Get an introduction to the principles of public key cryptography, then see how WS-Security applies them for signing and encrypting SOAP messages using public-private key pairs in combination with secret keys. Dennis Sosnoski continues his Java Web services series with a discussion of WS-Security and WS-SecurityPolicy signing and encryption features, along with example code using Axis2 and Rampart.

View more content in this series

Security is crucial when Web services exchange business data. Negative financial or legal consequences can result if data is intercepted by third parties, or if fraudulent data is accepted as valid. It's always possible to design and implement an application's own security handling for Web services — as for any form of data exchange — but that's a risky approach, because even a minor and obscure oversight can lead to serious vulnerabilities. One of the main benefits of SOAP compared to simpler forms of data exchange is that it allows for modular extensions. Security has been a major focus for extensions almost since SOAP’s initial release, resulting in the standardization of WS-Security and related technologies that allow security to be configured for each service as appropriate.

Requirements for security in information exchange generally have three aspects:

- **Confidentiality**: Only the intended recipient of a message has access to the message content.
- **Integrity**: The message is received without modification.
- **Authenticity**: The origin of the message can be verified.

WS-Security lets you easily address all three aspects. In this article, you'll see how to do this using Axis2 and the Rampart WS-Security extension. But first, I'll give you a quick rundown of the principles of public-key cryptography — the basis for most of WS-Security's encrypting and signing features.
Public-key cryptography

Throughout most of history, secure message exchange has been based on some form of shared secret. The secret might take the form of a code, in which the parties to the exchange have an agreed-upon set of substitutions for phrases or actions. Or it could be a cipher, in which text is converted into other text using some algorithm. The secret could even take other forms, such as a language unknown to other parties who might have access to the messages. The shared secret made the message exchange secure. If some other party discovered the secret, though, the message exchange would be compromised, with potentially devastating results for the parties to the exchange. (Think Enigma and the German military communications in World War II, for example.)

Public-key cryptography is an inherently different approach to security that doesn't require a shared secret. It's based on the idea of mathematical "trap-door" functions, which are easy to compute in one direction but very difficult to compute in the reverse direction. For instance, it's easy to find the product of two prime numbers (even very large primes, if you're working on a computer), but it's much more difficult to analyze that product to find the original two factors. If you build an encryption algorithm around a function's easy direction, anyone wanting to break your encryption needs to work at it from the hard side. And with a well-chosen algorithm, the task of breaking the encryption can be made so difficult that it's impractical to do so in a time frame that threatens the message exchange (at least until someone develops a usable quantum computer, or really good psychic abilities).

With public-key cryptography, a party wanting to receive encrypted messages creates a pair of key values. Each key value can be used separately to encrypt messages, but it cannot be used to decrypt the messages it was used to encrypt. Instead, the other key in the pair must be used for the decryption. As long as the keys' owner keeps one of the keys secret, the other key can be made public. Anyone with access to the public key can use it to encrypt messages, which can then be decrypted only by the key owner. Because separate keys are used for encrypting and decrypting messages, this form of cryptography is called asymmetric encryption.

Signing messages

When your public key is used to encrypt a message, only you (as the holder of the private key) can decrypt that message. This ensures confidentiality, one of the three aspects of secure message exchange. But it's also possible to use your private key to encrypt a message, and when this is done anyone with a copy of your public key can decrypt that message. This seems not very useful on the surface — what good is an encrypted message that anyone can read? — but it serves well as a way of verifying message authenticity. Someone who receives an encrypted message that's
purportedly from you can use your public key to decrypt the message and compare it with some expected value. If the result matches, they know that you created the message.

