamespace="http://ttdev.com/ss">
<wsp:Policy wsu:Id="p1">
  <sp:AsymmetricBinding>
    ...
  </sp:AsymmetricBinding>
<sp:Wss10> ————— Supports WSS 1.0
  <wsp:Policy>
    <sp:MustSupportRefEmbeddedToken /> ——— It can deal with tokens (certificates)
    <sp:MustSupportRefIssuerSerial /> ——— directly included in the messages
  </wsp:Policy>
  </sp:Wss10>
  <sp:SignedParts>
    <sp:Body />
  </sp:SignedParts>
</wsp:Policy>
...
</wsdl:definitions>
```

It can also use the issuer DN and serial number to look up the certificate

45



## Signing SOAP messages

- Generate the service stub and client stub. Fill out the code in the implementation class:

```
public class SecureServiceImpl implements SecureServiceSkeletonInterface {
    public String concat(String s1, String s2) {
        return s1 + s2;
    }
}
```

- Create *SecureClient.java* in the client package:

46



## Signing SOAP messages

```
import org.apache.axis2.context.ConfigurationContext;
import org.apache.axis2.context.ConfigurationContextFactory;

public class SecureClient {
    public static void main(String[] args) throws RemoteException {
        ConfigurationContext context = ConfigurationContextFactory
            .createConfigurationContextFromFileSystem("repository");
        SecureServiceStub stub = new SecureServiceStub(context);
        stub._getServiceClient().engageModule("rampart");
        String result = stub.concat("xyz", "111");
        System.out.println(result);
    }
}
```

Tell the Axis client to load configurations from the "repository" folder in the current folder (project root). Here it will find the module archive for rampart.

Having rampart available is not enough, you must engage it.

47

**seed**  
beyond the obvious

## Signing SOAP messages

- For rampart to sign the <Body>, it needs access to the policy. Fortunately <wsdl2code> has extracted the policy information from the WSDL and put it into the Java code generated.
- What is missing is, what is the alias of the certificate to use, the password, the location of the keystore and etc.
- All this information can be specified in a Java String or in a text file. Here, let's put it into a text file **rampart-config.xml** in the project root:

48

**seed**  
beyond the obvious



## Signing SOAP messages

The rampart configuration happens to be also in the form of a policy, although it is supposed to be used by the client itself.

All the other elements here are in the rampart namespace

```
<?xml version="1.0" encoding="UTF-8"?>
<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns="http://ws.apache.org/rampart/policy">
  <RampartConfig>
    <user>cl</user>
    <passwordCallbackClass>
      com.ttdev.secure.client.PasswordCallbackHandler
    </passwordCallbackClass>
    <signatureCrypto>
      <crypto
        provider="org.apache.ws.security.components.crypto.Merlin">
          <property
            name="org.apache.ws.security.crypto.merlin.keystore.type">
            JKS
          </property>
          <property
            name="org.apache.ws.security.crypto.merlin.file">
            c:/keys/client.ks
          </property>
          <property
            name="org.apache.ws.security.crypto.merlin.keystore.password">
            client-ks-pass
          </property>
        </crypto>
      </signatureCrypto>
    </RampartConfig>
  </wsp:Policy>
```

The alias of the entry in the keystore. Use its private key to sign the message.

It will create an instance of this class and ask it for the password

Configurations for signing

A Java keystore supports different formats. JKS is the default.

The path to the keystore

The keystore password

Three properties for Merlin only. It has the concept of keystore (a Java concept) and etc.

Rampart uses a cryptographic provider to perform signing, encryption and etc. You specify the class of the provider to use this. Here you're telling it to use the Merlin provider which comes with rampart and uses the JDK to perform these tasks.

49

**seed**  
beyond the obvious

## Signing SOAP messages

- To load the configuration file into rampart, modify the *SecureClient.java*:

50

**seed**  
beyond the obvious

## Signing SOAP messages

