IBM Research Report

Sign and Encrypt Any Element in a SOAP Message

Hyen-Vui (Henry) Chung

IBM Software Group 11501 Burnet Road Austin, TX 78758-3400

Michael McIntosh, Paula Austel

IBM Research Division Thomas J. Watson Research Center P.O. Box 704 Yorktown Heights, NY 10598

Masayoshi Teraguchi

IBM Research 1623-14 Shimotsuruma Yamato 242 Japan



Research Division Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home.

Sign and Encrypt Any Element in a SOAP Message

Hyen-Vui (Henry) Chung Senior Software Engineer, WebSphere Architecture and Development Austin, TX

Michael McIntosh Senior Software Engineer, IBM Research Hawthorne, NY

Paula Austel Senior Software Engineer, IBM Research Hawthorne, NY

Masayoshi Teraguchi Researcher, IBM Research Yamato, Japan

May, 2006

This article describes how to use IBM® WebSphere® Application Server Version 6.0 (hereafter called Application Server) and IBM Rational® Application Developer (hereafter called Application Developer) to sign and encrypt any element in a SOAP message using the Web Services Security 1.0 standard (WS-Security). WS-Security is designed to be flexible and extensible. However, that flexibility and extensibility is a double-edged sword: it enables security for many message-level scenarios, but adds significant complexity to the development process. WebSphere Application Server provides a simple keyword-based mechanism to specify which SOAP message elements are to be signed and encrypted. Keywords are defined to support the majority of common usage scenarios for standard message elements. However, SOAP messages frequently contain non-standard application-defined elements that must also be protected. This article describes how you can use an XPATH expression with WebSphere Application Server to sign and encrypt any element in a SOAP message. The article is intended for Web services application developers who need to secure their SOAP messages using message-level security.

You should have a good understanding of Java[™] programming, XML, XPATH, Web services and general cryptographic security technology.

Introduction	2
WS-Security High-Level Architecture in WebSphere	3
WS-Security Deployment Model	4
Predefined keywords	5
XPATH	7
Overview of the sample application	7
Run the sample application	9
Generate the keys	9
Results	10
Configure WS-Security constraints for the request generator	11
Configure	12
integrity (digital signature)	12
Configure Confidentiality (Encryption)	18
Configure WS-Security constraints for the request consumer	21
Configure required integrity	23
Configure required confidentiality	28
SOAP message with WS-Security	31
Conclusion.	33

Introduction

Web services is an emerging technology for designing and developing loosely coupled, distributed systems. Web Services Security 1.0 (WS-Security) is an OASIS standard for message level security using XML encryption, XML signature, and security tokens.

WebSphere Application Server has provided support for WS-Security since release 5.0.2. WS-Security support in Application Server V5.0.2 and V5.1.x was based on a prestandard draft of the specification. The WS-Security 1.0 specification became an OASIS standard in April 2004, and Application Server V6 supports this version of the standard. Similarly, Rational Application Developer V5.0.x and V5.1.x support the draft version of the WS-Security specification, and Rational Application Developer V6 supports both the draft version and the 1.0 standard.

The OASIS Web Services Security Technical Committee designed the WS-Security standard to be flexible and extensible. The intent was to provide a standard that developers could use to secure as many Web services scenarios as possible. However, flexibility and extensibility comes with a price: complexity. The challenge for vendors who provide implementations of the WS-Security standard lies in striking a balance between ease of use and flexibility. The WS-Security implementation in WebSphere Application Server concentrates on ease of use for the most common usage scenarios, while enabling more complex usage patterns for more expert users.

This article focuses on how to use Rational Application Developer and WebSphere Application Server to secure SOAP messages using XML Digital Signature and XML Encryption. There are two methods in Rational Application Developer for selecting a SOAP element to be signed and encrypted. One uses a predefined keyword and the other uses an XPATH expression.

Each of these methods address different requirements. The predefined keyword method provides an easy way of selecting SOAP elements, like body, securitytoken, bodycontent, and so on. WebSphere provides a list of keywords that cover the most frequently signed and encrypted SOAP elements. This method makes it easy to select SOAP elements, but the trade-off is less flexibility, since the selections are limited to the predefined keywords.

The XPATH method provides the flexibility to select virtually any SOAP element in a message. XPATH is a powerful language for XML navigation, but the syntax is cryptic and complex.

This article covers both the predefined keyword and XPATH methods for selecting elements for signing and encryption.

WS-Security High-Level Architecture in WebSphere

Before we dive into the details of how to use digital signature and encryption support of WS-Security in Rational Application Developer, it would be beneficial to describe the high-level architecture.

Some vendors provide WS-Security support through APIs, but IBM WS-Security support is based on a deployment model. The WS-Security requirements are expressed as security constraints in deployment descriptors, which are separated from the application business logic. The deployment descriptors are XML files that describe security constraints. The application server runtime reads the deployment descriptors and enforces the security constraints. This programming model is similar to the J2EE model. Both the deployment descriptor and API –based programming models have their merits and limitations. This article focuses only on the deployment model and does not discuss in any detail the differences between these two approaches.

WS-Security processing is declared as security constraints in deployment descriptors using development tools, such as Rational Application Developer. These security constraints are separate from the application business logic. There is no industry standard format for the WS-Security deployment descriptor, but the security constraints are similar to those defined in the WS-SecurityPolicy language. During SOAP message exchange, the WS-Security runtime, which is implemented as a global handler in the Web service engine, intercepts the SOAP message in the outbound message and applies WS-Security mechanisms based on the security constraints in the deployment descriptor ((a) and (c) in Figure 1). Similarly, on the inbound message, the WS-Security runtime validates the security constraints based on the deployment descriptor. For example, if the WS-Security in the SOAP message satisfies the requirement in the deployment descriptor, then the SOAP message is dispatched to the target Web Service, otherwise, the SOAP message is rejected with a SOAP fault ((b) in Figure 1).



