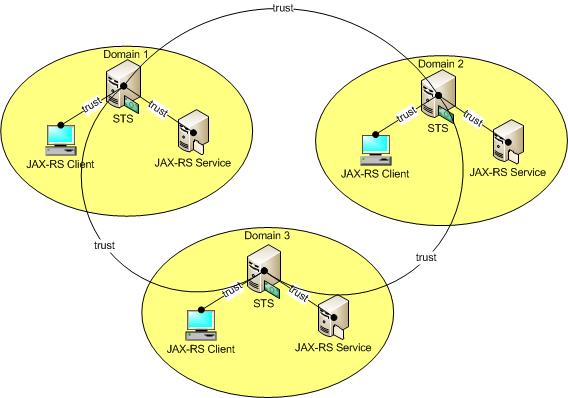
**Wednesday, 1 June 2016**

[CXF JAX-RS Security: SAML Token validation with SecurityTokenService](http://ashakirin-cxf-security.blogspot.ca/2016/06/cxf-jax-rs-security-saml-token.html)

**Introduction**

In the previous blog [CXF JAX-RS Security: authentication with SAML Token using SecurityTokenService](http://ashakirin-cxf-security.blogspot.de/2016/03/cxf-jax-rs-security-authentication-with.html)we have seen how to configure JAX-RS Client to obtain SAML from STS and send it to the JAX-RS service in the REST request. The token was validated locally by the JAX-RS service.  
  
This approach works perfectly, but in some scenarios such validation is inconvenient or even impossible. Imagine that we have complex distributed system with multiple domains. Every domain has own STS. There is a trust relationship between all Security Token Services, but services trust only STS from own domain:

[](https://1.bp.blogspot.com/-W25NBkK7H6I/Vzy9wH3K9CI/AAAAAAAAAMg/uOoZzlaebbEoq6MffmweKDhX5A2X12LnACLcB/s1600/multi-dokmain-sts.jpg)

Assume that the domain1 client calls domain2 service. SAML token in client request is signed by domain1 STS. Domain2 service hasn't trust relationship with domain1 STS and cannot validate SAML token, received from client, locally.  
What is the solution? Domain2 service delegates SAML validation to own STS service, which has trust relationship with domain1 STS.

**JAX-RS Client**

The client configuration will be the same as in previous blog: no any changes in spring configuration nor in keystore are necessary.

**JAX-RS Service**

Service configuration and keystore have to be updated for the communication with STS service in order to validate SAML token:

    ...

    <bean id="stsClient" class="org.apache.cxf.ws.security.trust.STSClient">  
        <constructor-arg ref="cxf" />  
        <property name="wsdlLocation" value="http://localhost:8080/sts/STS/X509?wsdl" />  
        <property name="serviceName"  
            value="{http://docs.oasis-open.org/ws-sx/ws-trust/200512/}SecurityTokenService" />  
        <property name="endpointName"  
            value="{http://docs.oasis-open.org/ws-sx/ws-trust/200512/}X509\_Port" />  
        <property name="properties">  
            <map>  
                <entry key="ws-security.callback-handler"  value="demo.jaxrs.saml.server.ServiceCallbackHandler" />  
                <entry key="ws-security.sts.token.username" value="myservicekey" />  
                <entry key="security.signature.properties" value="serviceKeystore.properties" />          
                <entry key="ws-security.sts.token.properties" value="serviceKeystore.properties" />  
            </map>  
        </property>  
    </bean>

    <bean id="samlValidator"  
        class="org.apache.cxf.rs.security.saml.SamlHeaderInHandler">  
        <property name="validator">  
            <bean class="org.apache.cxf.ws.security.trust.STSTokenValidator">  
                 <constructor-arg value="true" />  
                 <property name="stsClient" ref="stsClient"/>  
            </bean>  
        </property>  
    </bean>

    ...

The STSClient bean is declared and configured with appropriate STS WSDL location, service name, endpoint name and security properties. This client is injected into samlValidator bean to enable remote SAML validation by STS. Note: by default, STSTokenValidator at first tries to verify SAML signature locally and only if local keystore doesn't contain STS certificate as trusted entry, delegates validation to STS service. It is possible to change this behavior using STSTokenValidator constructor argument alwaysValidateToSts. If this argument is set to true, CXF skip local validation and immediately delegates validation to STS server.  
  
The service keystore must contain STS certificate as trusted entry to enable secure communication with STS.

**STS Service**

Single change in STS Service is additional trust entry in keystore - it is JAX-RS service certificate. It is necessary to enable secure communication between STS and JAX-RS service.

**Example and source code**

Updated [sources](http://ashakirin/cxf-jaxrs-security/tree/master/jaxrs-saml)are located on the GitHub. JAX-RS Service configuration is stored into separate configuration file (server-context-sts-validation.xml). Therefore example code can be run with STS validation or without it depending on used spring configuration.

Of course, the same functionality can be implemented as well programmatically, without Spring.

Posted by [Andrei Shakirin](https://www.blogger.com/profile/00740530708764404230)at [00:19](http://ashakirin-cxf-security.blogspot.ca/2016/06/cxf-jax-rs-security-saml-token.html) [No comments:](http://ashakirin-cxf-security.blogspot.ca/2016/06/cxf-jax-rs-security-saml-token.html#comment-form)

[Email This](https://www.blogger.com/share-post.g?blogID=5743243276808589319&postID=3253622624673839272&target=email)[BlogThis!](https://www.blogger.com/share-post.g?blogID=5743243276808589319&postID=3253622624673839272&target=blog)[Share to Twitter](https://www.blogger.com/share-post.g?blogID=5743243276808589319&postID=3253622624673839272&target=twitter)[Share to Facebook](https://www.blogger.com/share-post.g?blogID=5743243276808589319&postID=3253622624673839272&target=facebook)[Share to Pinterest](https://www.blogger.com/share-post.g?blogID=5743243276808589319&postID=3253622624673839272&target=pinterest)

**Wednesday, 30 March 2016**

[CXF JAX-RS Security: authentication with SAML Token using SecurityTokenService](http://ashakirin-cxf-security.blogspot.ca/2016/03/cxf-jax-rs-security-authentication-with.html)

**Introduction**

Authentication with [SAML Token](http://saml.xml.org/saml-specifications)  is widely used not only in SOAP world, but also for the REST services. SAML provides the number of benefits like:

* standardization (SAML is widely accepted standard)
* security (user credentials never leave the firewall boundary)
* Single Sign On option (SAML token can be used to access multiple applications with a credentials entered only once)
* extensibility (using custom assertion attributes)

[Apache CXF framework](https://cxf.apache.org/) supports [SAML authentication](http://cxf.apache.org/docs/jax-rs-saml.html) for JAX-RS out the box. There are two main approaches to issue SAML token in JAX-RS applications:

1. JAX-RS client issues SAML token itself (using interceptor) and sends it to the JAX-RS service.
2. JAX-RS client delegates creation of SAML token to SecurityTokeService (STS) and sends the received token to JAX-RS service.

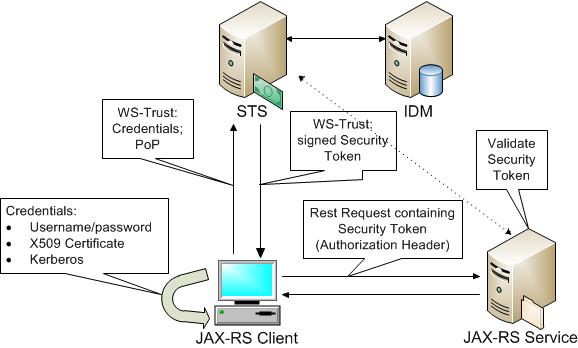
The first approach is briefly described in [CXF Wiki](http://cxf.apache.org/docs/jax-rs-saml.html#JAX-RSSAML-CreatingSAMLAssertions) and illustrated in this [system test](http://svn.apache.org/repos/asf/cxf/trunk/systests/rs-security/src/test/java/org/apache/cxf/systest/jaxrs/security/saml/).  
Approach works in many cases, however it has some drawbacks:

* In case of using multiple clients, the server have to trust all SAML issuer certificates
* Client have to know a lot of details about issuing SAML: versions, subject, confirmation method, private keys, certificates

The second approach helps to resolve these drawbacks using central SecurityTokenService (STS) to issue the SAML tokens. In this case JAX-RS client communicates with STS using WS-Trust protocol and delegate creation of SAML Token to STS. This post describes mainly the STS based approach.