```
import org.apache.axiom.om.impl.builder.StAXOMBuilder;
import org.apache.axis2.description.PolicyInclude;
import org.apache.neethi.Policy;
import org.apache.neethi.PolicyEngine;

public class SecureClient {
    public static void main(String[] args) throws RemoteException,
        FileNotFoundException, XMLStreamException {
        ConfigurationContext context = ConfigurationContextFactory
            .createConfigurationContextFromFileSystem("repository");
        SecureServiceStub stub = new SecureServiceStub(context);
        stub.getServiceClient().engageModule("rampart");
        StAXOMBuilder builder = new StAXOMBuilder("rampart-config.xml");
        OMElement configElement = builder.getDocumentElement();
        Policy rampartConfig = PolicyEngine.getPolicy(configElement);
        stub.getServiceClient().getAxisService().getPolicyInclude()
            .addPolicyElement(PolicyInclude.SERVICE_POLICY, rampartConfig);
        String result = stub.concat("xyz", "111");
        System.out.println(result);
    }
}
```

Load the rampart-config.xml file and get the <Policy> element

Convert the <Policy> XML element into a Policy Java object

This AxisService object represents your web service as it is described by the WSDL (including the policy in there)

Add that Policy object to the existing policy. Apply this extra Policy to the whole web service.

51

**seed**  
beyond the obvious

## Signing SOAP messages

- Of course we need to create a *PasswordCallbackHandler* class in the client package:

```
public class PasswordCallbackHandler implements CallbackHandler {
    public void handle(Callback[] callbacks)
        throws IOException, UnsupportedCallbackException {
        for (int i = 0; i < callbacks.length; i++) {
            WSPasswordCallback pwcb = (WSPasswordCallback) callbacks[i];
            String id = pwcb.getIdentifer();
            if (id.equals("c1")) {
                pwcb.setPassword("c1-pass");
            }
        }
    }
}
```

52

**seed**  
beyond the obvious

## Signing SOAP messages

---

- We may wonder why it is so complicated just to tell it the password and why not just specify the password in the rampart-config.xml file??
- To answer it: It is so that we can look it up in a database and etc.

53



## Signing SOAP messages

---

- Now launch the TCP Monitor and let it listen on port 1234. For it to work, specify the port 1234 in the client:

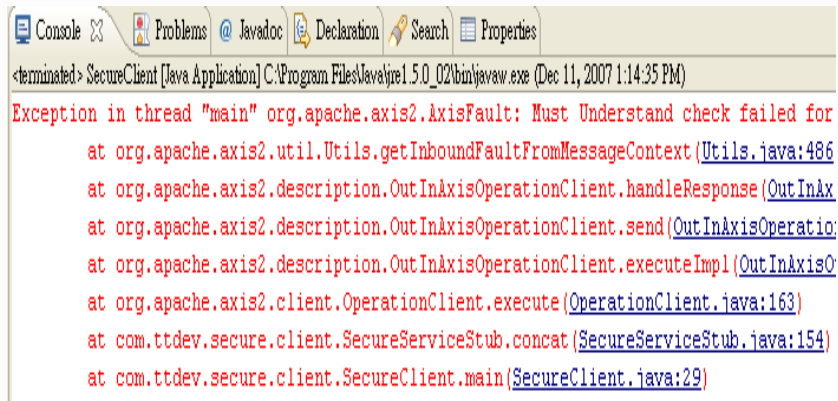
```
public class SecureClient {
    public static void main(String[] args) throws RemoteException,
        FileNotFoundException, XMLStreamException {
        ConfigurationContext context = ConfigurationContextFactory
            .createConfigurationContextFromFileSystem("repository");
        SecureServiceStub stub = new SecureServiceStub(context,
            "http://localhost:1234/axis2/services/SecureService");
        stub._getServiceClient().engageModule("rampart");
        StAXOMBuilder builder = new StAXOMBuilder("rampart-config.xml");
        OMElement configElement = builder.getDocumentElement();
        Policy rampartConfig = PolicyEngine.getPolicy(configElement);
        stub._getServiceClient().getAxisService().getPolicyInclude()
            .addPolicyElement(PolicyInclude.SERVICE_POLICY, rampartConfig);
        String result = stub.concat("xyz", "111");
        System.out.println(result);
    }
}
```

54



## Signing SOAP messages

- Run it and we will see an error in the console saying the a header was not understood:



The screenshot shows an IDE console window with the following content:

```
<terminated> SecureClient [Java Application] C:\Program Files\Java\jre1.5.0_02\bin\javaw.exe (Dec 11, 2007 1:14:35 PM)

Exception in thread "main" org.apache.axis2.AxisFault: Must Understand check failed for
    at org.apache.axis2.util.Utils.getInboundFaultFromMessageContext (Utils.java:486)
    at org.apache.axis2.description.OutInAxisOperationClient.handleResponse (OutInAxisOperationClient.java:163)
    at org.apache.axis2.description.OutInAxisOperationClient.send (OutInAxisOperationClient.java:154)
    at org.apache.axis2.description.OutInAxisOperationClient.executeImpl (OutInAxisOperationClient.java:143)
    at org.apache.axis2.client.OperationClient.execute (OperationClient.java:163)
    at com.ttdev.secure.client.SecureServiceStub.concat (SecureServiceStub.java:154)
    at com.ttdev.secure.client.SecureClient.main (SecureClient.java:29)
```

55



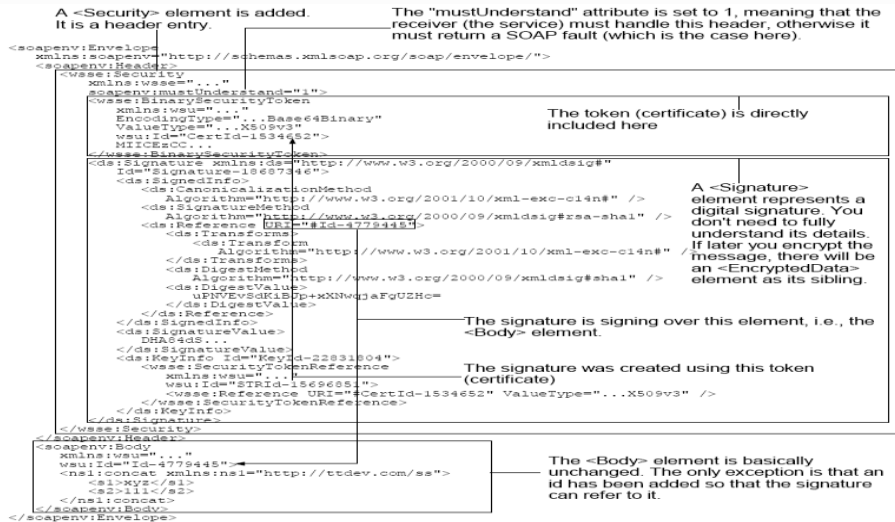
## Signing SOAP messages

- This is fine as the web service is not yet prepared to handle the digital signature.
- What is interesting is in the request message as shown in the TCP Monitor:

56



# Signing SOAP messages



57



## Supporting digital signatures in the web service

- Ideally, when generating the service stub, <wsdl2code> should consult the policy in the WSDL and setup rampart properly. However, the current version of Axis is not doing that. That's why the web service is not understanding the <Security> header element.
- To fix the problem, add the policy to *services.xml*:

58



## Supporting digital signatures in the web service

```
<?xml version="1.0" encoding="UTF-8"?>
<serviceGroup>
  <service name="SecureService">
    <messageReceivers>
      <messageReceiver mep="http://www.w3.org/ns/wsd1/in-out"
        class="com.ttdev.secure.SecureServiceMessageReceiverInOut" />
    </messageReceivers>
    <parameter name="ServiceClass">
      com.ttdev.secure.SecureServiceImpl
    </parameter>
  </service>
</serviceGroup>
```

59

**seed**  
beyond the obvious

## Supporting digital signatures in the web service

