WS-SecurityPolicy

WS-SecurityPolicy

CXF 2.2 introduced support for using WS-SecurityPolicy to configure WSS4J instead of the custom configuration documented on the WS-Security page. However, all of the "background" material on the WS-Security page still applies and is important to know. WS-SecurityPolicy just provides an easier and more standards based way to configure and control the security requirements. With the security requirements documented in the WSDL as WS-Policy fragments, other tools such as .NET can easily know how to configure themselves to inter-operate with CXF services.

CXF supports WS-SecurityPolicy versions 1.1 and later. It does not support WS-SecurityPolicy 1.0.

Backwards compatibility configuration note

From Apache CXF 3.1.0, some of the WS-Security based configuration tags have been changed to just start with "security-". This is so that they can be shared with the JAX-RS XML Security component. Apart from the prefix change, the tags are exactly the same. Older "ws-security-" values continue to be accepted in CXF 3.1.0. See the Security Configuration page for information on the new shared configuration tags.

Enabling WS-SecurityPolicy

In CXF 2.2, if the cxf-rt-ws-policy and cxf-rt-ws-security modules are available on the classpath, the WS-SecurityPolicy stuff is automatically enabled. Since the entire security runtime is policy driven, the only requirement is that the policy engine and security policies be available.

If you are using the full "bundle" jar, all the security and policy stuff is already included.

Policy description

With WS-SecurityPolicy, the binding and/or operation in the wsdl references a WS-Policy fragment that describes the basic security requirements for interacting with that service. The WS-SecurityPolicy specification allows for specifying things like asymmetric/symmetric keys, using transports (https) for encryption, which parts/headers to encrypt or sign, whether to sign then encrypt or encrypt then sign, whether to include timestamps, whether to use derived keys, etc... Basically, it describes what actions are necessary to securely interact with the service described in the WSDL.

However, the WS-SecurityPolicy fragment does not include "everything" that is required for a runtime to be able to able to create the messages. It does not describe things such as locations of key stores, user names and passwords, etc... Those need to be configured in at runtime to augment the WS-SecurityPolicy fragment.

Configuring the extra properties

There are several extra properties that may need to be set to provide the additional bits of information to the runtime. Note that you should check that a particular property is supported in the version of CXF you are using. First, see the Security Configuration page for information on the configuration tags that are shared with the JAX-RS XML Security component. Here are configuration tags that only apply to the WS-SecurityPolicy layer, and hence all start with "ws-security" (as opposed to the common tags which now start with "security-").

Boolean WS-Security configuration tags, e.g. the value should be "true" or "false".

constant	default	definition
ws-security.validate.token	true	Whether to validate the password of a received UsernameToken or not.
ws-security.username-token.always.encrypted	true	Whether to always encrypt UsernameTokens that are defined as a SupportingToken. This should not be set to false in a production environment, as it exposes the password (or the digest of the password) on the wire.
ws-security.is-bsp-compliant	true	Whether to ensure compliance with the Basic Security Profile (BSP) 1.1 or not.
ws-security.self-sign-saml-assertion	false	Whether to self-sign a SAML Assertion or not. If this is set to true, then an enveloped signature will be generated when the SAML Assertion is constructed. Only applies up to CXF 2.7.x.
ws-security.enable.nonce.cache	(varies)	Whether to cache UsernameToken nonces. See here for more information.

ws-security.enable.timestamp.cache	(varies)	Whether to cache Timestamp Created Strings. See here for more information.
ws-security.enable.saml.cache	(varies)	Whether to cache SAML2 Token Identifiers, if the token contains a "OneTimeUse" Condition.
ws-security.enable.streaming	false	Whether to enable streaming WS-Security.
ws-security.return.security.error	false	Whether to return the security error message to the client, and not one of the default error QNames.
ws-security.must-understand	true	Set this to "false" in order to remove the SOAP mustUnderstand header from security headers generated based on a WS-SecurityPolicy.
ws-security.store.bytes.in.attachment	(varies)	**CXF 3.1.3/3.0.6** Whether to store bytes (CipherData or BinarySecurityToken) in an attachment if MTOM is enabled. True by default in CXF 3.1.x, false for CXF 3.0.x.
ws-security.use.str.transform	true	**CXF 3.1.5/3.0.8** Whether to use the STR (Security Token Reference) Transform when (externally) signing a SAML Token. The default is true.
ws-security.add.inclusive.prefixes	true	**CXF 3.1.7** Whether to add an InclusiveNamespaces PrefixList as a CanonicalizationMethod child when generating Signatures using WSConstants.C14N_EXCL_OMIT_COMMENTS.
ws-security.expand.xop.include	(varies)	**CXF 3.3.3/3.2.10** Whether to search for and expand xop:Include Elements for encryption and signature (on the outbound side) or for signature verification (on the inbound side). This ensures that the actual bytes are signed, and not just the reference. The default is "true" if MTOM is enabled, false otherwise.