In practice, more is involved in signing messages than just encrypting with your private key. You need some way of establishing the expected value for the decrypted message, for one thing. This is generally done using another variation of mathematical trap-door function: a hash function that is easy to compute but difficult to duplicate (meaning it's hard to make any changes to a message without changing the hash value for that message, or to find another message with the same hash value as a supplied message). With such a hash function, you can generate the hash value (generally called a digest in security discussions) for the message you want to sign, then encrypt that digest using your private key and send the encrypted digest value with the message. Anyone receiving the message can use the same hash algorithm on that message, then decrypt the supplied encrypted digest using your public key and compare the two values. If the values match, the recipient can then be certain (within the limits of current technology, and assuming that you've kept your private key secret) that the message is from you.

One other step is involved in signing messages when you're dealing with XML. XML messages are expressed in text form, but some aspects of the text representation are considered irrelevant by XML (such as the order of attributes on an element or whitespace used within an element start or end tag). Because of this issue with text representations, the W3C (owner of the XML specification) decided to convert XML messages to a canonical text form before computing a digest value (see Related topics). Several canonicalization algorithms are defined that can be used with XML messages. The particular algorithm used generally doesn't matter much, as long as both parties involved in a message exchange agree to use the same one.

Using a signed message digest guarantees both message integrity (because modifications to the message will change the digest value) and authenticity (because your private key is used to encrypt the digest). Because confidentiality is ensured on messages sent to you by encryption that uses your public key, all the major aspects of message exchange security are covered by using a public-private key pair. Of course, with a single key pair only one direction of a message exchange is secured — but if the other party to the exchange also has its own public/private key pair, full security can be provided for messages going in each direction.

Certificates

So public-private key pairs can be used for both encryption and signing of messages exchanged between two parties, provided each party has access to the other's public key. That leaves the issue of how you get the public key in a way that maintains security. Among the various ways of doing this, the most widely used is to have one or more trusted third parties vouch for the public key. Digital certificates are the mechanism for providing public keys in this vouched-for form.

A digital certificate is basically a wrapper around a public key, which includes identifying information for the party owning that key. This wrapped body is then signed by a trusted third party, and the signature is included in the certificate. The trusted third party vouches for the public key and identifying information by issuing the certificate with its signature. Of course, this leaves the bootstrap issue of how you establish the trusted third party's identity. This is generally done by
hardcoding certificates for certain trusted third parties called issuing authorities into software (such as the JVM).

There's a lot more to digital certificates beyond the basics described here, including ways of revoking certificates issued by mistake (which, unfortunately, happens) or with compromised private keys, time spans in which the certificate is valid, and extensions to specify the intended use of a certificate. See Related topics for links to more information on digital certificates and public-key encryption in general. You can also check the documentation for the keytool security tool included in your JDK installation. The keytool documentation gives a good introduction to certificate structure and handling, along with keystores (discussed next) and other aspects of security. You'll also see an example later in this article of working with the keytool.

Keystores

Most Java security code works with private keys and digital certificates using keystores. A keystore is just a file that contains key and certificate entries in encrypted form. A password is required for access to the keystore. Each private key in the keystore is again encrypted, requiring an additional password, in order to help maintain the keys' security. Any software that uses the keystore and private key needs to have both keystore and private-key passwords available at run time. This limits the security provided by these passwords (because anyone with access to the source code can find out how the passwords are loaded). So you need to maintain physical security of the system hosting the software and any backups to that system. And you should keep the keystore and password only on that system and those backups in order to keep your private keys secure.

Secret keys and symmetric encryption

Although public-key cryptography using asymmetric encryption is the basis for many of the useful features of WS-Security, old-fashioned secret-key cryptography still plays an important role. Asymmetric-encryption algorithms tend to be much more computationally intensive than symmetric algorithms based on secret keys (whereby the same key is used for both encrypting and decrypting, meaning the key must always be kept secret) for equivalent levels of protection. For this reason, the two techniques are often used in combination: high-cost asymmetric encryption is used to safeguard the exchange of a secret key, which can then be used for low-cost symmetric encryption. You'll see an example of this approach later in the article, when you look at WS-Security encryption of a message.

Setting up

This article uses the same example application as "Axis2 WS-Security basics," which shows how to implement the WS-Security UsernameToken using Axis2 and Rampart. A few changes are needed to support using WS-Security public-key cryptography features, though, so a separate code package accompanies this article (see Download).