Figure 1. High-level architecture overview of WS-Security deployment model

WS-Security Deployment Model

The Application Server deployment model contains two configuration files: the deployment descriptor and the binding. The deployment descriptor is used to configure the high-level and platform-independent security constraints similar to WS-SecurityPolicy. The binding is used to configure platform-specific security constraints for the specific deployed instance. This section summarizes what you can configure in the deployment descriptor. We'll explain the details of the binding configuration in a later section.

You can configure the following security constraints with Application Developer, as shown in Figure 1:

For the client outbound (request generator: (a) in Figure 1):

- Integrity (SOAP elements to be signed)
- Confidentiality (SOAP elements to be encrypted)
- Security token (generated for authentication or identity assertion)
- Timestamp

For the Web service inbound (request consumer: (b) in Figure 1):

- Required integrity (SOAP elements should be signed)
- Required confidentiality (SOAP elements should be encrypted)
- Required security token (used for authentication or identity assertion)
- Required timestamp

For the Web service outbound (response generator: (c) in Figure 1):

• Integrity (SOAP elements to be signed)

- Confidentiality (SOAP elements to be encrypted)
- Timestamp

For the client inbound (response consumer: (d) in Figure 1):

- Required integrity (SOAP elements should be signed)
- Required confidentiality (SOAP elements should be encrypted)
- Required timestamp

As described earlier, for integrity and confidentiality, Application Server provides two types of selection of the SOAP elements to be signed or encrypted: predefined keyword based selection and XPATH selection.

Predefined keywords

Application Server provides some useful predefined keywords for selecting the common SOAP elements to be signed or encrypted. The predefined keywords for digital signature are:

- **relatesto**: Select the <wsa:RelatesTo> element defined in the Web Services Addressing (WS-Addressing) specification.
- **messageid**: Select the <wsa:MessageID> element defined in the WS-Addressing specification.
- to: Select the <wsa:To> element defined in the WS-Addressing specification.
- action: Select the <wsa:Action> element defined in the WS-Addressing specification.
- **securitytoken**: Select all security tokens used for authentication.
- **enckey**: Select all <ds:KeyInfo> elements used in the <enc:EncyptedKey> elements or the <enc:EncryptedData> elements.
- **dsigkey**: Select all <ds:KeyInfo> elements used in the <ds:Signature> elements.
- timestamp: Select the <wsu:Timestamp> element that is the last child of the <wsse:Security> element.
- **body**: Select the SOAP body element.



Figure 2. Keywords for digital signature

The predefined keywords for encryption include the following:

- (1) **usernametoken**: Select the <wsse:UsernameToken> element under the <wsse:Security> element.
- (2) **digestvalue**: Select all <ds:DigestValue> elements in the <ds:Signature> element.
- (3) bodycontent: Select all child nodes of the SOAP body element.



Figure 3. Keywords for encryption

Application Server provides a list of predefined keywords for most commonly used SOAP elements in a SOAP message. The predefined keywords provide a fast and easy way for specifying the SOAP elements for digital signature and encryption. This may be all you need in most scenarios. Later in this article, we'll provide a sample of how to use keywords to sign and encrypt the SOAP body and SOAP body content, respectively.

ХРАТН

Predefined keywords may not satisfy all scenarios; for example, signing and encrypting custom SOAP headers or signing or encrypting only parts of headers or the SOAP body. For that reason, WebSphere also provides another mechanism for selecting SOAP elements for digital signature and encryption. The alternative mechanism is to use XPATH language to select SOAP elements.

The XPATH language is a W3C standard for addressing parts of an XML document. It is flexible and powerful, but the language syntax is complex.

Application Server also provides XPATH support to select the parts of SOAP message for digital signature and encryption. The XPATH support¹ for selecting part of the SOAP message doesn't support full XPATH specification, but only node-set selection. For example, the following XPATH expression selects the SOAP body element. Of course, you can select the same SOAP body element with the predefined body keyword for digital signature feature described in the previous section. As you can see, the XPATH expression is much more complex than the predefined keyword. The sample in this article describes how to select the content of a custom SOAP header for encryption.

```
/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/' and
local-name()='Envelope']/*[namespace-
uri()='http://schemas.xmlsoap.org/soap/envelope/' and local-
name()='Body']
```

Overview of the sample application

This article provides a sample to demonstrate the following:

- Using the predefined keyword body for signing and signature validation of the SOAP body and using the predefined keyword bodycontent for encrypting and decrypting the SOAP body content.
- Using XPATH expression to select a custom SOAP header and the SOAP header content for signature and encryption, respectively.

The sample application is a simple "Hello, World" Web service that sends a SOAP message with a custom header (TestHeader). The Web service client is implemented as a Web client and the Web service implementation is a Servlet Java Bean. In this sample, we'd like to:

¹ Don't confuse this with the XPATH reference for digital signature.

- Sign the SOAP body using the predefined keyword body and
- Sign the TestHeader using the XPATH expression:

```
/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/' and
local-name()='Envelope']/*[namespace-
uri()='http://schemas.xmlsoap.org/soap/envelope/' and local-
name()='Header']/*[namespace-uri()='http://com.ibm.hvc.example1' and
local-name()='TestHeader']
```

- Encrypt the SOAP body content using predefined keyword bodycontent and
- Encrypt the TestHeader content using the XPATH expression:

```
/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/' and
local-name()='Envelope']/*[namespace-
uri()='http://schemas.xmlsoap.org/soap/envelope/' and local-
name()='Header']/*[namespace-uri()='http://com.ibm.hvc.example1' and
local-name()='TestHeader']/node()
```

The following is a sample SOAP message without security:



Figure 4 . Sample unsecured SOAP message

Following is an overview of the steps. More details are provided in the subsequent sections:

- 1. Generate the keys for digital signature and encryption. The sample uses different sets of keys for digital signature and encryption.
- 2. Configure the WS-Security constraints for the request generator (client outbound)
- 3. Configure the WS-Security constraints for the request consumer (Web service inbound)

You can use a Web service you have already developed, or download the sample application provided with this article.