Non-boolean WS-Security Configuration parameters

ws-security.timestamp.timeToLive	The time in seconds to append to the Creation value of an incoming Timestamp to determine whether to accept the Timestamp as valid or not. The default value is 300 seconds (5 minutes).
ws-security.timestamp.futureTimeToLive	The time in seconds in the future within which the Created time of an incoming Timestamp is valid. The default value is "60". See here for more information.
ws-security.spnego.client.action	The SpnegoClientAction implementation to use for SPNEGO. This allows the user to plug in a different implementation to obtain a service ticket.
ws-security.nonce.cache.instance	This holds a reference to a ReplayCache instance used to cache UsernameToken nonces. The default instance that is used is the EHCacheReplayCache.
ws-security.timestamp.cache.instance	This holds a reference to a ReplayCache instance used to cache Timestamp Created Strings. The default instance that is used is the EHCacheReplayCache.
ws-security.saml.cache.instance	This holds a reference to a ReplayCache instance used to cache SAML2 Token Identifiers, when the token has a "OneTimeUse" Condition. The default instance that is used is the EHCacheReplayCache.
ws-security.cache.config.file	Set this property to point to a configuration file for the underlying caching implementation. The default configuration file that is used is cxf-ehcache.xml in the cxf-rt-ws-security module. From CXF 3.4.0, this cache file only applies to the TokenStore caching implementation, not for the WSS4J ReplayCache.

org.apache.cxf.ws.security.tokenstore.TokenStore	The TokenStore instance to use to cache security tokens. By default this uses the EHCacheTokenStore if EhCache is available. Otherwise it uses the MemoryTokenStore.
ws-security.cache.identifier	The Cache Identifier to use with the TokenStore. CXF uses the following key to retrieve a token store: "org.apache.cxf.ws.security.tokenstore.TokenStore-<identifier>". This key can be used to configure service-specific cache configuration. If the identifier does not match, then it falls back to a cache configuration with key "org.apache.cxf.ws.security.tokenstore.TokenStore". The default "<identifier>" is the QName of the service in question.
ws-security.role.classifier	If one of the WSS4J Validators returns a JAAS Subject from Validation, then the WSS4JInInterceptor will attempt to create a SecurityContext based on this Subject. If this value is not specified, then it tries to get roles using the DefaultSecurityContext in cxf-rt-core. Otherwise it uses this value in combination with the SUBJECT_ROLE_CLASSIFIER_TYPE to get the roles from the Subject.
ws-security.role.classifier.type	If one of the WSS4J Validators returns a JAAS Subject from Validation, then the WSS4JInInterceptor will attempt to create a SecurityContext based on this Subject. Currently accepted values are "prefix" or "classname". Must be used in conjunction with the SUBJECT_ROLE_CLASSIFIER. The default value is "prefix".
ws-security.asymmetric.signature.algorithm	This configuration tag overrides the default Asymmetric Signature algorithm (RSA-SHA1) for use in WS-SecurityPolicy, as the WS-SecurityPolicy specification does not allow the use of other algorithms at present.
ws-security.symmetric.signature.algorithm	This configuration tag overrides the default Symmetric Signature algorithm (HMAC-SHA1) for use in WS-SecurityPolicy, as the WS-SecurityPolicy specification does not allow the use of other algorithms at present.
ws-security.password.encryptor.instance	A PasswordEncryptor instance, which is used to encrypt or decrypt passwords in the Merlin Crypto implementation
ws-security.delegated.credential	A delegated credential to use for WS-Security. Currently only a Kerberos GSSCredential Object is supported. This is used to retrieve a service ticket instead of using the client credentials.
ws-security.security.token.lifetime	**CXF 3.1.9** The security token lifetime value (in milliseconds). The default is "300000" (5 minutes).