The root directory of the example code is jws05code. Inside this directory you'll find:

- An Ant build.xml file
- An Ant build.properties file, which configures the operation of the example application
- The library.wsdl file giving the service definition for the example application
• A log4j.properties file used to configure client-side logging
• Several property-definition XML files (all named XXX-policy-client.xml or XXX-policy-server.xml)

Before you can use the example code, you need to:

1. Edit the build.properties file to set the path to your Axis2 installation.
2. Make sure your Axis2 installation has been updated with the Rampart code, as discussed in the Installing Rampart section of "Axis2 WS-Security basics." (A good way to check is to look in the repository/modules directory for a rampart-x.y.mar module file.)
3. Add the org.bouncycastle.jce.provider.BouncyCastleProvider Bouncy Castle security provider (necessary for the public-key cryptography features used in the example code) to your JVM security configuration (the lib/security/java.security file) (see Related topics).
4. Add the Bouncy Castle JAR (named bcprov-jdkxx-yyy.jar, where xx is your Java version and yyy is the Bouncy Castle code version) to both your Axis2 installation's lib directory and your Axis2 server application's WEB-INF/lib directory.

Now you're ready to build the example application and try out the security examples covered in the next sections.

Signing messages
Signing requires a lot more specification than the UsernameToken example in "Axis2 WS-Security basics." You need to:

• Identify the private/public key pair that's used to create the signature in each direction, and supply the passwords used to access the keystore and the private key.
• Specify the set of algorithms used for XML canonicalization, digest generation, and actual signing.
• Specify which portions of the message are to be included in the signature.

Part of this information is handled as configuration data, embedded in the WS-SecurityPolicy document for the service. Other parts are included in the run-time message exchange.

Listing 1 shows a WS-Policy document used for configuring the Axis2 client to sign messages. (Listing 1 is edited to fit the page width. See the full text as sign-policy-client.xml in the example code.)

Listing 1. WS-Policy/WS-SecurityPolicy for signing (client)

```xml
<wsp:Policy wsu:Id="SignOnly"
  xmlns:wsu="http://.../oasis-200401-wss-wssecurity-utility-1.0.xsd"
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:AsymmetricBinding
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
        <wsp:Policy>
          <sp:InitiatorToken>
            <wsp:Policy>
              <sp:X509Token
```
In Listing 1, the `<sp:AsymmetricBinding>` element in the policy gives the basic configuration information for using public-key cryptography in the message exchange. Within this element, several nested elements are used for the specifics of the configuration. The `<sp:InitiatorToken>` identifies the key pair used for signing messages from the client (the initiator) to the server (the recipient), in this case stating that the public key will be in the form of an X.509 certificate and will be sent with each message from the client (sp:IncludeToken="http://.../AlwaysToRecipient"). The `<sp:RecipientToken>` identifies the key pair used for signing messages on the response path from the server to the client, again using an X.509 certificate and with the certificate included in each message from the server (sp:IncludeToken="http://.../AlwaysToInitiator").
The `<sp:AlgorithmSuite>` element identifies the set of algorithms used for the signing. The `<sp:IncludeTimestamp>` says that a timestamp will be used with each message (useful for preventing replay-type attacks on a service, whereby a message is captured in transit and then resubmitted in an attempt to confuse or break the service). The `<sp:OnlySignEntireHeadersAndBody>` element says that signing will be done on the entire header or body of the message, rather than on some nested element (another security measure, preventing certain types of attacks that rewrite a message). The `<sp:SignedParts>` element identifies the message parts to be signed — in this case, the SOAP message Body.

The last part of the Listing 1 WS-Policy document gives Rampart-specific configuration information. This is a more-complex version of the Rampart configuration used in "Axis2 WS-Security basics," this time including a `<ramp:user>` element to identify the key that will be used for signing messages, and a `<ramp:signatureCrypto>` element to configure the keystore that contains the client private key and server certificate. The referenced keystore file must be present in the classpath at run time. In the example application, the keystore file is copied to the client/bin directory during the build.