Run the sample application

Download the sample application and run it, as described below. The sample application has been tested on WebSphere Application Server 6.0.2.

- 1. Copy the key stores to the \${USER_INSTALL_ROOT}/etc/ws-security/hvc directory, where \${USER_INSTALL_ROOT} is the profile directory.
- 2. Deploy the sample application EAR file (SignEncryptAnyElement.ear).
- 3. Start the sample application.
- 4. Bring up a Web browser. The client URL is http://localhost:9080/SayHelloClient/.
- 5. Click OK to send a Web service request. You can change the port to redirect the request to a network monitor to capture the SOAP message; for example, TCPMON.

Generate the keys

The sample uses a self-signed certificate for digital signature. You can easily extend this to use a CA-issued certificate for digital signature. There are a few tools available to generate a self-signed certificate. In this case, we're using the keytool provided by the Java Development Toolkit for generating the keys for digital signature and encryption.

 Generate an RSA key pair for digital signature. The alias of the key is john, and DN is CN=John Smith, OU=Development, O=ACME, L=OneCity, ST=OneState, C=US. The password is johnsmith in the sender key store sendersigner.jks, and the store password is signer.

```
keytool -genkey -alias john -keyalg RSA -validity 365 -keystore
sendersigner.jks -storepass signer -dname "CN=John Smith,
OU=Development, O=ACME, L=OneCity, ST=OneState, C=US" -keypass
johnsmith
```

- Export the public certificate of alias john and import it into the receiver trust store. This is used by the receiver to validate the trust of the public certificate. If you use a CA-issued certificate, you need to import the CA public certificate into the trust store instead.
 - a. Export john public certificate to a file john.cert:

```
keytool -export -alias john -keystore sendersigner.jks -storepass
signer -file john.cert
```

b. Import john public certificate in file john.cert into the receiver trust store (receivertruststore.jks). It is very important that you make sure the certificate is authentic. The certificates in the trust store are trusted by the receiver.

keytool -import -alias john -keystore receivertruststore.jks - storepass truststore -file john.cert

3. Generate an RSA key pair for encryption. The alias of the key is dev2, and the DN is CN=dev2, OU=Development, O=ACME, L=OneCity, ST=OneState, C=US. The password is dev2receiver in the receiver key store receiver.jks, and the store password is receiver.

keytool -genkey -alias dev2 -keyalg RSA -validity 365 -keystore receiver.jks -storepass receiver -dname "CN=dev2, OU=Development, O=ACME, L=OneCity, ST=OneState, C=US" -keypass dev2receiver

- 4. Export the public key of alias dev2, which is in the public certificate and import it into the sender key store. The sender uses the public key of dev2 to encrypt the message.
 - a. Export dev2 public certificate to a file dev2.cert:

keytool -export -alias dev2 -keystore receiver.jks -storepass receiver -file dev2.cert

b. Import dev2 public certificate in file dev2.cert into the sender key store (sender.jks):

keytool -import -alias dev2 -keystore sender.jks -storepass sender -file dev2.cert

Note that the expiration of the sample keys in the sample application is ten years, but in real life applications, you may want to have a shorter expiration for security reasons. You can also generate new keys using the script file provided with this article.

Results

The following key stores are created after this step:

Key store	Content	Purpose
sendersigner.jks	John Smith public	Client uses John Smith
	(certificate) and private	private key to sign the SOAP
	keys	message
Sender.jks	Dev2 public (certificate)	Client uses Dev2 public key
	key	to encrypt the SOAP message
receiver.jks	Dev2 public (certificate)	Web service uses Dev2
	and private keys	private key to decrypt the
		message
receivertruststore.jks	John Smith public	Web service uses the trust
	(certificate) key	store to verify trust of the
		signer certificate

Configure WS-Security constraints for the request generator

This section describes how to configure the WS-Security constraints and bindings for the request generator (client outbound request), which defines what WS-Security constraints apply to the client outbound SOAP message. This section assumes you have already developed your application using Rational Application Developer⁴.

The WS-Security editors for constraints and bindings are tabs in the Web Deployment Descriptor (for Web-based client).

1. Open the Web Deployment Descriptor in the Project Explorer of the J2EE perspective by double-clicking the **web.xml** file of the Web application (the example Web service client is a Web-based client), as shown in Figure 5:



Figure 5 . Open the Web Deployment Descriptor of the Web-based Web service client

2. Open the WS-Security constraints editor by clicking the WS Extension tab and the bindings editor by clicking the WS Binding tab, as shown in Figure 6:

⁴ You can also use Application Server Toolkit to configure the WS-Security constraints and bindings.

😵 Web Deployment Descriptor ×	•
General Information	* Usage
Display name: SayHelloClient	The following Enterprise Applications use this web module:
Description:	Refresh
Session time out:	-
Distributable	
* Serviets and JSPs	۲ 📃
The following servlets and JSPs are used in this application:	Web Library Projects
SavHaluServlet Details	The following JARs are mapped to Java projects:
	Add Remove
K = >	
• Filters	
The following filters are used in this application:	
Overview Serviets Filter Security References WS Handler Pages Var	rables WS Extension WS Binding Extensions Source

Figure 6 . Open the WS Extension and WS Binding editors

Configure integrity (digital signature)

The XML digital signature standard is a complex specification, which is reflected in the number of steps required to enable digital signature. First, you need to define what to sign (integrity security constraints) and then define the binding information, such as the token generator to send the public key of the signer as a X509 binary security token, the key locator to locate the signer key, the key information for the security token reference, and signature information (signature algorithm, digest method and transform).