**Communication**

The communication between actors is depicted bellow:

[](https://1.bp.blogspot.com/-OufhEL7sA2Q/VuXDpubt1tI/AAAAAAAAAMM/38kQu4NQu4IQ-3YNYaPsSihiuF-OZtNgw/s1600/authentication-rest-sts.jpg)

1. JAX-RS client sends request to STS using WS-Trust protocol. In order to authenticate the end user, this request contains Proof of Possession (PoP) (UsernameToken, X509 with signature, Kerberos, etc)
2. STS verifies user credentials (normally against Identity Management System)
3. If verification is successful, STS issues SAML token and sends it back to the JAX-RS client
4. JAX-RS client injects SAML Token into HTTP request (using HTTP Authorization header, Form values or enveloped)
5. JAX-RS Service extracts SAML token, verifies STS certificate, signature and expire date. Optionally JAX-RS service can delegate validation of SAML token to STS.
6. If verification was successful, STS Service processes request and sends response

**Configuration and source code**

The [source code](http://ashakirin/cxf-jaxrs-security/tree/master/jaxrs-saml)of JAX-RS client, JAX-RS service and STS with running instructions is available on the GitHub  
Let  go through all components step by step:

**Security Token Service**

 STS code was build on the base of  [Glen Mazza's Tutorial](https://web-gmazza.rhcloud.com/blog/entry/cxf-sts-tutorial) and contains following parts:

1. CXF configuration (cxf-servlet.xml) exposing two endpoints for UsernameToken and X509 based authentications
2. Keystore stskeystore.jks and stsKeystore.properties containing STS public/private key paar and trusted entry with client certificate
3. PasswordCallbackHandler providing password for STS private key

Note:

* stskeystore.jks must contains either client certificate itself or client CA certificate as trusted entry to be able to verify validity of client certificate (in case of X509 authentication)
* x509Endpoints bean in cxf-servlet.xml contains the list of JAX-RS Service endpoints for which SAML will be issued. Currently STS configured to accept any service endpoint.

STS can be either deployed to tomcat as war or started used tomcat plugin (start mvn tomcat7:run). X509 authentication doesn't requires HTTPS.

**JAX-RS Client**

In CXF versions < 3.1.2 it was required to configure complete STSClient in order to communicate with SecurityTokenService. The change request [#CXF-6267](https://issues.apache.org/jira/browse/CXF-6267) helps to make this configuration a bit easier.

It is necessary to instantiate SamlHeaderOutInterceptor and STSTokenOutInterceptor with two constructor arguments: STS endpoint and authentication parameters:

 <bean id="samlHeadOutInterceptor" class="org.apache.cxf.rs.security.saml.SamlHeaderOutInterceptor" />  
  
<bean id="issuedTokenInterceptor"  
    class="org.apache.cxf.ws.security.policy.interceptors.STSTokenOutInterceptor">  
    <constructor-arg name="authParams" ref="authParams" />  
    <constructor-arg name="stsWsdlLocation"  
        value="http://localhost:8080/sts/STS/X509?wsdl" />  
    <constructor-arg name="bus" ref="cxf" />  
</bean>  
  
<bean id="authParams"  
    class="org.apache.cxf.ws.security.policy.interceptors.STSTokenOutInterceptor$AuthParams">  
    <constructor-arg name="authMode" value="X509" />  
    <constructor-arg name="userName" value="alice" />  
    <constructor-arg name="alias" value="myclientkey" />  
    <constructor-arg name="callbackHandler"  
        value="demo.jaxrs.saml.client.ClientCallbackHandler" />  
    <constructor-arg name="keystoreProperties" value="clientKeystore.properties" />  
</bean>

The authentication parameters bean specifies:

1. Authentication mode (X509 or TRANSPORT, I use X509 in this post)
2. User name
3. Keystore alias
4. Keystore password callback handler
5. Keystore properties

The SamlHeaderOutInterceptor is required to set received SAML Token into authorization header.  
  
After that, JAX-RS client needs to configure these two interceptors:      
  
<jaxrs:client id="bookStoreProxy" address="https://localhost:9000"  
    serviceClass="demo.jaxrs.saml.common.BookStore">  
    <jaxrs:providers>  
        ...  
    </jaxrs:providers>  
    <jaxrs:outInterceptors>  
        <ref bean="issuedTokenInterceptor" />  
        <ref bean="samlHeadOutInterceptor" />  
    </jaxrs:outInterceptors>  
</jaxrs:client>  
  
That's all.  
In case of X509 based authentication client sends to STS request, containing user certificate and signed with user private key. STS validates the certificate and checks the signature. If both are valid,  Proof Of Possession (PoP) is successful and STS issues SAML Token for this user. SamlHeaderOutInterceptor inserts SAML received from STS into HTTP request Authorization header.  
Client keystore must contain following entries:

1. User certificate and corresponded private key to sign STS request and establish SSL connection with JAX-RS service (myclientkey alias)
2. STS certificate or it's CA certificate  as trusted entry to validate trust relationship with STS server (mystskey alias)
3. JAX-RS service certificate as trusted entry to establish SSL connection with JAX-RS service (myservicekey alias), see Client SSL Configuration for details.

**Client SSL Configuration**

In order to verify SAML token, the JAX-RS service should not only check SAML validity and signature itself, but also ensure that request containing SAML was send by trusted client. [SAML Subject Confirmation](http://dulanja.blogspot.de/2013/01/saml-subject-confirmation-methods.html) methods address this problem. By default our JAX-RS client will use Holder Of Key (HoK) Subject Confirmation method. This means that the client must sign the request or part of the request with the private key corresponding to client certificate inside the SAML token. CXF provides two ways to achive this:

1. Sign XML payload (works only for XML media types)
2. Use client private and public keys in SSL connection

In this post I use the second method as the most flexible and independent on media type. Therefore it is necessary to configure SSL connection between JAX-RS client and JAX-RS service using appropriate client keys:  
  
<http:conduit name="https://localhost:9000.\*">  
    <http:client ConnectionTimeout="3000000" ReceiveTimeout="3000000" />  
    <http:tlsClientParameters>  
        <sec:keyManagers keyPassword="ckpass">  
            <sec:keyStore file="src/main/resources/clientstore.jks"  
                password="cspass" type="JKS" />  
        </sec:keyManagers>  
        <sec:trustManagers>  
            <sec:keyStore file="src/main/resources/clientstore.jks"  
                password="cspass" type="JKS" />  
        </sec:trustManagers>  
    </http:tlsClientParameters>  
</http:conduit>

 I used the same clientstore.jks for SSL communication as in STSTokenOutInterceptor. Now JAX-RS service will be able to compare SSL client certificate and certificate from SAML token and ensure that the client owns private key corresponds to this certificate.