The password-callback classes are a little different from those used in "Axis2 WS-Security basics." For that article, the password callback was only needed on the server and only needed to verify (for a plaintext UsernameToken) or set (for a hash UsernameToken) the password that matched a particular user name. For the public-key cryptography used in this article, the callback must supply the password used to protect a private key within the keystore. And separate callbacks are needed for the client (to supply the password for the client private key) and server (to supply the password for the server private key). Listing 2 shows the client-side version of the callback:

### Listing 2. Client password callback

```java
/**
 * Simple password callback handler. This just checks if the password for the private key
 * is being requested, and if so sets that value.
 */
public class PWCBHandler implements CallbackHandler {
    public void handle(Callback[] callbacks) throws IOException {
        for (int i = 0; i < callbacks.length; i++) {
            WSPasswordCallback pwcb = (WSPasswordCallback)callbacks[i];
            String id = pwcb.getIdentifer();
            int usage = pwcb.getUsage();
            if (usage == WSPasswordCallback.DECRYPT || usage == WSPasswordCallback.SIGNATURE) {
                // used to retrieve password for private key
                if ("clientkey".equals(id)) {
                    pwcb.setPassword("clientpass");
                }
            }
        }
    }
}
```

The callback in Listing 2 is designed to support both signing and decrypting using the same private-public key pair, so it checks for both cases and returns the same password in either case.
Along with the **Listing 1** WS-Policy used for the client, there's a matching one for the server (sign-policy-server.xml in the example code), which differs only in the details of the Rampart configuration. Likewise, the server version of the **Listing 2** password-callback class differs only in the identifier value and password returned.

**Running the example application**

The build.properties file has lines referencing the `client.policy` and `server.policy` files to be used by the example application. These are set respectively to `sign-policy-client.xml` and `sign-policy-server.xml` in the supplied version of the file, so you should only need to build the application. You can do this with Ant by opening a console to the `jws05code` directory and entering `ant`. If everything is configured correctly, you should end up with a `library-signencr.aar` Axis2 service archive in the `jws05code` directory. Deploy the service to your Axis2 server installation by uploading the `.aar` file using the Axis2 Administration page, and then try out the client by entering `ant run` at the console. If everything is set up correctly, you should see the output shown in Figure 1:

**Figure 1. Console output for the running application**

To see the actual WS-Security information in the messages, you need to use a tool such as TCPMon (see [Related topics](##)). First get TCPMon set up and accepting connections from the client on one port, which it then forwards to the server running on a different port (or a different host). You can then edit the build.properties file and change the host-port value to the listening port for TCPMon. If you again enter `ant run` at the console, you should then see the messages being exchanged. Listing 3 shows a sample client-message capture:

**Listing 3. First message from client to server**

```xml
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header>
    <wsse:Security xmlns:wsse=".../oasis-200401-wss-wssecurity-secext-1.0.xsd"
      soapenv:mustUnderstand="1">
      <wsu:Timestamp xmlns:wsu=".../oasis-200401-wss-wssecurity-utility-1.0.xsd"
        wsu:Id="Timestamp-3753023">
      </wsu:Timestamp>
      <wsse:BinarySecurityToken
        xmlns:wsu=".../oasis-200401-wss-wssecurity-utility-1.0.xsd"
        EncodingType=".../oasis-200401-wss-soap-message-security-1.0#Base64Binary"
        ValueType=".../oasis-200401-wss-x509-token-profile-1.0#X509v1"
        wsu:Id="CertId-2650016">MIICoDC...0n33w==</wsse:BinarySecurityToken>
    </wsse:Security>
  </soapenv:Header>
  <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig">
    <!-- Signature content -->
  </ds:Signature>
</soapenv:Envelope>
```
The `<wsse:Security>` header within the SOAP message holds all the run-time security configuration information and signature data. The first item present is a `<wsu:Timestamp>`, as requested by the WS-SecurityPolicy configuration. The timestamp includes two time values: the time at which it was created and the time at which it expires. These two values are five minutes apart in this case, which is the default for Rampart. (You can change the values as part of the Rampart policy configuration.) The amount of time between the two is somewhat arbitrary, but five minutes is a reasonable value — it’s long enough to cope with a reasonable amount of clock skew (differences in the system clock times) between the client and the server, while still being short enough to limit the potential for replay attacks using the message. After the timestamp, the next item in the security header is a `<wsse:BinarySecurityToken>`. This security token is the client certificate, in base64 binary-encoded form.