```
</parameter>
<parameter name="useOriginalwsdl">true</parameter>
<parameter name="modifyUserWSDLPortAddress">true</parameter>
<operation name="concat"
  mep="http://www.w3.org/ns/wsd1/in-out">
  <actionMapping>
    http://ttdev.com/ss/NewOperation
  </actionMapping>
  <outputActionMapping>
    http://ttdev.com/ss/SecureService/concatResponse
  </outputActionMapping>
</operation>
<wsp:Policy
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy"
  xmlns:wsu="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wsa="http://docs.oasis-open.org/wss/2004/01/
  oasis-200401-wss-wssecurity-utility-1.0.xsd"
  wsa:Id="p1">
  <sp:AsymmetricBinding>
    <wsp:Policy>
      <sp:InitiatorToken>
        <wsp:Policy>
          <sp:X509Token
            sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/
            securitypolicy/IncludeToken/AlwaysToRecipient">
            <wsp:Policy>
              <sp:WsX509V3Token10 />
            </wsp:Policy>
          </sp:X509Token>
        </wsp:Policy>
      </sp:InitiatorToken>
      <sp:RecipientToken>
        <wsp:Policy>
          <sp:X509Token
            sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/
            securitypolicy/IncludeToken/AlwaysToRecipient">
            <wsp:Policy>
              <sp:WsX509V3Token10 />
            </wsp:Policy>
          </sp:X509Token>
        </wsp:Policy>
      </sp:RecipientToken>
      <sp:AlgorithmSuite>
        <wsp:Policy>
          <sp:TripleDesRsa15 />
        </wsp:Policy>
      </sp:AlgorithmSuite>
      <wsp:Policy>
        <sp:AsymmetricBinding>
          <sp:Ws10>
            <wsp:Policy>
              <sp:MustSupportRefEmbeddedToken />
              <sp:MustSupportRefIssuerSerial />
            </wsp:Policy>
          </sp:Ws10>
          <sp:SignedParts>
            <sp:Body />
            <sp:SignedParts>
            </sp:SignedParts>
          </sp:SignedParts>
        </sp:AsymmetricBinding>
      </wsp:Policy>
    </sp:AsymmetricBinding>
  </wsp:Policy>
</sp:Policy>
</service>
</serviceGroup>
```

60

**seed**  
beyond the obvious

## Supporting digital signatures in the web service

- Then engage the rampart module and add the rampart configuration as a policy assertion:

61

seed  
beyond the obvious

## Supporting digital signatures in the web service

```
<?xml version="1.0" encoding="UTF-8"?>
<serviceGroup>
  <service name="SecureService">
    <messageReceivers>
      </messageReceivers>
    <parameter ...>
    <parameter ...>
    <operation name="concat" ...>
    ...
    </operation>
    <module ref="rampart" />
  </service>
</serviceGroup>
```

Engage the rampart module. The ordering of the <module> element doesn't really matter as long as it is directly in the <service> element.

It is used as a policy assertion

You'll create this class later

The keystore password

You'll create this keystore entry later for the web service

You'll create this keystore later

Alias	Private key	Certificate
s1	...	...

```
<wsp:Policy ...>
  <sp:AsymmetricBinding>
  ...
  </sp:AsymmetricBinding>
  <sp:Wss10>
  ...
  </sp:Wss10>
  <sp:SignedParts>
  <sp:Body />
  </sp:SignedParts>
  <RampartConfig xmlns="http://ws.apache.org/rampart/policy">
    <user>s1</user>
    <passwordCallbackClass>
      com.ttdev.secure.PasswordCallbackHandler
    </passwordCallbackClass>
    <signatureCrypto>
      <crypto>
        provider="org.apache.ws.security.components.crypto.Merlin">
        <property name="org.apache.ws.security.crypto.merlin.keystore.type">
          jks
        </property>
        <property name="org.apache.ws.security.crypto.merlin.file">
          c:/keys/service.ks
        </property>
        <property name="org.apache.ws.security.crypto.merlin.keystore.password">
          service-ks-pass
        </property>
        </crypto>
      </signatureCrypto>
    </RampartConfig>
  </wsp:Policy>
</service>
</serviceGroup>
```

62

seed  
beyond the obvious

## Supporting digital signatures in the web service

---

- Next, create *PasswordCallbackHandler.java* in the *com.ttdev.secure* package to provide the password to decrypt the private key in the "s1" alias:

```
public class PasswordCallbackHandler implements CallbackHandler {
    public void handle(Callback[] callbacks)
        throws IOException, UnsupportedCallbackException {
        for (int i = 0; i < callbacks.length; i++) {
            WSPasswordCallback pwcb = (WSPasswordCallback) callbacks[i];
            String id = pwcb.getIdentifer();
            if (id.equals("s1")) {
                pwcb.setPassword("s1-pass");
            }
        }
    }
}
```

63



## Supporting digital signatures in the web service

---

- Get a certificate for the web service
- Do we need to import c1's (Client's) certificate?
- No. As the client will include it in the message, we don't need it in the keystore.
- On the other hand, do we need to import s1's (Web service's) certificate into the keystore for the client?
- Yes. This is because the web service will not send its certificate to the client, but just the issuer's DN and serial number of the certificate. So the client needs this certificate in its keystore. So, import it:

64





## Supporting digital signatures in the web service

- Now, run the client again. This time it will work. If we check the SOAP response message in TCP Monitor, we'll see:

65



## Supporting digital signatures in the web service

```
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header>
    <wsse:Security
      xmlns:wsse="..."
      soapenv:mustUnderstand="1">
        <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
          Id="Signature-25591289">
          <ds:SignedInfo>
            <ds:CanonicalizationMethod
              Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
            <ds:SignatureMethod
              Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
            <ds:Reference URI="#Id-6923467">
              <ds:Transforms>
                <ds:Transform
                  Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
              </ds:Transforms>
            </ds:Reference>
            <ds:DigestValue>
              UPGGHViqdM6mQrGJ3lFGFwdWBk4=
            </ds:DigestValue>
          </ds:SignedInfo>
          <ds:SignatureValue>
            M680t...
          </ds:SignatureValue>
          <ds:KeyInfo Id="KeyId-17240206">
            <wsse:SecurityTokenReference
              xmlns:wsse="..."
              wsu:Id="STRId-13623369">
              <ds:X509Data>
                <ds:X509IssuerSerial>
                  <ds:X509IssuerName>
                    CN=CA, O=Test CA, ST=Some-State, C=US
                  </ds:X509IssuerName>
                  <ds:X509SerialNumber>5</ds:X509SerialNumber>
                </ds:X509Data>
              </ds:X509Data>
            </wsse:SecurityTokenReference>
          </ds:KeyInfo>
        </ds:Signature>
      </wsse:Security>
    </soapenv:Header>
    <soapenv:Body>
      <ns1:wsu"..."
        wsu:Id="Id-6923467">
        <ns1:concatResponse xmlns:ns1="http://ttdev.com/ss">
          <?xml?>
        </ns1:concatResponse>
      </soapenv:Body>
    </soapenv:Envelope>
```

There is no <BinarySecurityToken> here. It means the s1 certificate is not sent.

Use the issuer DN and certificate serial number (5 here) to identify the certificate. It is up to the client to look it up.

66



## Supporting digital signatures in the web service

---

- That is, it is telling the service that the certificate used to sign the message is issued by *CN=CA,O=Test CA,ST=Some-State,C=US* and the serial number of the certificate is 5.
- It is hoping that the client can use this information to locate the certificate and then use the public key in it to verify the signature.
- For this to work, the client may scan all the certificates in the keystore to try to find it. It means we must import s1's certificate into the keystore on the client.

67



## Supporting digital signatures in the web service

---

- To check that the service is really verifying the signature, note messages like below in the console:

```
[INFO] Deploying module: soapmonitor-1.3
[INFO] script module activated
[INFO] Deploying Web service: BizService
[INFO] Deploying Web service: ImageService
[INFO] Deploying Web service: ManualService
[INFO] Deploying Web service: SecureService
[INFO] Deploying Web service: SimpleService
[INFO] Deploying Web service: version.aar
[INFO] Deploying Web service: WrappedService
[INFO] [SimpleAxisServer] Started
[SimpleAxisServer] Started
[INFO] Listening on port 8080
[INFO] Undeploying Web service: SecureService
[INFO] Deploying Web service: SecureService
[INFO] Undeploying Web service: SecureService
[INFO] Deploying Web service: SecureService
[INFO] Verification successful for URI "#Id-4779445"
[INFO] Verification successful for URI "#Id-4779445"
```

68



## Encrypting SOAP messages

---

- At the moment the messages are signed, but they aren't encrypted and thus people on the Internet can see them.
- If the information is confidential, we should encrypt it.
- To do that, modify the policy in the WSDL file:

69



## Encrypting SOAP messages

---

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions ...>
  <wsp:Policy wsu:Id="p1">
    <sp:AsymmetricBinding>
      ...
    </sp:AsymmetricBinding>
    <sp:Wss10>
      ...
    </sp:Wss10>
    <sp:SignedParts>
      <sp:Body />
    </sp:SignedParts>
    <sp:EncryptedParts>
      <sp:Body /> _____ The <Body> element of the SOAP
    </sp:EncryptedParts> message should be encrypted
  </wsp:Policy>
  ...
</wsdl:definitions>
```

70



## Encrypting SOAP messages

- Generate the service stub and client stub again. Modify rampart-config.xml for the client:

71



## Encrypting SOAP messages

```
<wsp:Policy ...>
  <RampartConfig>
    <user>cl</user>
    <encryptionUser>sl</encryptionUser>
    <passwordCallbackClass>
      com.ttdev.secure.client.PasswordCallbackHandler
    </passwordCallbackClass>
    <signatureCrypto>
      <crypto
        provider="org.apache.ws.security.components.crypto.Merlin">
          <property
            name="org.apache.ws.security.crypto.merlin.keystore.type">
            JKS
          </property>
          <property
            name="org.apache.ws.security.crypto.merlin.file">
            c:/keys/client.ks
          </property>
          <property
            name="org.apache.ws.security.crypto.merlin.keystore.password">
            client-ks-pass
          </property>
        </crypto>
      </signatureCrypto>
    <encryptionCrypto>
      <crypto
        provider="org.apache.ws.security.components.crypto.Merlin">
          <property
            name="org.apache.ws.security.crypto.merlin.keystore.type">
            JKS
          </property>
          <property
            name="org.apache.ws.security.crypto.merlin.file">
            c:/keys/client.ks
          </property>
          <property
            name="org.apache.ws.security.crypto.merlin.keystore.password">
            client-ks-pass
          </property>
        </crypto>
      </encryptionCrypto>
    </RampartConfig>
  </wsp:Policy>
```

This is a keystore alias. Get the certificate for the alias "sl" from the keystore and use the public key there to encrypt the message. Note that you don't need the password to get the public key.

Specify the cryptographic provider to perform encryption. Here, you still use the Merlin provider (JDK). You also specify its configurations (the path to the keystore and the keystore password). Here, everything is the same as the cryptographic provider for signing.

72



## Encrypting SOAP messages

- For the web service, modify services.xml:

73



## Encrypting SOAP messages

```
<serviceGroup>
  <service name="SecureService">
    ..
    <ws:Policy .. wsu:Id="pi">
      <sp:AsymmetricBinding>
        ..
        </sp:AsymmetricBinding>
        <sp:Wss10>
          ..
          </sp:Wss10>
          <sp:SignedParts>
            <sp:Body />
            </sp:SignedParts>
            <sp:EncryptedParts>
              <sp:Body />
              </sp:EncryptedParts>
            < RampartConfig
              xmlns="http://ws.apache.org/rampart/policy">
                <user>
                  <encryptionUser>
                    <encryptionUser>
                      <passwordCallbackClass>
                        com.ttdev.secure.PasswordCallbackHandler
                      </passwordCallbackClass>
                      <signatureCrypto>
                        <crypto>
                          <provider>org.apache.ws.security.components.crypto.Merlin">
                            <property>
                              <name>org.apache.ws.security.crypto.merlin.keystore.type">
                                JKS
                              </property>
                              <name>org.apache.ws.security.crypto.merlin.file">
                                ci/keys/service.ke
                              </property>
                              <name>org.apache.ws.security.crypto.merlin.keystore.password">
                                service-ks-pass
                              </property>
                            </property>
                          </crypto>
                        </signatureCrypto>
                      <encryptionCrypto>
                        <crypto>
                          <provider>org.apache.ws.security.components.crypto.Merlin">
                            <property>
                              <name>org.apache.ws.security.crypto.merlin.keystore.type">
                                JKS
                              </property>
                              <name>org.apache.ws.security.crypto.merlin.file">
                                ci/keys/service.ke
                              </property>
                              <name>org.apache.ws.security.crypto.merlin.keystore.password">
                                service-ks-pass
                              </property>
                            </property>
                          </crypto>
                        </encryptionCrypto>
                      </RampartConfig>
                    </ws:Policy>
                  </serviceGroup>
                
```

The <Body> element of the SOAP message should be encrypted

Encrypt the response using ci's public key

Specify the cryptographic provider to perform encryption. It is the same as the one used for signing. It is also identical to the one used by the client except that it uses a different keystore file.

74



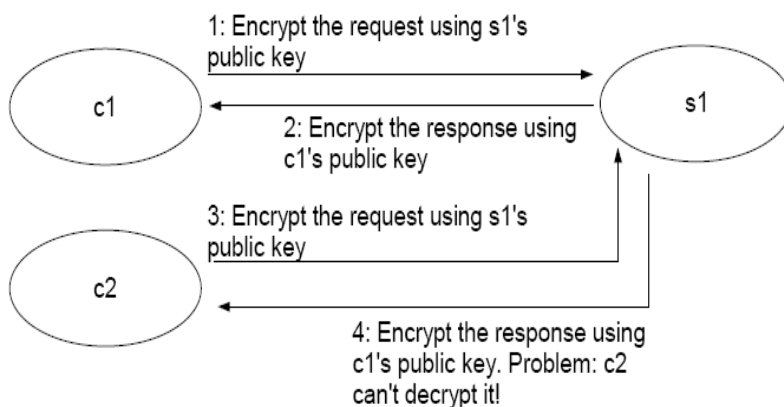
## Encrypting SOAP messages

- However, there is a problem here. As we're encrypting the response message using c1's public key, how can it find out c1's public key?
- we'll need to put c1's certificate in the keystore for the web service. In addition, this web service can only talk to a single client c1.
- If there is another client c2, it can encrypt the request using s1's public key, but s1 will encrypt the response using the public key of c1 (NOT c2), making c2 fail to decrypt it:

75

**seed**  
beyond the obvious

## Encrypting SOAP messages



76

**seed**  
beyond the obvious

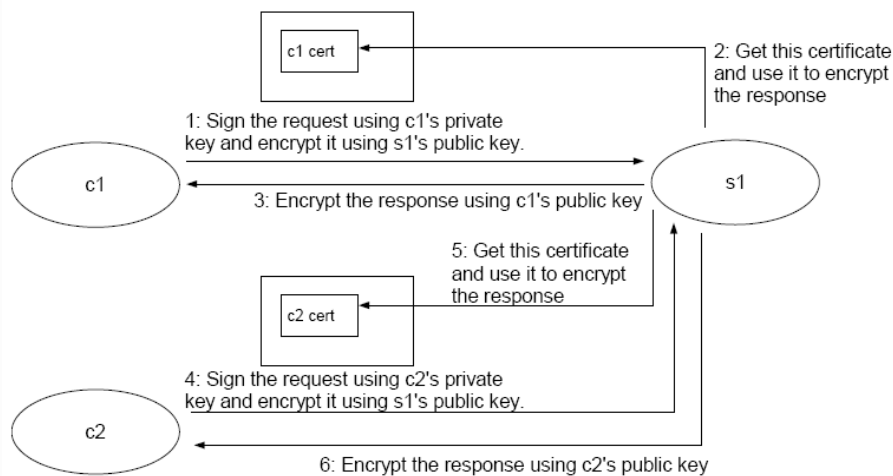
## Encrypting SOAP messages

- To solve this problem, rampart supports a special way of operation.
  - ♦ If c1 both signs and encrypts the request, it will sign it using its own private key.
  - ♦ If it also includes its certificate in the request, then rampart can be instructed to look up this certificate in the request and use it to encrypt the response.
- Therefore, it will use c1's certificate to encrypt the response. If c2 sends it a request, it will encrypt the response using c2's certificate:

77

seed  
beyond the obvious

## Encrypting SOAP messages



78

seed  
beyond the obvious

## Encrypting SOAP messages

---

- To enable this operation, put a special value *"useReqSigCert"* into the `<encryptionUser>` element:

79



## Encrypting SOAP messages

---

```
<serviceGroup>
  <service name="SecureService">
    ...
    <wsp:Policy ... wsu:Id="p1">
      ...
      <RampartConfig
        xmlns="http://ws.apache.org/rampart/policy">
        <user>sl</user>
        <encryptionUser>useReqSigCert</encryptionUser>
        ...
      </RampartConfig>
    </wsp:Policy>
  </service>
</serviceGroup>
```

It stands for "use request signing certificate".  
That is, use the certificate that signed the  
request message.

80





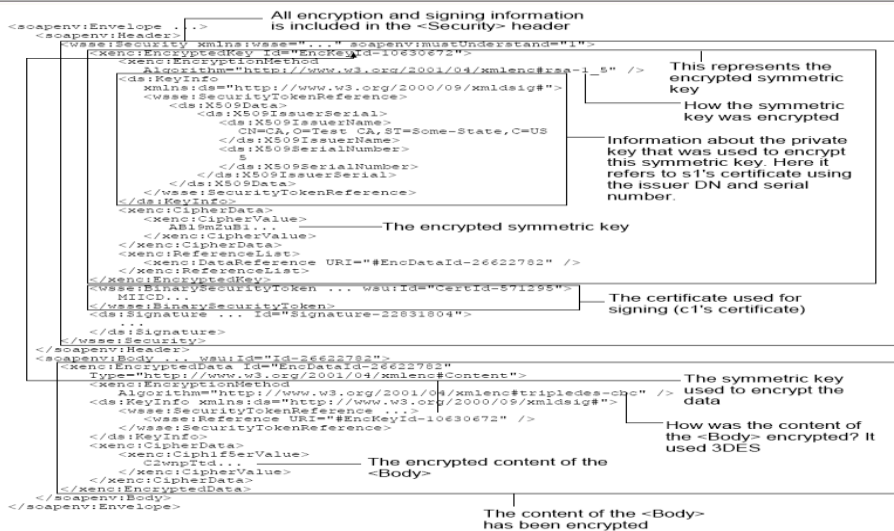
## Encrypting SOAP messages

- Now run the client and it should work. To verify that the messages are indeed encrypted, check them out in the TCP Monitor:

81



## Encrypting SOAP messages



82



## Quick Recap . . .

---

- **WS-Policy** allows us to specify non-functional requirements (QOS) such as security on web services.
- We include a policy in the WSDL file and the generated client stub will use it.
- For the web service, we still need to include it into the services.xml file.

83



## Quick Recap . . .

---

- To sign or encrypt a message, specify in the policy the configuration settings such as algorithms to use, whether to include the certificate (token) and how (direct include or issuer DN plus serial number and etc.). We also specify which parts should be signed and which parts should be encrypted.

84



## Quick Recap . . .

---

- The Rampart module implements the WS-Security standard and can be used to satisfy security requirements expressed in policies.
- It gets information from the policy. In addition, we also need to provide further configurations to it using an XML file or a string. Such configurations include the user name alias, password callback class, what cryptographic provider to use (e.g., JDK), the location of the keystore and the keystore password.