**JAX-RS Service**

Configuration of JAX-RS service is quite simple:

<bean id="bookStore" class="demo.jaxrs.saml.server.BookStoreImpl" />  
  
<bean id="samlValidator" class="org.apache.cxf.rs.security.saml.SamlHeaderInHandler">  
    <property name="validator">  
        <bean class="org.apache.wss4j.dom.validate.SamlAssertionValidator" />  
    </property>  
</bean>  
  
<jaxrs:server id="bookStoreServer" address="https://localhost:9000/">  
    <jaxrs:serviceBeans>  
        <ref bean="bookStore" />  
    </jaxrs:serviceBeans>  
    <jaxrs:providers>  
        <ref bean="samlValidator" />  
        ...  
    </jaxrs:providers>  
    <jaxrs:properties>  
        <entry key="ws-security.signature.properties" value="serviceKeystore.properties" />  
    </jaxrs:properties>  
</jaxrs:server>

I configured default STSTokenValidator and specifies serviceKeystore.properties as signature properties.

Service keystore must contain following entries:

1. Service certificate and corresponded private key to establish SSL connection with JAX\_RS client (myservicekey alias)
2. STS certificate or it's CA certificate  as trusted entry to validate trust relationship with STS server (mystskey alias)
3. JAX-RS client certificate as trusted entry to establish SSL connection with JAX-RS client (myclientkey alias), see Client SSL Configuration for details.

Of course, JAX-RS service need SSL configuration as well:  
  
<httpj:engine-factory bus="cxf">  
    <httpj:engine port="9000">  
        <httpj:tlsServerParameters>  
            <sec:keyManagers keyPassword="skpass">  
                <sec:keyStore file="src/main/resources/servicestore.jks"  
                    password="sspass" type="JKS" />  
            </sec:keyManagers>  
            <sec:trustManagers>  
                <sec:keyStore file="src/main/resources/servicestore.jks"  
                    password="sspass" type="JKS" />  
            </sec:trustManagers>  
            <sec:clientAuthentication required="true" />  
        </httpj:tlsServerParameters>  
    </httpj:engine>  
</httpj:engine-factory>  
  
The JAX-RS service will receive request with SAML token, extract the token, validate expire day, conditions, signature, verifies trust relationship with STS server and Holder Of Key Subject Confirmation using SSL client certificate. If all validations are successful - service operation will be called, otherwise client will receive 401 Unauthorized response.

Posted by [Andrei Shakirin](https://www.blogger.com/profile/00740530708764404230)at [12:09](http://ashakirin-cxf-security.blogspot.ca/2016/03/cxf-jax-rs-security-authentication-with.html) [No comments:](http://ashakirin-cxf-security.blogspot.ca/2016/03/cxf-jax-rs-security-authentication-with.html#comment-form)

[Email This](https://www.blogger.com/share-post.g?blogID=5743243276808589319&postID=2480253113703568239&target=email)[BlogThis!](https://www.blogger.com/share-post.g?blogID=5743243276808589319&postID=2480253113703568239&target=blog)[Share to Twitter](https://www.blogger.com/share-post.g?blogID=5743243276808589319&postID=2480253113703568239&target=twitter)[Share to Facebook](https://www.blogger.com/share-post.g?blogID=5743243276808589319&postID=2480253113703568239&target=facebook)[Share to Pinterest](https://www.blogger.com/share-post.g?blogID=5743243276808589319&postID=2480253113703568239&target=pinterest)

Labels: [Certificate](http://ashakirin-cxf-security.blogspot.ca/search/label/Certificate), [CXF](http://ashakirin-cxf-security.blogspot.ca/search/label/CXF), [SAML](http://ashakirin-cxf-security.blogspot.ca/search/label/SAML), [Security](http://ashakirin-cxf-security.blogspot.ca/search/label/Security), [STS](http://ashakirin-cxf-security.blogspot.ca/search/label/STS), [WS-Trust](http://ashakirin-cxf-security.blogspot.ca/search/label/WS-Trust), [X509](http://ashakirin-cxf-security.blogspot.ca/search/label/X509)

**Saturday, 1 March 2014**

[Using WS-Policy in CXF projects](http://ashakirin-cxf-security.blogspot.ca/2014/03/using-ws-policy-in-cxf-projects.html)

WS-Policy provides flexible mechanism to activate desired functionality on the client or service sides. Article describes how to define policies in custom CXF projects, implement policy-aware interceptors and explains some aspects of internal CXF design regarding WS-Policy.

**How to define policies**

There are basically 3 main possibilities to define WS-Policy in CXF projects:

1. WSDL Policy attachment
2. Spring configuration
3. Dynamically via message context property

Let look into them in details.

**WSDL Policy attachment**

WS-Policies can be attached and referenced in WSDL elements. [Web Services Policy 1.5 - Attachment](http://www.blogger.com/blogger.g?blogID=3247277779350112227) standard describes all possible alternatives. WS-Policies can be placed inside WSDL itself or referenced as external documents. CXF will automatically recognize, read and use policies defined or referenced in WSDL. Sample of attached policy is shown below:

<wsdl:definitions name="HelloWorld" targetNamespace="http://apache.org/hello\_world\_soap\_http"

…

<wsdl:service name="SOAPService">

<wsdl:port binding="tns:Greeter\_SOAPBinding" name="SoapPort">

<soap:address location="http://localhost:9000/SoapContext/SoapPort"/>

<wsp:Policy xmlns:wsp="http://www.w3.org/ns/ws-policy">

<wsam:Addressing xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">

<wsp:Policy/>

</wsam:Addressing>

</wsp:Policy>

</wsdl:port>

</wsdl:service>

</wsdl:definitions>

**Spring configuration**

It is possible to define policies directly in Spring configuration of client and service as jaxws feature. CFX will recognize and use configured WS-Policies:  
Client:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:jaxws="http://cxf.apache.org/jaxws"

xmlns:cxf="http://cxf.apache.org/core"

xmlns:p="http://cxf.apache.org/policy"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="

http://cxf.apache.org/core http://cxf.apache.org/schemas/core.xsd

http://cxf.apache.org/policy http://cxf.apache.org/schemas/policy.xsd

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"

http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">

<jaxws:client id="CRMServiceClient" name="{http://services.talend.org/CRMService}CRMServiceProvider"

xmlns:serviceNamespace="http://services.talend.org/CRMService"

serviceClass="org.talend.services.crmservice.CRMService"

serviceName="serviceNamespace:CRMServiceProvider"

endpointName="serviceNamespace:CRMServicePort"

address="${endpoint.prefix}/CRMServiceProvider">

<jaxws:features>

<p:policies>

<wsp:PolicyReference xmlns:wsp="http://www.w3.org/ns/ws-policy" URI="classpath:/saml.policy"/>

</p:policies>

</jaxws:features>

</jaxws:client>

</beans>

Service:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:jaxws="http://cxf.apache.org/jaxws"

xmlns:cxf="http://cxf.apache.org/core"

xmlns:p="http://cxf.apache.org/policy"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="

http://cxf.apache.org/core http://cxf.apache.org/schemas/core.xsd

http://cxf.apache.org/policy http://cxf.apache.org/schemas/policy.xsd

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"

http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">

<jaxws:endpoint id="CRMService"

xmlns:serviceNamespace="http://services.talend.org/CRMService"

serviceName="serviceNamespace:CRMServiceProvider"

endpointName="serviceNamespace:CRMServicePort"

implementor="#CRMServiceBean"

address="/CRMServiceProvider">

<jaxws:features>

<p:policies>

<wsp:PolicyReference xmlns:wsp="http://www.w3.org/ns/ws-policy" URI="classpath:/saml.policy"/>

</p:policies>

</jaxws:features>

</jaxws:endpoint>

</beans>

**Dynamically through message property**

Sometimes policies cannot be configured statically, because they are obtained or calculated dynamically for concrete message (for example using Policy Server or Service Registry). For such cases CXF provide a possibility to load policy dynamically and set it into the message context property. It can be done for example in custom interceptor that fulfils the following:

1. Get policy from external location and build it for current message.
2. Parse WS-Policy XML using Neethi library.
3. Store result Policy object into *PolicyConstants.POLICY\_OVERRIDE* message content property.  
   Important is that this custom policy interceptor is called before CXF *PolicyInInterceptor* or *PolicyOutInterceptor*. Than CXF will automatically recognize Policy stored into this property and use it with highest priority.