The third item in the security header is a `<ds:Signature>` block, with three child elements. The first child element, `<ds:SignedInfo>`, is the only part of the message that is signed directly. The first child elements within `<ds:SignedInfo>` identify the algorithms used for its own canonicalization and signing. These are followed by a `<ds:Reference>` child element for each message component included in the signing. Each child `<ds:Reference>` element references a particular message component by identifier and gives the canonicalization and digest algorithms.
applied to that component, along with the resulting digest value. The remaining child elements of
<ds:SignedInfo> give the actual signature value and a reference to the public key used to verify
the signature (in this case the certificate included in the <wsse:BinarySecurityToken> earlier in the
header, as identified by wsu:Id="CertId-2650016").

Encrypting and signing messages

Adding encryption to the signed message exchange is easy, requiring just the addition of an
<sp:EncryptedParts> element to the policy to identify the component(s) to be signed, and some
additional Rampart configuration information. Listing 4 shows the client version of a policy for this
purpose (again edited to fit the page width — see the signencr-policy-client.xml file in the example
code for the full text), with the additions to the Listing 1 policy in bold:

Listing 4. WS-Policy/WS-SecurityPolicy for signing and then encrypting (client)

```xml
<!-- Client policy for first signing and then encrypting all messages, with the
certificate included in the message from client to server but only a thumbprint
on messages from the server to the client. -->
<wsp:Policy wsu:Id="SignEncr"
 xmlns:wsu="http://.../oasis-200401-wss-wssecurity-utility-1.0.xsd"
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:AsymmetricBinding
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
        <wsp:Policy>
          <sp:InitiatorToken>
            <wsp:Policy>
              <sp:X509Token
                sp:IncludeToken="http://.../IncludeToken/AlwaysToRecipient="/>
            </wsp:Policy>
          </sp:InitiatorToken>
          <sp:RecipientToken>
            <wsp:Policy>
              <sp:X509Token
                sp:IncludeToken="http://.../IncludeToken/Never"/>
            </wsp:Policy>
          </sp:RecipientToken>
          <sp:AlgorithmSuite>
            <wsp:Policy>
              <sp:TripleDesRsa15/>
            </wsp:Policy>
          </sp:AlgorithmSuite>
          <sp:Layout>
            <wsp:Policy>
              <sp:Strict/>
            </wsp:Policy>
          </sp:Layout>
          <sp:IncludeTimestamp/>
          <sp:OnlySignEntireHeadersAndBody/>
        </wsp:Policy>
      </sp:AsymmetricBinding>
      <sp:SignedParts xmlns:sp="http://.../200702">
        <sp:Body/>
      </sp:SignedParts>
      <sp:EncryptedParts xmlns:sp="http://.../200702">
```
<sp:Body/>
</sp:EncryptedParts>

<ramp:RampartConfig xmlns:ramp="http://ws.apache.org/rampart/policy">
<ramp:user>clientkey</ramp:user>
<ramp:encryptionUser>serverkey</ramp:encryptionUser>
<ramp:passwordCallbackClass>com.sosnoski.ws.library.adb.PWCBHandler</ramp:passwordCallbackClass>

