WS-Trust

WS-Trust

WS-Trust support in CXF builds upon the WS-SecurityPolicy implementation to handle the IssuedToken policy assertions that could be found in the WS-SecurityPolicy fragment.

Note: Because the WS-IssuedToken support builds on the WS-SecurityPolicy support, this is currently only available to "wsdl first" projects.

WS-Trust extends the WS-Security specification to allow issuing, renewing, and validation of security tokens. A lot of what WS-Trust does centers around the use of a "Security Token Service", or STS. The STS is contacted to obtain security tokens that are used to create messages to talk to the services. The primary use of the STS is to acquire SAML tokens used to talk to the service. Why is this interesting?

When using "straight" WS-Security, the client and server need to have keys exchanged in advance. If the client and server are both in the same security domain, that isn't usually a problem, but for larger, complex applications spanning multiple domains, that can be a burden. Also, if multiple services require the same security credentials, updating all the services when those credentials change can by a major operation.

WS-Trust solves this by using security tokens that are obtained from a trusted Security Token Service. A client authenticates itself with the STS based on policies and requirements defined by the STS. The STS then provides a security token (example: a SAML token) that the client then uses to talk to the target service. The service can validate that token to make sure it really came from the trusted STS.

When the WS-SecurityPolicy runtime in CXF encounters an IssuedToken assertion in the policy, the runtime requires an instance of org.apache.cxf.ws.security.trust.STSClient to talk to the STS to obtain the required token. Since the STSClient is a WS-SecurityPolicy client, it will need configuration items to be able to create its secure SOAP messages to talk to the STS.

General Configuration

There are several ways to configure the STSClient:

Direct configuration of an STSClient bean in the properties:
In this scenario, a STSClient object is created directly as a property of the client object. The wsdlLocation, service/endpoint names, etc... are all configured in line for that client.

```
<jaxws:client name="{http://cxf.apache.org/}MyService" createdFromAPI="true">
    <jaxws:properties>
        <entry key="ws-security.sts.client">
            <!-- direct STSClient config and creation -->
            <bean class="org.apache.cxf.ws.security.trust.STSClient">
                <constructor-arg ref="cxf"/>
                <property name="wsdlLocation"
                   value="target/wsdl/trust.wsdl"/>
                <property name="serviceName"
                   value="{http://cxf.apache.org/securitytokenservice}SecurityTokenService"/>
                <property name="endpointName"
                   value="{http://cxf.apache.org/securitytokenservice}SecurityTokenEndpoint"/>
                <property name="properties">
                    <map>
                       <entry key="ws-security.username" value="alice"/>
                       <entry key="ws-security.callback-handler"
                          value="client.MyCallbackHandler"/>
                       <entry key="ws-security.signature.properties"
                          value="clientKeystore.properties"/>
                       <entry key="ws-security.encryption.properties"
                          value="clientKeystore.properties"/>
                       <entry key="ws-security.encryption.username"
                          value="mystskey"/>
                    </map>
                </property>
            </bean>
        </entry>
    </jaxws:properties>
</jaxws:client>
```

The above example shows a configuration where the STS uses the UsernameToken profile to validate the client. It is assumed the keystore identified within clientKeystore.properties contains both the private key of the client and the public key (identified above as mystskey) of the STS; if not, create separate property files for the signature properties and the encryption properties, pointing to the keystore and truststore respectively.

Remember the jaxws:client createdFromAPI attribute needs to be set to true (as shown above) if you created the client programmatically via the CXF API's--i.e., Endpoint.publish() or Service.getPort().

This also works for "code first" cases as you can do:

```
STSClient sts = new STSClient(...);
sts.setXXXX(....)
.....
((BindingProvider)port).getRequestContext().put("ws-security.sts.client", sts);
```

Sample clientKeystore.properties format:

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=KeystorePasswordHere
org.apache.ws.security.crypto.merlin.keystore.alias=ClientKeyAlias
org.apache.ws.security.crypto.merlin.keystore.file=NameOfKeystore.jks
```

Indirect configuration based on endpoint name:
If the runtime does not find a STSClient bean configured directly on the client, it checks the configuration for a STSClient bean with the name of the endpoint appended with ".sts-client". For example, if the endpoint name for your client is "{http://cxf.apache.org/}TestEndpoint", then it can be configured as:

```
<bean name="{http://cxf.apache.org/}TestEndpoint.sts-client"
    class="org.apache.cxf.ws.security.trust.STSClient" abstract="true">
    <property name="wsdlLocation" value="WSDL/wsdl/trust.wsdl"/>
    <property name="serviceName"
        value="{http://cxf.apache.org/securitytokenservice}SecurityTokenService"/>
    <property name="endpointName"
        value="{http://cxf.apache.org/securitytokenservice}SecurityTokenEndpoint"/>
    <property name="properties">
        <map>
            <entry key="ws-security.signature.properties"
                value="etc/alice.properties"/>
            <entry key="ws-security.encryption.properties"
                value="etc/bob.properties"/>
            <entry key="ws-security.encryption.username" value="stskeyname"/>
        </map>
    </property>
</bean>
```

This properties configured in this example demonstrate STS validation of the client using the X.509 token profile. The abstract="true" setting for the bean defers creation of the STSClient object until it is actually needed. When that occurs, the CXF runtime will instantiate a new STSClient using the values configured for this bean.

Default configuration:
If an STSClient is not found from the above methods, it then tries to find one configured like the indirect, but with the name "default.sts-client". This can be used to configure sts-clients for multiple services.

WS-Trust 1.4 Support

CXF supports some of the new functionality defined in the WS-Trust 1.4 specification. The currently supported features are listed below.

ActAs

The ActAs capability allows an initiator to request a security token that allows it to act as if it were somebody else. This capability becomes important in composite services where intermediate services make additional requests on-behalf of the true initiator. In this scenario, the relying party (the final destination of an indirect service request) may require information about the true origin of the request. The ActAs capability allows an intermediary to request a token that can convey this information.

The content of the ActAs element to be sent in the STS RequestSecurityToken call can be set in one of two ways:

1. By specifying a value for the JAX-WS property SecurityConstants.STS_TOKEN_ACT_AS ("ws-security.sts.token.act-as")
2. By specifying a value for the STSClient.actAs property.

For either case, the value can be one of the following:

- A String
- A DOM Element
- A CallbackHandler object to use to obtain the token

For example, the following code fragment demonstrates how to use an interceptor to dynamically set the content of the ActAs element in the STS RST, by specifying a value for SecurityConstants.STS_TOKEN_ACT_AS. Note that this interceptor is applied to the secured client, the initiator, and not to the STSClient's interceptor chain.