 I have published a small [sample](https://github.com/ashakirin/cxf.howtos/tree/master/ws-policy.dynamic) illustrating how to apply policy dynamically.

**Create custom policy assertions and associate interceptors**

It is quite easy to define own policy assertions and associate interceptors with it. Topic is already well described in CXF document <http://cxf.apache.org/docs/developing-assertions.html>, I just provide a list of steps necessary to do:

1. Provide Assertion Builder class for custom assertion implementing *AssertionBuilder<T>* interface.  
   Interface type can be *Element*, *XMLStreamReader* or *OMElement*.   
   Interface contains two methods: *build()* and *getKnownElements()*.   
   Implementation of *build()* method should construct Assertion from the incoming type. It can be *PrimitiveAssertion* (without attributes or child elements), NestedPrimitiveAssertion (without attributes but with nested policy element) and *JaxbAssertion* (assertion described by any XML schema).  
   *getKnownElements()* method must return QNames of assertion elements from which assertion can be built.
2. Implement policy interceptor provider class extending *AbstractPolicyInterceptorProvider* class. The main task of policy interceptor provider is to say which interceptors must be activated for specified policy assertion. Policy interceptor provider constructor gives assertions QNames as argument of super constructor and adds corresponded interceptors using getters:

public class AuthorizationInterceptorProvider extends AbstractPolicyInterceptorProvider {

private static final long serialVersionUID = -5248428637449096540L;

private static final AuthorizationInInterceptor IN\_AUTHZ\_INTERCEPTOR = new AuthorizationInInterceptor();

private static final AuthorizationInInterceptor OUT\_AUTHZ\_INTERCEPTOR = new AuthorizationOutInterceptor();

private static final Collection<QName> ASSERTION\_TYPES;

static {

ASSERTION\_TYPES = new ArrayList<QName>();

ASSERTION\_TYPES.add(AuthorizationConstants.AUTHORIZATION\_ASSERTION);

}

public AuthorizationInterceptorProvider() {

super(ASSERTION\_TYPES);

getInInterceptors().add(IN\_AUTHZ\_INTERCEPTOR);

getOutInterceptors().add(OUT\_AUTHZ\_INTERCEPTOR);

}

}

Assertion builder and policy interceptor provider can be registered using CXF bus extension mechanism: just create a file META-INF/cxf/bus-extensions.txt containing the following:

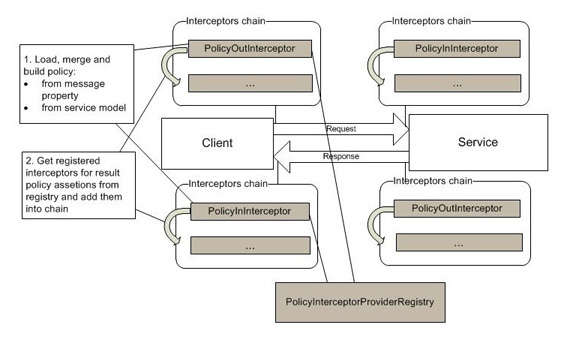
org.company.AuthorizationInterceptorProvider::true

org.company.AuthorizationAssertionBuilder::true

Boolean value at the end specifies lazy loading strategy.   
CXF automatically recognizes the assertion builder and policy interceptor provider and store them into registries: *AssertionBuilderRegistry* and *PolicyInterceptorProviderRegistry*. Since CXF 2.6.0 it is possible to register multiple interceptor providers for single assertion.

**How and where CXF processes policies**

As I already mentioned, CXF provides two interceptors: org.apache.cxf.ws.policy.PolicyInInterceptor and org.apache.cxf.ws.policy.PolicyOutInterceptor. These interceptors are responsible to load policy from destination, parse, merge them and add all associated interceptors into message interceptor chain. Functionality of policy interceptors are represented on the following figure:

[](http://3.bp.blogspot.com/-StcWQvRzkWU/T0t6lCTYakI/AAAAAAAAABA/LYcU0RoRLN8/s1600/cxf-ws-policies.jpg)

Briefly, policy interceptors make following steps:

1. Check message property *PolicyConstants.POLICY\_OVERRIDE*.
2. If *PolicyConstants.POLICY\_OVERRIDE* contains policy, it will be taken for further processing.
3. If property is empty, policy will be asked from ServiceModel. Here CXF loads policies attached to WSDL or provided via Spring configuration.
4. If any policy on step 2 or step 3 is found, *EffectivePolicy* will be created. Appropriate WS-policies will be merged for the current message and built into Neethi *Policy* object.
5. All interceptors registered for result policy assertions will be added to message interceptor chain.

Additionally, CXF verifies satisfied policy assertions in *PolicyVerfificationInInterceptor*, *PolicyVerificationInFaultInterceptor*, *PolicyVerificationOutInterceptor*. If assertion is not processed and not satisfied in corresponded interceptor, than In- interceptors throw Fault and Out- interceptors provide appropriate log messages.  
The practical using of WS-Policy is illustrated in ws\_policy and ws\_security CXF samples.

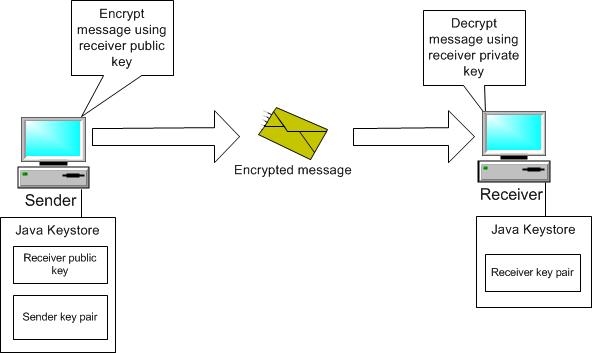
Posted by [Andrei Shakirin](https://www.blogger.com/profile/00740530708764404230)at [09:24](http://ashakirin-cxf-security.blogspot.ca/2014/03/using-ws-policy-in-cxf-projects.html) [No comments:](http://ashakirin-cxf-security.blogspot.ca/2014/03/using-ws-policy-in-cxf-projects.html#comment-form)

[Email This](https://www.blogger.com/share-post.g?blogID=5743243276808589319&postID=6977140226715715946&target=email)[BlogThis!](https://www.blogger.com/share-post.g?blogID=5743243276808589319&postID=6977140226715715946&target=blog)[Share to Twitter](https://www.blogger.com/share-post.g?blogID=5743243276808589319&postID=6977140226715715946&target=twitter)[Share to Facebook](https://www.blogger.com/share-post.g?blogID=5743243276808589319&postID=6977140226715715946&target=facebook)[Share to Pinterest](https://www.blogger.com/share-post.g?blogID=5743243276808589319&postID=6977140226715715946&target=pinterest)

Labels: [CXF](http://ashakirin-cxf-security.blogspot.ca/search/label/CXF), [Security](http://ashakirin-cxf-security.blogspot.ca/search/label/Security), [WS-Policy](http://ashakirin-cxf-security.blogspot.ca/search/label/WS-Policy), [WS-PolicyAttachment](http://ashakirin-cxf-security.blogspot.ca/search/label/WS-PolicyAttachment)

[CXF security: getting certificates from central PKI](http://ashakirin-cxf-security.blogspot.ca/2014/03/cxf-security-getting-certificates-from.html)

CXF security uses asymmetric algorithms for different purposes: encryption of symmetric keys and payloads, signing security token and messages, SSL transport bindings.  
  