To configure integrity, complete the following steps:

- 1) Define the integrity constraints (what to sign) in the **WS Extension** tab.
 - In the WS Extension editor, select **Request Generator configuration** -> **Integrity**, then click **Add**.
 - Specify int for Integrity Name.
 - Specify 1 for **Order** (sign first, then encrypt).
 - In the **Message Parts** field, specify two parts to be signed in this integrity constraint: the SOAP body using the predefined keyword body, and the TestHeader using the following XPATH expression:

```
/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
and local-name()='Envelope']/*[namespace-
uri()='http://schemas.xmlsoap.org/soap/envelope/' and local-
name()='Header']/*[namespace-uri()='http://com.ibm.hvc.example1'
and local-name()='TestHeader']
```



Figure 7. Configure integrity

- 2) Next, define the binding information for integrity (digital signature). Open the bindings editor in the **WS Binding** tab by clicking **Security Request Generator Binding Configuration**.
 - Select Token Generator and click Add to create an X509 Token Generator (x509) to send the public certificate of the signer CN=John Smith, OU=Development, O=ACME, L=OneCity, ST=OneState, C=US.
 - Leave the **Security token** field blank, because the token is not a standalone token for authentication, but is used for digital signature.

loken generator name:	x509			
oken generator class:	com.ibm.wsspi.wssecu	rity.token.X509Toke	nGenerator	-
ecurity taken				*
Use value type				
Value type	XS09 certificate token			*
Local name:	http://docs.oasis-ope	n.org/vss/2004/01/a	asis-200401-wss-x509-token	-profile-1.04XS09
URI:				
Call back handler:	com.ibm.wsspi.wssecu	rity.auth.calback.XS	09CalbackHandler	
iser ID:				
assword:				
Use key store				
Key store storepass:	signer			
Key store path:	\$(USER_INSTALL_RO	OT)/etc/ws-security/	hvc/sendersigner.jks	
Key store type:	3/5			*
Keyi				
Akes	Key pa	85	Keyname	
Alasi john Add Remov	Key pa johnan	es ith	Key name ON=John Smith, O	U=Developme
Alas john Add Exmov	Key pa johran	nn Idh	Key name Oti-Jahn Smith, O	U-Develapme
Alas john Add Remov Call back handler prop	Kity pa johnan sthy:	os ith Talue	Key name Oti-Juhn Smith, O	U=Developme
Add Exercise add Exercise Call back handler prop	Kay pa johnan artiy:	es ch Value	Key name Oti-Jahn Smith, O	U-Developme
Add Exercise John Call back handler prop	Kay pa johnan artiy:	es rth Value	Key name Oti-Jahn Smith, O	U-Developme
Add Execord Add Execord Call back handler prop Name Add Execord	Kity pa johnan artiy:	es rth Value	Key name Oti-Jahn Smith, O	U-Developme
Add Exmov Add Exmov Call back handler prop Name Add Exmov	Key pa johran	es eth Value	Key name Oti-John Smith, O	U=Developme
Add Ennov Add Ennov Call back handler prop Name Add Ennov Property: Name	Key pa Johnen erty:	os	Key name Oti-Jahn Smith, O	U=Developme
Add Ennov add Ennov Call back handler prop Name Add Ennov Property: Name	kiry pa johnan mhys	es	Key name Oti-Jahn Smith, O	U=Developme
Add Exercov add Exercov Call back handler prop Name Add Exercov Property: Name	s johnen software antys	es oft	Key name Oti-Jahn Smith, O	U=Developme
Add Execov Add Execov Call back handler prop Name Add Execov Property: Name Add Execov	Key pa	es ch Value Value	Key name Oti-Juhn Smith, O	U-Developme
Add Exercov Call back handler prop Name Add Exercov Property: Name Add Exercov T Use certificate path	Kity pa johnan athy:	es vth Value Value	Key name Oti-Jahn Smith, O	U=Developme
Add Exercov Call back handler prop Name Add Exercov Property: Name Add Exercov	Key pa Johnan	es	Key name Oti-Jahn Smith, O	U=Developme

Figure 8. Generate token

3) Select **Key Locators** and click **Add** to create a key locator (or signer) to locate the private key of the signer CN=John Smith, OU=Development, O=ACME, L=OneCity, ST=OneState, C=US.

Say Locator dials	ug		×
Key locator name:	bigner		
Key locator dass:	com. Ibm. wsspi, wssecurit	y keyinfo KeySt	oreKeyLocator 💌
🔽 Use key store			
Key store storepass:	signer		
Key store path:	\$(USER_INSTALL_ROOT)/etc/ws-securit	y/hvc/sendersigner.jks
Key store type:	305		
Key:			
Alas	Key pass		Keyname
john	johnsmith		CN=John Snith, OU=Devel
Add Remove			
Name		Value	
Add Remove			OK Cancel

Figure 9. Create key locator signer

4) Select Key Information and click Add to create security token reference (STR) key information (strref). Note that the Token field is the token generator name x509 created in step 2; the Key locator field is the name of the key locator created in step 3 which is signer, and the Key name field is the signer CN=John Smith, OU=Development, O=ACME, L=OneCity, ST=OneState, C=US.

Key Information	i dialog 🛛 🔀
Key information name:	strref
Key information type:	STRRE®
Key information class:	com.bm.ws.webservices.wssecurity.keyinfo.STRReferenceContentGeneration
✓ Use key locator	
Key locator:	signer 💌
Key name:	CN=John Smith, OU=Development, O=ACME, L=OneOty, ST=OneState 💌
🔽 Use token	
Token:	x509
Property:	
Name	Value
	1
Add Remov	<u>c</u>
	OK Cancel

Figure 10. Create STR key information

5) Select **Signing Information** and click **Add** to define the signing information (int). Note that the **Key information element** is the name of the key information created in step 4, which is strref.