Validator implementations for validating received security tokens

ws-security.ut.validator	The WSS4J Validator instance to use to validate UsernameTokens. The default value is the UsernameTokenValidator.
ws-security.saml1.validator	The WSS4J Validator instance to use to validate SAML 1.1 Tokens. The default value is the SamlAssertionValidator.
ws-security.saml2.validator	The WSS4J Validator instance to use to validate SAML 2.0 Tokens. The default value is the SamlAssertionValidator.
ws-security.timestamp.validator	The WSS4J Validator instance to use to validate Timestamps. The default value is the TimestampValidator.
ws-security.signature.validator	The WSS4J Validator instance to use to validate trust in credentials used in Signature verification. The default value is the SignatureTrustValidator.
ws-security.bst.validator	The WSS4J Validator instance to use to validate BinarySecurityTokens. The default value is the NoOpValidator.
ws-security.sct.validator	The WSS4J Validator instance to use to validate SecurityContextTokens. The default value is the NoOpValidator.

Kerberos Configuration tags

constant	default	definition
ws-security.kerberos.request.credential.delegation	false	Whether to request credential delegation or not in the KerberosClient.
ws-security.kerberos.use.credential.delegation	false	Whether to use credential delegation or not in the KerberosClient.
ws-security.kerberos.is.username.in.servicename.form	false	Whether the Kerberos username is in servicename form or not.
ws-security.kerberos.client	n/a	A reference to the KerberosClient class used to obtain a service ticket.
ws-security.kerberos.jaas.context	n/a	The JAAS Context name to use for Kerberos.
ws-security.kerberos.spn	n/a	The Kerberos Service Provider Name (spn) to use.

Configuring via Spring

The properties are easily configured as client or endpoint properties--use the former for the SOAP client, the latter for the web service provider.

```
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:jaxws="http://cxf.apache.org/jaxws"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
   http://cxf.apache.org/jaxws
   http://cxf.apache.org/schemas/jaxws.xsd">

   <jaxws:client name="{http://cxf.apache.org}MyPortName"
      createdFromAPI="true">
      <jaxws:properties>
         <entry key="security.callback-handler"
             value="interop.client.KeystorePasswordCallback"/>
         <entry key="security.signature.properties"
             value="etc/client.properties"/>
         <entry key="security.encryption.properties"
             value="etc/service.properties"/>
         <entry key="security.encryption.username"
             value="servicekeyalias"/>
      </jaxws:properties>
   </jaxws:client>

</beans>
```

For the jaxws:client's *name* attribute above, use the namespace of the WSDL along with the *name* attribute of the desired wsdl:port element under the WSDL's service section. (See here and here for an example.)

```
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:jaxws="http://cxf.apache.org/jaxws"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
   http://cxf.apache.org/jaxws
   http://cxf.apache.org/schemas/jaxws.xsd">

   <jaxws:endpoint
      id="MyService"
      address="https://localhost:9001/MyService"
      serviceName="interop:MyService"
      endpointName="interop:MyServiceEndpoint"
      implementor="com.foo.MyService">

      <jaxws:properties>
         <entry key="security.callback-handler"
             value="interop.client.UTPasswordCallback"/>
         <entry key="security.signature.properties"
             value="etc/keystore.properties"/>
         <entry key="security.encryption.properties"
             value="etc/truststore.properties"/>
         <entry key="security.encryption.username"
             value="useReqSigCert"/>
      </jaxws:properties>

   </jaxws:endpoint>
</beans>
```

See this blog entry for a more end-to-end example of using WS-SecurityPolicy with X.509 keys.

Configuring via API's

Configuring the properties for the client just involves setting the properties in the client's RequestContext:

```
Map<String, Object> ctx = ((BindingProvider)port).getRequestContext();
ctx.put("security.encryption.properties", properties);
port.echoString("hello");
```