If you look in samples and tutorials, the public keys (in form of X509 certificates) are normally stored in java keystores.  
  
For example, if sender encrypts the message payload sending to the receiver, he should have access to receiver certificate saved in local keystore. The sender uses this certificate for message encryption and receiver decrypts request with corresponded own private key:

[](http://1.bp.blogspot.com/-etDNwkUog6Y/UVmN_ScEebI/AAAAAAAAAE8/YyJd7ldY95o/s1600/classic-message-encryption.jpg)

Seems to be OK? Imagine now that you have production environment with 100 different clients of this service and service certificate is expired. You should reissue and replace certificate in ALL client keystores! Even more, if keystores are packaged into war files or OSGi bundles – they should be unpackaged and updated. Not really acceptable for enterprise environments.  
  
Therefore large service landscapes have concept of central certificates management. It means that X509 certificates are not stored locally in keystores, but are provided and administrated centrally.   
  
Normally it is a responsibility of [PublicKey Infrastructure](http://en.wikipedia.org/wiki/Public-key_infrastructure" \t "_blank) (PKI) established in organization. PKI is responsible for create, manage, store, distribute, synchronize and revoke public certificates and certification authorities (CAs).  
  
W3C specifies standard SOAP interface to access and administrate keys remotely: [XML Key Management Specification](http://www.w3.org/TR/xkms2/) (XKMS 2.0). XKMS contains two parts:     
a) Key Information Service providing keys locating and validating functionality  
b) Key Registration Service responsible for registration, revocation, recovery and reissuing of keys.  
Note, that beginning from CXF 3.0.0, XKMS service and client implementation as well as XKMS Crypto Provider (see next chapter) will be available in CXF distribution.  
  
**WSS4J Crypto Providers**  
Fine, assume your organization has established simple PKI infrastructure and provides some kind of remote access to look up and validate the certificates (Rest or SOAP XKMS based).   
  
The question now is the following: how to configure my CXF client and service to use public certificates not from local keystore, but from central PKI? Is there any easy way to do it?  
  
Fortunately yes! CXF uses [WSS4J](http://ws.apache.org/wss4j/) for most of security aspects and WSS4J has own concept of crypto providers (don’t mix them with crypto providers of Java Cryptography Architecture).   
  
WSS4J crypto providers are responsible to obtain X509 certificates and private keys, verify X509 trust chain and construct X509 certificates.  
  
WSS4J crypto providers must implement [Crypto](http://ws.apache.org/wss4j/apidocs/org/apache/ws/security/components/crypto/Crypto.html) interface. WSS4J also provides base abstract class with common functionality [CryptoBase](http://ws.apache.org/wss4j/apidocs/org/apache/ws/security/components/crypto/CryptoBase.html" \t "_blank) and keystore based implementation of crypto provider [Merlin](http://ws.apache.org/wss4j/apidocs/org/apache/ws/security/components/crypto/Merlin.html).  
  
If you do not configure anything in CXF, WSS4J uses keystore based Merlin crypto provider by default. But, remember, we would like to get our certificates from PKI instead of keystore. Therefore it will be necessary to create own WSS4J crypto provider for this purpose. Interesting for us are two following methods:

* java.security.cert.X509Certificate[] [getX509Certificates](http://ws.apache.org/wss4j/apidocs/org/apache/ws/security/components/crypto/Merlin.html#getX509Certificates%28org.apache.ws.security.components.crypto.CryptoType%29)([CryptoType](http://ws.apache.org/wss4j/apidocs/org/apache/ws/security/components/crypto/CryptoType.html) cryptoType)
* boolean [verifyTrust](http://ws.apache.org/wss4j/apidocs/org/apache/ws/security/components/crypto/Merlin.html" \l "verifyTrust%28java.security.cert.X509Certificate[],%20boolean%29" \t "_blank)(java.security.cert.X509Certificate[] certs, boolean enableRevocation)

Frist method looks up and returns X509 certificates based on CryptoType identifier. CryptoType specifies certificate identifier as subject DN, issuer DN and serial number, alias, thumbprint or SKI bytes.  
  
We still obtain private keys from keystore, therefore corresponded methods will be delegated to standard Merlin provider. Skeleton implementation of PKI crypto provider can look like:

public class PKICryptoProvider extends CryptoBase {

    private Crypto defaultCrypto;

    private Properties keystoreProps;

    public PKICryptoProvider() {

        try {

            // load keystore properties and alias for private keys

            // …

            defaultCrypto = CryptoFactory.getInstance(keystoreProps);

        } catch (WSSecurityException e) {

            throw new IllegalStateException(

                    "Cannot instantiate default crypto provider: "

                            + e.getMessage(), e);

        }

    }

    @Override

    public X509Certificate[] getX509Certificates(CryptoType cryptoType)

            throws WSSecurityException {

        CryptoType.TYPE type = cryptoType.getType();

        X509Certificate[] certs = new X509Certificate[0];

        switch (type) {

        case SUBJECT\_DN: {

            // get X509 certificate remotely from PKI on the base of subjectDN

            break;

        }

        case ALIAS: {

            // get X509 certificate remotely from PKI on the base of alias

            break;

        }

        case ISSUER\_SERIAL: {

            // get X509 certificate remotely from PKI on the base of issuer DN

            // and serial number

            break;

        }

        default: {

            throw new IllegalArgumentException("Not supported cryptoType: "

                    + cryptoType);

        }

        }

        return certs;

    }

    @Override

    public String getX509Identifier(X509Certificate cert) {

        return cert.getSubjectDN().getName();

    }

    @Override

    public PrivateKey getPrivateKey(X509Certificate certificate,

            CallbackHandler callbackHandler) throws WSSecurityException {

        // …

        return defaultCrypto.getPrivateKey(identifier, password);

    }

    @Override

    public PrivateKey getPrivateKey(String identifier, String password)

            throws WSSecurityException {

        // …

        return defaultCrypto.getPrivateKey(identifier, password);

    }

    @Override

    public boolean verifyTrust(X509Certificate[] certs)

            throws WSSecurityException {

        return verifyTrust(certs, false);

    }

    @Override

    public boolean verifyTrust(X509Certificate[] certs, boolean enableRevocation)

            throws WSSecurityException {

        // call PKI remote method to validate certificate trust chain

        return false;

    }

    @Override

    public boolean verifyTrust(PublicKey publicKey) throws WSSecurityException {

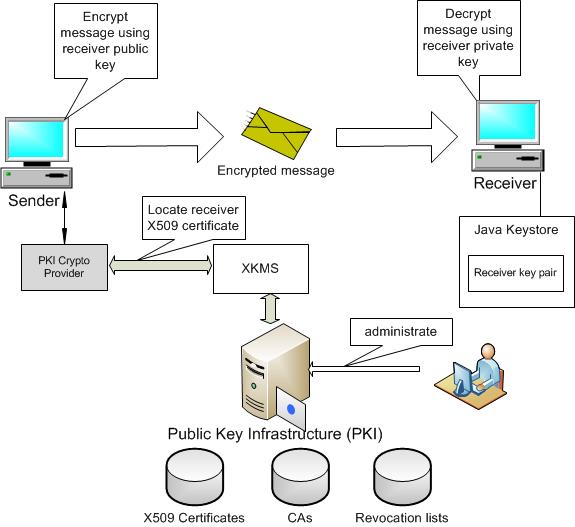
        // call PKI remote method to validate certificate trust chain

        return false;

    }

}

As far as crypto provider makes a lot of remote calls, it makes sense to care about certificate caching.  
  
Using PKI Crypto provider our message encryption picture will change in following way:

[](http://2.bp.blogspot.com/-RGIRM5NfcwQ/UfezzH9AKyI/AAAAAAAAAGg/Jv_9nSZ4kb4/s1600/classic-message-encryption-PKI.jpg)

CXF distributions (starting from version 3.0.0) will contain XKMS based WSS4J Crypto implementation, so you can just reuse it.  
  