🥹 Signing Information dialog	3	×
Signing information mana:	lint	_
Canonicalization method algorithm	http://www.w3.org/2001/30/mil-exc-c14n#	•
Show only FIPS Complete Alon	riture	
Signature method algorithms http	(//www.w3.org/2000,09/wikdsig#rsa-sha1	•
Key information name: strict		_
Key information elements streef		•
Use key information signature		
Тури:	kayinfo	Ŧ
Key information signature property	:	_
Name	Value	-
Jacob Land		
Add Remove		
Property:		_
Nane	Value	
Add Renove		
	DK Caro	

Figure 11. Define signing information

6) Next, link the int signing information created in step 5 with the int security constraints defined in step 1. Make sure the int signing information created in step 5 is selected, then select **Part References** and click **Add** to add a part reference (int). Note that the **Integrity part** field in the **Part Reference** dialog is int, the integrity name you created in step 1. For this example, use the default digest method algorithm.

🙁 Part Reference	dialog	×
Part reference name:	int	-
Integrity part:	int _	•
G Show only FIPS C Digest method algorit	omplient Algorithms mm: http://www.w3.org/2000/09/xmidsig#sha1	
	OK Cancel	

Figure 12. Add part reference

7) Finally, define the transform algorithm for the int part reference created in step 6. Make sure the int part reference is selected, then select **Transforms** and click **Add** to create the transform algorithm (int). For this example, use the default transform.

Transform dialog		×
Name: Int		_
Algorithm: http://www.w3	3.org/2001/10/xml-exc-c19n#	•
Transform property:		
Name	Yalue	
-		
Add Remove	OK Cancel	

Figure 13. Define transform algorithm

Configure Confidentiality (Encryption)

To configure confidentiality, or encryption, you need to first define what to encrypt (confidentiality security constraints), and then define the binding information, such as the key locator to locate the public key to encrypt, key identifier key information, and encryption information (that is, data and key encryption algorithms).

To configure confidentiality, complete the following steps:

- Define the confidentiality constraints (what to encrypt) in the WS Extension tab. In the WS Extension editor, select Request Generator configuration -> Confidentiality, then click Add.
 - For **Confidentiality Name**, specify conf.
 - For **Order**, specify 2 (sign then encrypt).
 - In the **Message Parts** field, specify two parts to be encrypted in this confidentiality constraint: the SOAP body content using the predefined keyword bodycontent, and the TestHeader content using the following XPATH expression:

```
/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
and local-name()='Envelope']/*[namespace-
uri()='http://schemas.xmlsoap.org/soap/envelope/' and local-
name()='Header']/*[namespace-uri()='http://com.ibm.hvc.example1'
and local-name()='TestHeader']/node()
```

"hadycontent" keyward reflers to	😂 Confidentiality Dialog 🛛 🕅 🔀
body content:	Confidentiality/Nome binf
<g239:seyhellorequest< p=""></g239:seyhellorequest<>	Order 2
xmins:p219="http://www.hrc.com/H elloWorld/">World:/p239:3ayHello	Message Rants
Request>	Message parts dialect Message parts keyword
	http://www.w3.org/TR/1999/REC-spath-199 /*/namespace-un()=http://schenas.um/sca
"TELTH" expression refers to	
TestHeader content:	Add Ranove
chre:el)-1234/hre:el>	lines
	Nance keyword
	1 Second
	Timestanp
	Timestamp dalect Timestamp keyvord Timestamp exp
	Add Renove
	OK. Canal

Figure 14. Configure integrity

2. Select **Key Locators** and click **Add** to create the key locator (conf) to locate the public key of the receiver CN=Dev2, OU=Development, O=ACME, L=OneCity, ST=OneState, C=US. The public key is used to encrypt the shared key automatically generated by the WS-Security runtime.

🙁 Key Locator dia	log				6	×
Key locator name:	conf					-
Key locator class:	com.ibm	wsspi.wssecurit	y.keyinfo.Keyi	StoreKeyLocator		·
🔽 Use key store						
Key store storepass:	sender					
Key store path:	\$(USER	INSTALL_ROOT	}/etc/ws-secu	rity/hvc/sender.;	jks	
Key store type:	JKS					•
Keyi						
Alias		Key pass		Key name]
dev2		dev2sender		CN=dev2,	OU=Developme	
Add Remove	-					
Name			Value]
Add Remove				ОК	Cancel	

Figure 15. Create key locator conf

3. Select **Key Information** and then click **Add** to create KEYID (Key Identifier) key information (keyid). Note that no **Token** field is required, the **Key locator** field is the name of the key locator created in step 2 (conf), and the **Key name** field is the receiver CN=Dev2, OU=Development, O=ACME, L=OneCity, ST=OneState, C=US.

🙁 Key Informatio	n dialog	×
Key information name:	seyid	
Key information type:	KEYID	•
Key information class:	com.bm.ws.webservices.wssecurity.keyinfo.KeyIdContentGenerator	_
Use key locator		
Key locator:	conf	•
Key name:	CN=dev2, OU=Development, O=ACME, L=OneOty, ST=OneState, C=L	•
Use token		
Tolom:		Ŧ
Property:		
Name	Value	
and a	1	
Add		
	OK Cancel	

Figure 16. Create key information keyid

4. Finally, you need to create the encryption information. Select Encryption Information, then click Add to create encryption information (conf). The Data encryption method algorithm is AES (128 bits), the Key encryption method algorithm is RSA, the Key information name is keyid, which you defined in Step 3, and the Confidentiality part is conf, defined in step 1.