<ramp:signatureCrypto>
<ramp:crypto provider="org.apache.ws.security.components.crypto.Merlin">
<ramp:property name="org.apache.ws.security.crypto.merlin.keystore.type">JKS</ramp:property>
<ramp:property name="org.apache.ws.security.crypto.merlin.file">client.keystore</ramp:property>
<ramp:property name="org.apache.ws.security.crypto.merlin.keystore.password">nosecret</ramp:property>
</ramp:crypto>
</ramp:signatureCrypto>

<ramp:encryptionCrypto>
<ramp:crypto provider="org.apache.ws.security.components.crypto.Merlin">
<ramp:property name="org.apache.ws.security.crypto.merlin.keystore.type">JKS</ramp:property>
<ramp:property name="org.apache.ws.security.crypto.merlin.file">client.keystore</ramp:property>
<ramp:property name="org.apache.ws.security.crypto.merlin.keystore.password">nosecret</ramp:property>
</ramp:crypto>
</ramp:encryptionCrypto>

</ramp:RampartConfig>
</wsp:All>
</wsp:ExactlyOne>
</wsp:Policy>

Why not just encryption?

It would be nice to supply an example of using only encryption with Axis2 and Rampart, but this functionality is broken in Axis2 1.4 (and earlier). A code fix is in place for the 1.5 release. If you're using Axis2 1.5 or later (along with a corresponding Rampart release), try using the encr-policy-client.xml and encr-policy-server.xml policy files to encrypt the body of each message without using any signatures.

The first change in the Listing 4 policy is not required in order to use encryption, but it's a good idea. When using encryption, the client needs to have the server certificate available when sending the initial request (because the server public key from the certificate is used for the encryption). Because the client needs to have the server certificate anyway, there's never a reason to send the server certificate to the client. The changed <sp:RecipientToken> in the Listing 4 policy reflects this usage, saying that a certificate should not be sent (sp:IncludeToken=".../Never") and that a thumbprint reference (basically a hash of the certificate) should be used instead. The thumbprint reference is much more compact than a full certificate, so using the reference reduces the message size and processing overhead.

The change that actually implements encryption is the added <sp:EncryptedParts> element. This element says that encryption is to be used, and the content <sp:Body> element says that the SOAP message body is the part of the message to be encrypted.
The added Rampart configuration information in Listing 4 consists of a `<ramp:encryptionUser>` element giving the alias for the public key (that is, certificate) to be used for encrypting the message, and a `<ramp:encryptionCrypto>` element telling how to access the keystore that contains the certificate. In the example application, the same keystore is used for both the private key used for signing and the public key used for encrypting, so the `<ramp:encryptionCrypto>` element is just a renamed duplicate of the existing `<ramp:signatureCrypto>` element.

At run time, Rampart needs to get the password used to protect the private key to be used to decrypt the encrypted data. The password callback earlier used to get the private-key password for signing, shown in Listing 2, also provides the password for decryption, so no change is needed there.

**Running the example application**

To try the example application using signing followed by encryption, you first need to edit the build.properties file. Change the client policy line to `client.policy=signencr-policy-client.xml` and the server policy to `server-policy=signencr-policy-server.xml`. You can then rebuild the application by running `ant`, deploy the generated library-signencr.aar file to your Axis2 installation, and run `ant run` to give it a try.

Listing 5 shows a request-message capture when signing followed by encryption is used, with the significant differences from the Listing 3 signing-only version in bold:

**Listing 5. Message using signing and encryption**

```xml
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
  <soapenv:Header>
    <wsse:Security xmlns:wsse=".../oasis-200401-wss-wssecurity-secext-1.0.xsd" soapenv:mustUnderstand="1">
      <wsu:Timestamp xmlns:wsu=".../oasis-200401-wss-wssecurity-utility-1.0.xsd" wsu:Id="Timestamp-4067003">
        <wsu:Created>2009-04-21T23:15:47.701Z</wsu:Created>
      </wsu:Timestamp>
      <xenc:EncryptedKey Id="EncKeyId-urn:uuid:6E12E251E439C034FA12403557497352">
        <xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
        <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
          <wsse:SecurityTokenReference>
            <wsse:KeyIdentifier EncodingType="http://.../oasis-200401-wss-wssecurity-utility-1.0.xsd"ValueType="http://.../oasis-wss-soap-message-security-1.1#ThumbprintSHA1">uYn3PK2wXheN2lLZr4n2mJjoWE0=</wsse:KeyIdentifier>
          </wsse:SecurityTokenReference>
        </ds:KeyInfo>
      </xenc:EncryptedKey>
      <wsse:BinarySecurityToken xmlns:wsu="http://.../oasis-200401-wss-wssecurity-utility-1.0.xsd" EncodingType="http://.../oasis-200401-wss-wssecurity-utility-1.0#Base64Binary" ValueType="http://.../oasis-200401-wss-x509-token-profile-1.0#X509v1">MIICo...</ds:Signature>
    </wsse:Security>
  </soapenv:Header>
  <soapenv:Body>
    ...
  </soapenv:Body>
</soapenv:Envelope>
```
The first difference is the presence of a `<xenc:EncryptedKey>` element in the security header. This element's content gives a secret key in encrypted form, using the server's public key to do the encryption. The second difference is the actual SOAP `<Body>` content, which has been replaced by a `<xenc:EncryptedData>` element. This encrypted data element references the `<xenc:EncryptedKey>` value from the security header as the key for the symmetric encryption used on the `<Body>` content.

**Using your own self-signed certificates**

To get an official digital certificate signed by a recognized certificate authority, you need to generate a public-private key pair and use the public key to create a certificate request. You then send that certificate request to your preferred authority and pay it. The authority in turn verifies
your identity (an important step for the integrity of the whole process, though one that sometimes suffers from embarrassing lapses), and issues the certificate with its signature.

For testing or internal use, you can instead generate your own self-signed certificates. The example code for this article uses two such self-signed certificates, one for the client and one for the server. The client.keystore keystore used on the client side contains the client's private key and certificate, along with the server certificate (which must be stored on the client so that it'll be accepted as valid without the endorsement of a certificate authority when used for signing, and also so that it can be used directly for encryption). The server.keystore keystore used on the server side contains the server's private key and certificate, along with the client certificate (again necessary so that the certificate will be accepted as valid).

You can generate your own private keys and self-signed certificates and substitute your generated key-certificate pairs for those provided in the download. To do this using the keytool program included in the JDK, you open a console and first enter this command line (split here to fit the page width; you must enter it as a single line):

```
keytool -genkey -alias serverkey -keypass serverpass -keyalg RSA -sigalg SHA1withRSA -keystore server.keystore -storepass nosecret
```

The preceding command generates the server key and certificate with alias serverkey in a new keystore named server.keystore. (If you have an existing server.keystore in this directory, you first need to delete the existing key pair using that alias.) The keytool prompts for a number of information items used for the certificate generation (none of which really matter for use in testing) and then asks you to approve the information. Once you've done so by typing yes, keytool creates the keystore with the private key and certificate and then exits.

Next, run through the same procedure to create the client key pair and keystore, this time using this command line (entered as a single line):

```
keytool -genkey -alias clientkey -keypass clientpass -keyalg RSA -sigalg SHA1withRSA -keystore client.keystore -storepass nosecret
```

The next step is to export the certificate from the server keystore, and import that certificate into the client keystore. To export, use this command line (split here to fit the page width; you must enter it as a single line):

```
keytool -export -alias serverkey -keystore server.keystore -storepass nosecret -file servercert.cer
```

The export creates a certificate file named servercert.cer, which you can then import into the client keystore using this command line (entered as a single line):

```
keytool -import -alias serverkey -keystore client.keystore -storepass nosecret -file servercert.cer
```
When you run the import command, keytool prints out the details of the certificate and asks if you trust the certificate. Once you accept the key by typing yes, it adds the certificate to the keystore and exits.