```
public class ActAsOutInterceptor extends AbstractPhaseInterceptor<Message> {

    ActAsOutInterceptor () {
        // This can be in any stage before the WS-SP interceptors
        // setup the STS client and issued token interceptor.
        super(Phase.SETUP);
    }

    @Override
    public void handleMessage(Message message) throws Fault {
        message.put(SecurityConstants.STS_TOKEN_ACT_AS, ...);
    }
}
```

Alternatively, the ActAs content may be set directly on the STS as shown below.

```
<bean name="{http://cxf.apache.org/}TestEndpoint.sts-client"
    class="org.apache.cxf.ws.security.trust.STSClient" abstract="true">
    <property name="wsdlLocation" value="WSDL/wsdl/trust.wsdl"/>
    <property name="serviceName"
        value="{http://cxf.apache.org/securitytokenservice}SecurityTokenService"/>
    <property name="endpointName"
        value="{http://cxf.apache.org/securitytokenservice}SecurityTokenEndpoint"/>
    <property name="actAs" value="..."/>
    <property name="properties">
        <map>
            ...
        </map>
    </property>
</bean>
```

OnBehalfOf

The OnBehalfOf capability allows an initiator to request a security token on behalf of somebody else. The content of the OnBehalfOf element to be sent in the STS RequestSecurityToken call can be set in one of two ways:

1. By specifying a value for the JAX-WS property SecurityConstants.STS_TOKEN_ON_BEHALF_OF ("ws-security.sts.token.on-behalf-of")
2. By specifying a value for the STSClient.onBehalfOf property.

For either case, the value can be one of the following:

- A String
- A DOM Element
- A CallbackHandler object to use to obtain the token

WS-Trust using SPNego

As of CXF 2.4.7 and 2.5.3, CXF contains (client) support for WS-Trust using SPNego. See the following blog for an explanation of what this entails, and how to run some system tests in CXF for this feature.

WS-Trust using XKMS

Since CXF 2.7.7 Security Token Service (STS) can be configured to use XKMS Crypto provider. In this case X509 certificates can be located centrally and managed using standard XKMS interface. STS will automatically invoke XKMS client for locate or validate corresponded X509 certificate. See the following blog for the details and sample.

This feature can be especially useful for STS scenario with SymmetricKey. With this scenario, the STS and the WS consumer negotiate a symmetric key:

1. The WS-Client authenticates himself to STS and contributes material to the creation of symmetric key.
2. The STS verifies WS-Client authentication and generates symmetric key using material received from WS-Client
3. The STS encrypts symmetric key using WS-Service public key and inserts the encrypted key together with security token into SAML assertion
4. The STS signs SAML assertion and sends it together with key material for generation symmetric key to the WS-Client.
5. The WS-Client generates short-lived symmetric key from own material and the key material from the STS.

6. The WS-Client inserts the SAML token, into the message header. It encrypts the message texts or/and signs the message with the generated symmetric key. It then sends the user's message to the WS-Service.
7. The WS-Service checks the signature in the SAML token and uses its private key to decrypt the symmetric key contained in the SAML token.
8. The WS-Service verifies the signature of the WS-Client (Holder-of-Key) with the decrypted symmetric key. In this way, the STS confirms that the Holder-of-Key is the subject (the user) in the assertion.
9. The WS-Service uses the symmetric key to decrypt the message text.

On the step (3) STS needs the public key (certificate) of target WS-Service. Normally STS servers not only one, but multiple services (restricted by url patterns in TokenServiceProvider). This can be a serious drawback to manage public certificates of all services into STS local keystore.

XKMS Crypto provider provides elegant solution of this using following configuration:

- encryptionUsername (in StaticSTSProperties or jaxws:endpoint properties) should be set into special value: *useEndpointAsCertAlias* (STSConstants.USE_ENDPOINT_AS_CERT_ALIAS)
- encryptionCrypto should be set to XKMS Crypto implementation
- Service certificates should be saved into XKMS under service endpoint (use Application "*urn:apache:cxf:service:endpoint*" and service endpoint as identifier)

In this case STS recognizes encryptionName constant and will ask XKMS Crypto for the service certificate using AppliesTo endpoint address. XKMS will locate service certificate using this endpoint address.

STS can server multiple WS-Services and doesn't care about services certificates locally - they are stored and managed in central XKMS repository.

The following blog explains the details and contains the sample code.

Blogs on WS-Trust in CXF

Some blogs for up-to-date information about WS-Trust and other security topics in CXF:
http://coheigea.blogspot.com/
http://owulff.blogspot.com/
http://janbernhardt.blogspot.com/