Encryption Inform Encryption name: [conf	ation dialog			×
Show only FIPS Comp Data encryption method Key encryption method a	ilient Algorithm algorithm: http lgorithm: http	s p://www.w3.a p://www.w3.a	rg/2001/04/xmlenc#aes: rg/2001/04/xmlenc#rsa-	128-cbc •
Key information name: Key information element: Confidentiality part: Property:	keyid keyid conf			•
Name		Value		
Add Remove			ОК	Cancel

Figure 17. Create encryption information

Configure WS-Security constraints for the request consumer

This section describes how to configure the WS-Security constraints and bindings requirements for the request consumer (Web service inbound request). The WS-Security of the inbound SOAP message must meet the security constraints defined, otherwise the SOAP message is rejected with a SOAP fault. For example, if the security constraints define that the SOAP body must be signed and the inbound SOAP message body is not signed, the request is rejected and a SOAP fault is returned. This section assumes you have already developed your application and are using Rational Application Developer⁵.

The WS-Security editors for constraints and bindings are tabs in the Web services editor.

1. Open the Web Service editor in the Project Explorer of the J2EE perspective by double-clicking the **webservices.xml** file of the Web application. The example Web service is implemented as Servlet Java Bean:

⁵ You can also use Application Server Toolkit for configuring the WS-Security constraints and bindings.



Figure 18. Open the Web service editor for the Web service

2. Open the WS-Security constraints editor from the **Extensions** tab and the bindings editor from **Bindings Configuration** tab, as shown in the following:

🛎 Web Services Editor 🗶	
Web Services	
Editor for Web service deployment descriptor (nebservices.um)	
* Web service overview	Web service description overview
Details of the Web service deployment descriptor (webservices ami)	Details of the selected Web service description
Description:	Description:
Display name:	Display name:
T loons	* Icons
The following icons represent this Web service deployment descriptor	The following icons represent the selected Web service description
Small icon: Browse	Snal icon: Broke
Largeiton: Brouse	Large icon: Trovse
* Web service descriptions	 Web service description implementation details
The following are the deployed Web services in this module	Implementation details of the selected Web service description
Since and	Web service description name: HelisWorld
	NSOL file: INEE-BVF./hsdl/Heikmonfd.wsd Dowse:
Web Services Port Components Handlers Extensions Andines Sinding Configuration	3

Figure 19. Open the Extensions and Bindings editors

Configure required integrity

The number of steps for configuring required integrity is similar to the steps to configure integrity in the request generator configuration. First, you define what is required to be signed and then define the binding information, such as trust anchors, token consumer to validate the signer's public key as X509 binary security token, key locator to locate the public key for signature validation, key information for security token reference, and signature information (signature algorithm, digest method and transform requirement).

To configure required integrity, complete the following steps:

- Define the required integrity constraints (what needs to be signed) in the Extensions tab. In the Extensions editor, select Request Consumer Service Configuration Details → Required Integrity, then click Add.
 - Specify int for the **Required Integrity Name**.
 - In the **Message Parts** field, specify two parts required to be signed: the SOAP body using the predefined keyword body, and the TestHeader using the following XPATH expression:

```
/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
and local-name()='Envelope']/*[namespace-
uri()='http://schemas.xmlsoap.org/soap/envelope/' and local-
name()='Header']/*[namespace-uri()='http://com.ibm.hvc.example1'
and local-name()='TestHeader']
```



Figure 20. Configure required integrity

2. To define the binding information for required integrity, open the Bindings editor in the **Bindings Configuration** tab by selecting **Request Consumer Binding**

Configuration Details.

3. Select **Trust Anchor** and click **Add** to create the trust anchor configuration the name trust and the trust store \${USER_INSTALL_ROOT}/etc/ws-security/hvc/receivertruststore.jks.

Trust anchor name:	trust
Key store storepass	truststore
Key store path:	\${USER_INSTALL_ROOT};/etc/ws-security/hvc/receivertr
Key store type:	35
	and a second of

Figure 21. Configure trust anchor

- Select Token Consumer and click Add to create an X509 Token Consumer (x509) to validate the public certificate of the signer CN=John Smith, OU=Development, O=ACME, L=OneCity, ST=OneState, C=US.
 - Leave the **Security token** field blank, because this is not a standalone token for authentication, but is used for digital signature.
 - Specify trust in the **Trust anchor reference** field.
 - In the **jaas.config name** field, specify system.wssecurity.X509BST, which is one of the predefined JAAS login configuration names.

Johan Sonaumar Girling				×
Token consumer name:	x509			
Token consumer class:	com.bm.wsspi.wssecurity.1	oken. 11509TokenConsumer	-	
Security taken				
🔽 Use value type				
Value type	X509 certificate token			
Local name:	http://docs.oasis-open.org	/wss/2004/01/oasis-200401-wss-x509-token-profile	-1.0#X509	
URI:				
🗟 Use jaas.config				
javas.config.names	system.wasecurity.350985	r		
jaas.config property:				
Name		Value		
Add Remove				
Use trusted ID evaluator				
Trasted ID evaluator class:				
Trusted ID evaluator property:				
Name		Value		
Add Remove				
Use trusted ID evaluator refer	ence			
Trusted ID evaluator reference:				
Property:				
Name		Value		
Add Remove				
Vie certificate path settings				
G Certificate path reference				
Trust anchor reference:	unt			
Certificate store reference:				
C Trust any certificate				
		OK	Cancel	

Figure 22. Create token consumer

5. Select **Key Locators**, then click **Add** to create a key locator (int) for locating the signer public certificate CN=John Smith, OU=Development, O=ACME, L=OneCity, ST=OneState, C=US in the SOAP message. Note that the **Key locator class** field is com.ibm.wsspi.wssecurity.keyinfo.X509TokenKeyLocator, which locates the key with the X509 certificate embedded in the received incoming SOAP message.