For the final step, export the client certificate and import that into the server keystore, by first running (entered as a single line):

```
keytool -export -alias clientkey -keystore client.keystore -storepass nosecret
  -file clientcert.cer
```

Then run (split here to fit the page width; you must enter it as a single line):

```
keytool -import -alias clientkey -keystore server.keystore -storepass nosecret
  -file clientcert.cer
```

### Why export/import both certificates?

The text asks you to export a certificate for each party, and then import that certificate into the keystore for the other party. You would need to do this for the server certificate if you're using encryption, even if the certificate were signed by an accepted authority, because encryption requires access to the other party's public key. For the client certificate, on the other hand, the import into the server keystore is necessary only because the certificate is self-signed and cannot otherwise be validated. By importing the certificate into the keystore, you're approving it in advance, so that validation through an authority is not needed.

You can use this same approach to work with multiple clients using self-signed certificates, simply importing each client's certificate into the server keystore. As another alternative, rather than working with self-signed certificates, you can instead run your own certificate authority (with a tool such as OpenSSL) and require each client to get a certificate signed by that authority. That way, you can add the certificate authority to your server keystore, and any client presenting a certificate signed by that authority will be accepted. Or you can just pay to use official certificates signed by an accepted authority.

To make use of the new keys and certificates, you must copy the client.keystore file into the client/src directory of the example code before running the client build (or just copy it into the client/bin directory to take immediate effect) and the server.keystore file into the server/src directory of the example code before running the server build.

The sample keytool command lines in this section use the same file names and passwords as in the supplied example code. You can also change these values when you generate your own keys and certificates, but you then also need to change the example code to match. The keystore password and file name are parameters in the RampartConfig section of each party's policy file. The client key password is hardcoded in the client version of the `com.sosnoski.ws.library.adb.PWCBHandler` class, and the server key password in the server version of the same class.

### Wrapping up

In this article, you've seen how to use Axis2 and Rampart for policy-based WS-Security encryption and signatures. These powerful security features are essential for many types of business data exchange, but they do come with a cost in terms of added processing overhead. In the next Java
Web services article, you’ll see how the relative performance costs of different types of security stack up so that you can better judge how to use security in your own applications.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code for this article</td>
<td>j-jws5.zip</td>
<td>36KB</td>
</tr>
</tbody>
</table>
Related topics

- **Apache Axis2/Java**: Visit the project home for the Axis2 Web services engine.
- **Rampart**: Learn more about the Rampart WS-Security module for Axis2.
- "**Java Web Services: Axis2 Data Binding**" (Dennis Sosnoski, developerWorks, July 2007): This article discusses the main data-binding options for Axis2. Its sample code is the basis of the sample code for this column.
- "**Public-key cryptography**" and "**Digital signature**" (Wikipedia): Wikipedia provides heavily cross-linked introductions to these security topics, including a variety of useful links letting you explore in depth.
- "**Introducing XML canonical form**" (Uche Ogbuji, developerWorks, December 2004): Read an introduction to XML canonicalization, an important step in using digital signatures for XML data.
- **Understanding Web Services specifications**: This series of tutorials introduces many of the important Web services standards, including:
  - "**Understanding Web Services specifications: Web Services Description Language (WSDL)**" (Nicholas Chase, developerWorks, July 2006)
- **OASIS Web Services Security (WSS) TC**: This is the organization responsible for the WS-Security specification and token profiles. You can find links here to all versions of these standards.
- **The W3C Web Services Policy Working Group**: This group defines the WS-Policy specification.
- **OASIS Web Services Secure Exchange (WS-SX) TC**: This organization is responsible for WS-SecurityPolicy and related specifications.
- **Apache Axis2**: Download the latest Axis2 release.
- **Rampart**: Download the Rampart module for Axis2.
- **The Legion of the Bouncy Castle**: Download the correct Bouncy Castle .jar for your runtime.
- **TCPMon**: Download this open source utility for monitoring TCP connections.