**Configure custom WSS4J Crypto Providers**  
How CXF and WSS4J know that they should use our custom crypto provider instead standard Merlin? There are some ways to configure it.  
  
**Keystore properties**  
You can specify own crypto provider in keystore properties file.  
clientKeystore.properties:

org.apache.ws.security.crypto.provider=org.company.security.PKICryptoProvider

org.apache.ws.security.crypto.merlin.keystore.type=jks

org.apache.ws.security.crypto.merlin.keystore.password=secret

org.apache.ws.security.crypto.merlin.keystore.private.password=secret

org.apache.ws.security.crypto.merlin.keystore.alias=myclient

org.apache.ws.security.crypto.merlin.keystore.file=./etc/keystores/clientstore.jks

After it you configure keystore properties for CXF client or service as usual. For example in Spring it will be looks like:

<jaxws:client id="MyClient" xmlns:serviceNamespace="http://services.company.org/MyService"

    serviceClass="org.company.services.MyService" serviceName="serviceNamespace:MyServiceProvider"

    endpointName="serviceNamespace:MyServicePort" address="http://localhost:8080/services/MyService">

    <jaxws:properties>

        <entry key="ws-security.encyption.properties" value="clientKeystore.properties" />

        <entry key="ws-security.signature.properties" value="clientKeystore.properties" />

    </jaxws:properties>

</jaxws:client>

ws-security.encryption.properties says CXF that clientKeystore.properties will be used for the encryption. WSS4J will instantiate and call our PKICryptoProvider automatically. The same keystore properties can be specified for signature as well (ws-security.signature.properties).  
  
**Crypto provider object in Spring configuration**  
Sometimes more convenience way is to instantiate custom crypto provider explicitly. For example, if you want to inject proxy to communicate with PKI into PKICryptoProvider or use non-default constructor. For such cases CXF provides ws-security.encryption.crypto and ws-security.signature.crypto properties. You can pass pre-constructed PKICryptoProvider object using these properties.  
  
Spring configuration in this case looks like:

<bean id="pkiCryptoProvider" class="org.company.security.PKICryptoProvider">

<!—-inject proxies, fields here -->

</bean>

<jaxws:client id="MyClient" xmlns:serviceNamespace="http://services.company.org/MyService"

    serviceClass="org.company.services.MyService" serviceName="serviceNamespace:MyServiceProvider"

    endpointName="serviceNamespace:MyServicePort" address="http://localhost:8080/services/MyService">

    <jaxws:properties>

        …

        <entry key="ws-security.encyption.crypto" value-ref=" pkiCryptoProvider " />

        <entry key="ws-security.signature.crypto" value-ref=" pkiCryptoProvider " />

    </jaxws:properties>

</jaxws:client>

Of course you can use the same approach to configure PKICryptoProvider for CXF service (jaxws:endpoint).  
  
**Specify Crypto provider object programmatically**  
Crypto provider object can be also passed programmatically. Client code will look like:

PKICryptoProvider pkiCryptoProvider = new PKICryptoProvider();

SOAPService ss = new SOAPService(wsdlURL, SERVICE\_NAME);

Greeter port = ss.getSoapPort();

Map<String , Object> requestContext =

    ((javax.xml.ws.BindingProvider)port).getRequestContext();

    requestContext.put(SecurityConstants.ENCRYPTION\_CRYPTO,

    pkiCryptoProvider);

    requestContext.put(SecurityConstants.SIGNATURE\_CRYPTO,

    pkiCryptoProvider);

Remember that this code is not thread safe. To use thread safe request context it is necessary to add following property:

((BindingProvider)proxy).getRequestContext().put("thread.local.request.context", "true");

On the service side you can use this code:

…

@Resource

private WebServiceContext wsContext;

public ResponseData businessMethod(requestData) {

    PKICryptoProvider pkiCryptoProvider = new PKICryptoProvider();

    wsContext.getMessageContext().put(SecurityConstants.ENCRYPTION\_CRYPTO, pkiCryptoProvider);

    wsContext.getMessageContext().put(SecurityConstants.SIGNATURE\_CRYPTO, pkiCryptoProvider);

}

And in interceptor:

public class CustomSecurityInterceptor extends AbstractPhaseInterceptor<Message> {

    public CustomSecurityInterceptor () {

        super(Phase.PRE\_LOGICAL);

    }

    @Override

    public void handleMessage(Message message) throws Fault {

       PKICryptoProvider pkiCryptoProvider = new PKICryptoProvider();

       message.put(SecurityConstants.ENCRYPTION\_CRYPTO, pkiCryptoProvider);

       message.put(SecurityConstants.SIGNATURE\_CRYPTO, pkiCryptoProvider);

}

**Conclusion**  
In this blog we have discussed requirements and use cases to use central certificates management infrastructure, showed how to create custom PKI based crypto provider and configure it in CXF clients/services. PKI crypto provider can be used for communication with Security Token Service (STS) and inside STS to validate X509 certificates as well. But it is topic for the next blog.

Posted by [Andrei Shakirin](https://www.blogger.com/profile/00740530708764404230)at [09:21](http://ashakirin-cxf-security.blogspot.ca/2014/03/cxf-security-getting-certificates-from.html) [No comments:](http://ashakirin-cxf-security.blogspot.ca/2014/03/cxf-security-getting-certificates-from.html#comment-form)

[Email This](https://www.blogger.com/share-post.g?blogID=5743243276808589319&postID=7174226672959819538&target=email)[BlogThis!](https://www.blogger.com/share-post.g?blogID=5743243276808589319&postID=7174226672959819538&target=blog)[Share to Twitter](https://www.blogger.com/share-post.g?blogID=5743243276808589319&postID=7174226672959819538&target=twitter)[Share to Facebook](https://www.blogger.com/share-post.g?blogID=5743243276808589319&postID=7174226672959819538&target=facebook)[Share to Pinterest](https://www.blogger.com/share-post.g?blogID=5743243276808589319&postID=7174226672959819538&target=pinterest)

Labels: [Certificate](http://ashakirin-cxf-security.blogspot.ca/search/label/Certificate), [CXF](http://ashakirin-cxf-security.blogspot.ca/search/label/CXF), [PKI](http://ashakirin-cxf-security.blogspot.ca/search/label/PKI), [public key](http://ashakirin-cxf-security.blogspot.ca/search/label/public%20key), [Security](http://ashakirin-cxf-security.blogspot.ca/search/label/Security), [WSS4J](http://ashakirin-cxf-security.blogspot.ca/search/label/WSS4J), [X509](http://ashakirin-cxf-security.blogspot.ca/search/label/X509), [XKMS](http://ashakirin-cxf-security.blogspot.ca/search/label/XKMS)

[CXF security: integrate PKI to Security Token Service](http://ashakirin-cxf-security.blogspot.ca/2014/03/cxf-security-integrate-pki-to-security.html)

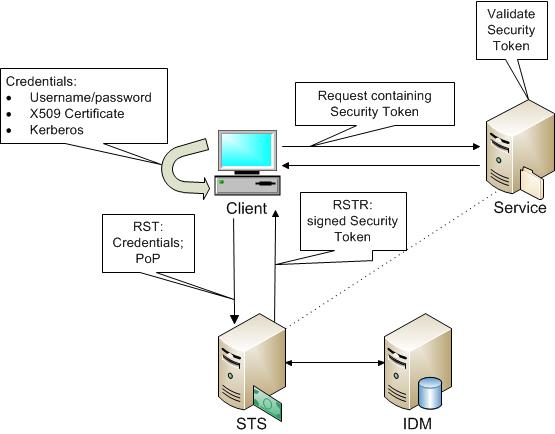
**Introduction**

In previous blog [CXF security: getting certificates from central PKI](http://ashakirin.blogspot.de/2013/04/cxf-security-getting-certificates-from.html) we have seen how to use Public Key Infrastructure and XKMS service to locate certificates in message encryption scenario. This blog is continuation of previous one and explains the integration of central PKI into SecurityTokenService (STS) for the authentication.