🙁 Xey Locator dia	slog			×
Key locator name:	int			
Key locator dass:	com.lbm.wsspi.wssecu	rity.keyinfo.XS	09TokenKeyLocator	•
Use key store				
Key store storepass:				
Key store path:				
Key store type:				-
Key:				
Alas	Key pass		Key name	
Add Remov	in .			
Name		Value		
Add Remov	19			
			ок	Cancel

Figure 23. Create key locators

6. Select **Key Information**, then click **Add** to create security token reference (STR) key information (strref). Note that the **Token** field is the token generator name x509 created in step 4, the **Key locator** field is the name of the key locator created in step 5 (int), and the **Key name** field is not required.

Key Information	n dialog		6	×
Key information name:	stref			
Key information type:	STRREP			ł
Key information class:	com.ibm.vis.webservices.	vssecurity ke	pirrfo.STRReferenceContentConsum	×
🔽 Use key locator				
Key locator:	int			,
Key name:				·
Vise token				
Token:	x509			,
Property:				
Name		Value		1
Add Revov	0			
			OK Cencel	J

Figure 24. Create key information

7. Select **Signing Information**, then click **Add** to define the signing information (int). Note that the **Key information element** field is strref, the key information created in step 6.

Staning Information dialog	X
Signing information name: Int	
Canonicalization method algorithm: http://ww	ww.w3.org/2001/30/xml-exc-c14n#
Show only FIPS Compliant Algorithms	
Signature method algorithm: http://www.wi	3.org/2000/09/xmldsig#rsa-sha1
Signing key information:	
Key information name:	Key information element:
strref	strref
1	
Add Remove	
Use key information signature	
Type:	yinfo 💌
Key information signature property:	
Name	Value
Add Remove	
Property:	
Name	Value
Add Remove	
	OK Cancel

Figure 25. Define signing information

8. Link the int signing information created in step 7 with the int security constraints defined in step 1. Make sure the int signing information created in step 7 is selected, then select **Part References**, and click **Add** to add a part reference (int). Note that that the **RequiredIntegrity part** field int, which is the Required Integrity name created in step 1. In this example, use the default digest method algorithm.

Part reference name:	int	_
lequiredintegrity parts	Int	
Digest method algorith	mpion: Agonemo m: [http://www.w3.org/2000/09/unidsig#sha1	¥

Figure 26. Add part reference

9. Finally, define the transform algorithm for the int part reference created in step 8. Make sure the int part reference is selected, then select **Transforms** and click **Add**

to create a transform algorithm. Use the default transform algorithm.

C Trans	form dialog		×
Name:	int		_
Algorithm:	http://www.w3.or	g/2001/30/xmi-exc-c14n#	٠
Transform	property:		
Name		Value	
-			
Add	Remove	OK Cancel	

Figure 27. Define transform algorithm

Configure required confidentiality

To configure required confidentiality, you must first define which part is required to be encrypted, then define the binding information, such as the key locator to locate private key to decrypt, key identifier key information, and encryption information (like data and key encryption algorithms).

To configure required confidentiality, complete the following steps:

- Define the Required Confidentiality constraints (which part is required to be encrypted) in the Extensions tab. In the Extensions editor, select Request Consumer Service Configuration Details -> Required Confidentiality, then click Add.
 - Specify conf for the Required Confidential Name.
 - In the **Message Parts** field, specify two parts that are required to be encrypted: the SOAP body content using the predefined keyword bodycontent and the TestHeader content using the following XPATH expression:

```
/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
and local-name()='Envelope']/*[namespace-
uri()='http://schemas.xmlsoap.org/soap/envelope/' and local-
name()='Header']/*[namespace-uri()='http://com.ibm.hvc.example1'
and local-name()='TestHeader']/node()
```



Figure 28. Configure required confidentiality

2. Select Key Locators, then click Add to create a key locator (conf) to locate the private key of the receiver CN=Dev2, OU=Development, O=ACME, L=OneCity, ST=OneState, C=US to decrypt the shared key that is used to decrypt the encrypted data.

🔾 Xay Locator diab	28				×
Key locator name:	conf				
Key locator class:	com.bn.	vsspi, vssecurity	Jaryinfa Keyi	StoreKeyLocator	٠
R Los key store					
Key store storepess:	receiver				
Key store paths	\$(LSER.)	INSTALL_ROOT)	/etc/ive-eecu	rity/hvc/receiver.jks	
Key store type:	.¥S				
Keyi					
Alas		Key pass		Key name	
dev2		dev greceiver		CN=dev2, OU=Develops	ne
Add Reneve	1				
	_				
Phaperty:					
Name			Yalue		
Add Remove					
				OK Can	oel

Figure 29. Create key locator

3. Select **Key Information**, then click **Add** to create KEYID (Key Identifier) key information (keyid). Note that no **Token** field is required, the **Key locator** field is the name of the key locator created in step 2 (conf), and the **Key name** field is the receiver CN=Dev2, OU=Development, O=ACME, L=OneCity, ST=OneState, C=US.

😂 Xey Informatio	n diblog	×
Key information name:	jeyid	
Key information type:	KEYID	•
Key information class:	com. bm. ws. webservices. wssecurity. keyinfo KeyidCo	intentConsumer
🔽 Use key locator		
Key locator:	canf	•
Key name:	ON=dev2, OU=Development, O=ACME, L=OneOty,	ST=OneState, C=L 💌
🗌 Use token		
Token:		v
Property:		
Name	Value	
and the second second		
Add Remos	VE.	
	OK	Cancel

Figure 30. Define key information

4. Finally, create the encryption information. Select Encryption Information, then click Add to create the encryption information (conf). The Data encryption method algorithm is AES (128 bits), the Key encryption method algorithm is RSA, the Key information Name is keyid, defined in step 3, and the Required Confidentiality Part is conf, defined in step 1.

tree person monitorities and	ewp
ncryption name: [conf	
Show only FIPS Compliant Algori	thms
ata encryption method algorithm:	http://www.w3.org/2001/04/xmlenc#aes128-cbc
ey encryption method algorithm:	http://www.w3.org/2001/04/xmlenc#rsa-1_5
and the law information	
Key information name:	Key information element:
æyid	karyid
Add Remove	
equiredConfidentiality part: conf	
operty:	United
THEFT IS	Taka
Add Remme	
	CW Canva

Figure 31. Create encryption information

5.

SOAP message with WS-Security

You can specify a port other than 9080 to redirect the request to a network traffic monitor to capture a SOAP message secured with the WS-Security constraints.

In the captured SOAP message below, you can see that the SOAP body and TestHeader are signed and the SOAP body content and the TestHeader content are encrypted.

```
<soapenv:Envelope .....">
<soapenv:Header>
  <wsse:Security xmlns:wsse="http://docs.oasis-</pre>
open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
soapenv:mustUnderstand="1">
   <wsse:BinarySecurityToken xmlns:wsu="http://docs.oasis-</pre>
open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
soap-message-security-1.0#Base64Binary" ValueType="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509"
wsu:Id="x509bst_2">MIIC.....joZQ==</wsse:BinarySecurityToken>
   <EncryptedKey xmlns="http://www.w3.org/2001/04/xmlenc#">
    <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-</pre>
1 5"/>
    <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
     <wsse:SecurityTokenReference>
      <wsse:KeyIdentifier ValueType="http://docs.oasis-</pre>
open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-
1.0#X509SubjectKeyIdentifier">59FYdEaWcm2ey3HNZyY8Rq0oE5w=</wsse:KeyIde
ntifier>
     </wsse:SecurityTokenReference>
    </ds:KeyInfo>
    <CipherData>
     <CipherValue>LfLr.....yK4Q=</CipherValue>
    </CipherData>
    <ReferenceList>
     <DataReference URI="#wssecurity encryption id 3"/>
     <DataReference URI="#wssecurity_encryption_id_4"/>
    </ReferenceList>
   </EncryptedKey>
   <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    <ds:SignedInfo>
     <ds:CanonicalizationMethod
Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
      <ec:InclusiveNamespaces xmlns:ec="http://www.w3.org/2001/10/xml-
exc-c14n#" PrefixList="wsse ds xsi soapenc xsd soapenv "/>
     </ds:CanonicalizationMethod>
     <ds:SignatureMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
     <ds:Reference URI="#wssecurity_signature_id_0">
      <ds:Transforms>
       <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-</pre>
c14n#">
```

```
<ec:InclusiveNamespaces
xmlns:ec="http://www.w3.org/2001/10/xml-exc-c14n#" PrefixList="xsi
soapenc xsd p239 wsu soapenv "/>
       </ds:Transform>
      </ds:Transforms>
      <ds:DigestMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <ds:DigestValue>Jmo/XVjmpblCmHGsAlAlhRb2tb4=</ds:DigestValue>
     </ds:Reference>
     <ds:Reference URI="#wssecurity_signature_id_1">
      <ds:Transforms>
       <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-</pre>
c14n#">
        <ec:InclusiveNamespaces
xmlns:ec="http://www.w3.org/2001/10/xml-exc-c14n#" PrefixList="hvc xsi
soapenc xsd wsu soapenv "/>
       </ds:Transform>
      </ds:Transforms>
      <ds:DigestMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#shal"/>
      <ds:DigestValue>bPotI7bz9q3eTPwwN8pGFcK8yBI=</ds:DigestValue>
     </ds:Reference>
    </ds:SignedInfo>
    <ds:SignatureValue>pyiq.....Y980=</ds:SignatureValue>
    <ds:KeyInfo>
     <wsse:SecurityTokenReference>
      <wsse:Reference URI="#x509bst_2" ValueType="http://docs.oasis-</pre>
open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509"/>
     </wsse:SecurityTokenReference>
    </ds:KeyInfo>
   </ds:Signature>
  </wsse:Security>
  <hvc:TestHeader xmlns:hvc="http://com.ibm.hvc.example1"</pre>
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
wssecurity-utility-1.0.xsd" wsu:Id="wssecurity_signature_id_1">
   <EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"</pre>
Id="wssecurity_encryption_id_4"
Type="http://www.w3.org/2001/04/xmlenc#Content">
    <EncryptionMethod
Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc"/>
    <CipherData>
     <CipherValue>W9Q+.....9Kpw==</CipherValue>
    </CipherData>
   </EncryptedData>
  </hvc:TestHeader>
 </soapenv:Header>
 <soapenv:Body xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-</pre>
200401-wss-wssecurity-utility-1.0.xsd"
wsu:Id="wssecurity_signature_id_0">
  <EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"</pre>
Id="wssecurity_encryption_id_3"
Type="http://www.w3.org/2001/04/xmlenc#Content">
   <EncryptionMethod
Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc"/>
   <CipherData>
    <CipherValue>7qH0.....i9hA==</CipherValue>
```

```
</CipherData>
</EncryptedData>
</soapenv:Body>
</soapenv:Envelope>
```

Conclusion

WebSphere Application Server supports the WS-Security 1.0 standard and the signing and encryption of any SOAP element within the message. There are two methods available for selection of elements to be signed and encrypted. The keyword-based method is easy to use and supports most common usage scenarios. The XPATH method, while more complex, allows you to select elements not supported by keywords to be signed and encrypted. Using these two methods, you can secure virtually any SOAP element in the SOAP message.

Note that the Signature element is not encrypted in this sample. The Basic Security Profile recommends encryption of the Signature element for a simple SOAP message, such as the one in this sample. You can use the following XPATH expression in the Confidentiality on the client side and RequiredConfidentiality on the service side to encrypt the Signature element:

```
/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/' and
local-name()='Envelope']/*[namespace-
uri()='http://schemas.xmlsoap.org/soap/envelope/' and local-
name()='Header']/*[namespace-uri()='http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd' and
local-name()='Security']/*[namespace-uri()='
http://www.w3.org/2000/09/xmldsig#' and local-name()='Signature']
```