**WS-Trust and SecurityTokenService**

In authentication scenarios client generates or obtains security token and sends it to the service for verification inside the message. Security token can be either user name and password, SAML assertion or Kerberos ticket. Generation of security token sometimes is non-trivial task, requires access to external systems and using third party libraries. Therefore it makes a lot of sense to free service participants from implementing any security processing logic on their own. This logic can be delegated to SecurityTokenService (STS) offering functionality to issue, validate, renew or remove Security Tokens.The STS is defined within the OASIS WS-Trust specification.

The communication between client, service and STS is depicted on following figure:

[](http://3.bp.blogspot.com/-dIBxtgnqH34/UfaVcTljeQI/AAAAAAAAAFw/Ss2jRvM_kSg/s1600/STS-classic.jpg)

The typical authentication scenario using WS-Trust requires the following steps:

1. Client obtains credentials and requests security token from STS.
2. STS verifies credentials using external IDM or other mechanism.
3. STS generates security token (for instance SAML), signs it with own private key and sends it back to client.
4. Client injects security token into protocol security header (SOAP or REST HTTP) and sends request message to the service.
5. Service extracts security token and validates it locally or using remote call to STS. As far as service and STS are in trusted relationship, it is enough just to validate STS signature of the security token.

You can find more details about STS service in [CXF WS-Trust documentation](http://cxf.apache.org/docs/ws-trust.html) and [Oli Wulff blog](http://owulff.blogspot.de/2012/02/saml-tokens-and-ws-trust-security-token.html" \t "_blank).  
Typically STS call is transparent for the client and triggered by appropriate WS-Policy assertion (IssuedToken).

**When STS validates user's X509 certificate?**

In some scenarios STS receives user certificate and uses it for authentication and for generation of SAML assertion:

1. Client sends user certificate in STS request as credentials for own authentication. In this case  client must additionally sign part of request with private key to proof of possession:
2. <wsse:Security
3. xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd" xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" soap:mustUnderstand="1">
4. <wsse:BinarySecurityToken EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#Base64Binary"
5. ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3"
6. wsu:Id="X509-3CBA5F87FE6EAABD0E13661135280161">...</wsse:BinarySecurityToken>
7. <wsu:Timestamp wsu:Id="TS-1">
8. <wsu:Created>2013-04-16T11:58:48.003Z</wsu:Created>
9. <wsu:Expires>2013-04-16T12:03:48.003Z</wsu:Expires>
10. </wsu:Timestamp>
11. <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
12. Id="SIG-2">
13. <ds:SignedInfo>
14. ...
15. <ds:Reference URI="#TS-1">
16. ...
17. </ds:Reference>
18. </ds:SignedInfo>
19. <ds:SignatureValue>...</ds:SignatureValue>
20. <ds:KeyInfo Id="KI-3CBA5F87FE6EAABD0E13661135280202">
21. <wsse:SecurityTokenReference
22. wsu:Id="STR-3CBA5F87FE6EAABD0E13661135280223">
23. <wsse:Reference URI="#X509-3CBA5F87FE6EAABD0E13661135280161"
24. ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3" />
25. </wsse:SecurityTokenReference>
26. </ds:KeyInfo>
27. </ds:Signature>
28. </wsse:Security>
29. ds:KeyInfo>
30. </ds:Signature>
31. ...

 You can see that BinarySecurityToken has X509 type and contains user certificate. Timestamp is signed with KeyInfo referencing to this certificate URI="#X509-3CBA5F87FE6EAABD0E13661135280161".

1. Client sends user certificate to be included into SAML token as SubjectConfirmationData/KeyInfo element. This certificate can be referenced and used for validation of XML signatures:
2. <saml2:Assertion xmlns:saml2="urn:oasis:names:tc:SAML:2.0:assertion"
3. xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. ID="\_C5CE27CE46751556E113662938382792" IssueInstant="2013-04-18T14:03:58.279Z"
5. Version="2.0" xsi:type="saml2:AssertionType">
6. ...
7. <saml2:Subject>
8. <saml2:NameID
9. Format="urn:oasis:names:tc:SAML:1.1:nameid-format:unspecified"
10. NameQualifier="http://cxf.apache.org/sts">alice@EXAMPLE.COM</saml2:NameID>
11. <saml2:SubjectConfirmation
12. Method="urn:oasis:names:tc:SAML:2.0:cm:holder-of-key">
13. <saml2:SubjectConfirmationData
14. xsi:type="saml2:KeyInfoConfirmationDataType">
15. <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
16. <ds:X509Data>
17. <ds:X509Certificate>...</ds:X509Certificate>
18. </ds:X509Data>
19. </ds:KeyInfo>
20. </saml2:SubjectConfirmationData>
21. </saml2:SubjectConfirmation>
22. </saml2:Subject>
23. </saml2:Assertion>
24. <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
25. Id="SIG-11">
26. <ds:SignedInfo>
27. ...
28. <ds:KeyInfo Id="KI-CC7D54F0FC1CF34D4913662938383496">
29. <ns3:SecurityTokenReference
30. xmlns:ns3="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
31. xmlns:wsse11="http://docs.oasis-open.org/wss/oasis-wss-wssecurity-secext-1.1.xsd"
32. wsse11:TokenType="http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0">
33. <ns3:KeyIdentifier
34. ValueType="http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLID">\_C5CE27CE46751556E113662938382792
35. </ns3:KeyIdentifier>
36. </ns3:SecurityTokenReference>
37. </ds:KeyInfo>
38. </ds:Signature>
39. ...

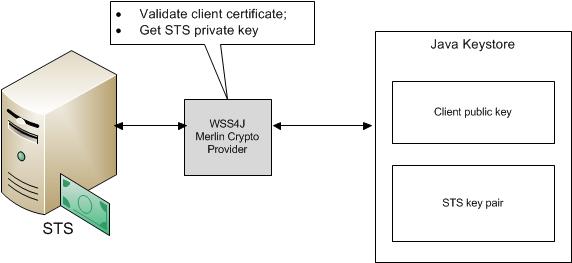
You can see that KeyIdentifier into Signature references SAML token ID="\_C5CE27CE46751556E113662938382792". That means, the certificate from SAML SubjectConfirmation must be used to verify this signature.

In both scenarios STS should validate the user certificate.

**User Certificate Validation**

**Validation through local java keystore**

STS validates client certificate using verifyTrust() method of WSS4J [Crypto](http://ws.apache.org/wss4j/apidocs/org/apache/ws/security/components/crypto/Crypto.html) interface. Default implementation of Crypto interface is [Merlin.](http://ws.apache.org/wss4j/apidocs/org/apache/ws/security/components/crypto/Merlin.html) Merlin provider is keystore based, that means, user certificate will be validated using local STS java keystore. Merlin checks first is user certificate itself in STS keystore. If yes, validation is successfully finished. If no, Merlin proves if keystore contains all trusted chain certificates.

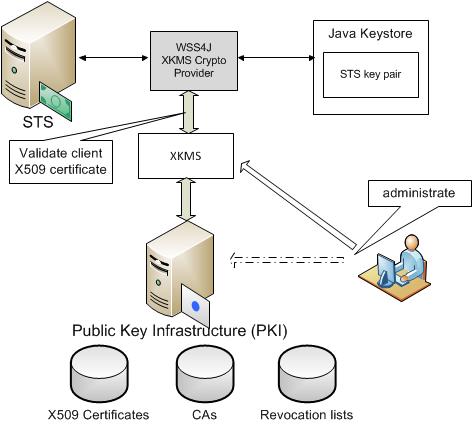
[](http://1.bp.blogspot.com/-7pIus2GUqiE/Ufe3Nn5S-oI/AAAAAAAAAGw/6syxe4Z8DzU/s1600/STS-keystore.jpg)

The problem in this approach is that STS local java keystore should contain either every client certificate or all certificates from certificate trusted chain. Furthermore, administrator should manage revocation lists for all client certificates in STS keystore.

That is not really acceptable for enterprise environments hosting a lot of clients. One possibility to resolve this issue is using Crypto XKMS implementation accessing central Public Key Infrastructure (PKI) to validate user certificate.

**Validation through PKI**

Using of XKMS based Crypto implementation is depicted bellow:

[](http://1.bp.blogspot.com/-l4skuC35EgM/Ufe764bjYRI/AAAAAAAAAHk/WXRn9fKXoF8/s1600/STS-PKI-validation.jpg)

You can see that STS uses XKMS based Crypto implementation instead Merlin. XKMS Crypto still gets STS private key from local java keystore, but it uses central PKI to validate user certificates. W3C specifies standard SOAP interface to access and administrate keys remotely: [XML Key Management Specification](http://www.w3.org/TR/xkms2/) (XKMS 2.0). XKMS Crypto provider invokes XKMS service to validate user certificate. Dependent on XKMS service implementation, it either validates user certificate locally or delegates validation call to remote PKI. Administrator can manage certificates and revocation lists either directly in PKI or through XKMS registration interface (XKRSS). Note, that beginning from CXF 3.0.0, XKMS service and client implementation as well as XKMS Crypto Provider are available in CXF distribution.

**Configure STS to use XKMS Crypto Provider**

How to say STS that it must use XKMS Crypto Provider instead default keystore based one (Merlin)?

The XKMS Crypto Provider should be instantiated and set as property either in StaticSTSProperties object or in STS jaxws:endpoint. That can be done either programmatic or via Spring/Blueprint configuration.  
The finished tutorial source code for STS configured with XKMS crypto provider is prepared on [xkms\_symmetric\_tutorial](https://github.com/ashakirin/cxf.tutorials/tree/master/cxf_sts_xkms_symmetric_tutorial). It is slightly modified code from [Glen Mazza blog](http://www.jroller.com/gmazza/entry/cxf_sts_tutorial).  
  
Sample spring configuration looks like:

 <!-- XKMS configuration -->

    <bean id="xkmsExtensions"

        class="org.apache.cxf.xkms.model.extensions.AdditionalClassesFactory" />

    <jaxws:client xmlns:serviceNamespace="http://www.w3.org/2002/03/xkms#wsdl"

        id="xkmsClient" serviceClass="org.w3.\_2002.\_03.xkms\_wsdl.XKMSPortType"

        serviceName="serviceNamespace:XKMSService" endpointName="serviceNamespace:XKMSPort"

        address="http://localhost:8080/xkms/XKMS/">

        <jaxws:properties>

            <entry key="jaxb.additionalContextClasses">

                <bean class="java.lang.Object" factory-bean="xkmsExtensions"

                    factory-method="create" />

            </entry>

        </jaxws:properties>

    </jaxws:client>

    <bean id="xkmsCryptoProviderFactory" class="org.apache.cxf.xkms.crypto.impl.XkmsCryptoProviderFactory">

        <constructor-arg ref="xkmsClient"/>

    </bean>

    <bean id="xkmsCryptoProvider" class="org.apache.cxf.xkms.crypto.impl.XkmsCryptoProvider"

          factory-bean="xkmsCryptoProviderFactory"

          factory-method="create">

          <constructor-arg type="java.lang.String" value="/stsKeystore.properties" />

    </bean>

 <!-- end of XKMS configuration -->

  <bean id="stsProperties" class="org.apache.cxf.sts.StaticSTSProperties">

        <property name="signatureUsername" value="sts" />

        <property name="signatureCrypto" ref="xkmsCryptoProvider" />

        <property name="callbackHandler" ref="pwdCallbackHandler" />

        <property name="encryptionUsername" value="useReqSigCert" />

        <property name="encryptionCrypto" ref="xkmsCryptoProvider" />

        <property name="issuer" value="STS Issuer" />

  </bean>

  <bean id="transportSTSProvider" class="org.sopera.csg.tesbext.sts.factory.ProviderFactory" factory-method="create">

        <argument ref="stsProperties" />

  </bean>

  <jaxws:endpoint id="transportSTS"

        implementor="#transportSTSProvider"

        address="/SecurityTokenService/Transport"

        wsdlLocation="wsdl/ws-trust-1.4-service.wsdl"

        xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"

        serviceName="wst:SecurityTokenService"

        endpointName="wst:Transport\_Port">

  </jaxws:endpoint>

jaxws:client is configuration of XKMS client, it is used as constructor argument of xkmsCryptoProviderFactory. Argument "/stsKeystore.properties" ofxkmsCryptoProvider is necessary to create default keystore based crypto provider. It is responsible to get STS private key from the local keystore. Lookup and validation of the public keys (X509) will be delegated to XKMS crypto provider and XKMS service.

Alternatively XKMS Crypto Provider can be set directly into STS jaxws:endpoint:

...

<jaxws:endpoint id="transportSTS"  
        implementor="#transportSTSProvider"  
        address="/SecurityTokenService/Transport"  
        wsdlLocation="wsdl/ws-trust-1.4-service.wsdl"  
        xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"  
        serviceName="wst:SecurityTokenService"  
        endpointName="wst:Transport\_Port">  
        <jaxws:properties>

          ...

          <entry key="ws-security.signature.crypto" value-ref="xkmsCryptoProvider"/>

          <entry key="ws-security.encryption.crypto" value-ref="xkmsCryptoProvider"/>  
        </jaxws:properties>

</jaxws:endpoint>

**Using XKMS for STS scenario with Symmetric Key**

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

|  |
| --- |
|  |
| STS scenario with SymmetricKey (used from [SAP resource](http://help.sap.com/saphelp_nw70ehp2/helpdata/en/e5/9f9913fc9c418db98c8693b2bbdb7c/content.htm)) |  |  |  |

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. The STS signs SAML assertion and sends it together with key material for generation symmetric key to the WS-Client.
4. The WS-Client generates short-lived symmetric key from own material and the key material from the STS.
5. 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.
6. 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.
7. 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. 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*](https://draft.blogger.com/null)" and service endpoint as identifier)

Spring configuration looks like:

    <!-- XKMS configuration -->

    <bean id="xkmsExtensions"

        class="org.apache.cxf.xkms.model.extensions.AdditionalClassesFactory" />

    <jaxws:client xmlns:serviceNamespace="http://www.w3.org/2002/03/xkms#wsdl"

        id="xkmsClient" serviceClass="org.w3.\_2002.\_03.xkms\_wsdl.XKMSPortType"

        serviceName="serviceNamespace:XKMSService" endpointName="serviceNamespace:XKMSPort"

        address="http://localhost:8080/xkms/XKMS/">

        <jaxws:properties>

            <entry key="jaxb.additionalContextClasses">

                <bean class="java.lang.Object" factory-bean="xkmsExtensions"

                    factory-method="create" />

            </entry>

        </jaxws:properties>

    </jaxws:client>

    <bean id="xkmsCryptoProviderFactory" class="org.apache.cxf.xkms.crypto.impl.XkmsCryptoProviderFactory">

        <constructor-arg ref="xkmsClient"/>

    </bean>

    <bean id="xkmsCryptoProvider" class="org.apache.cxf.xkms.crypto.impl.XkmsCryptoProvider"

          factory-bean="xkmsCryptoProviderFactory"

          factory-method="create">

          <constructor-arg type="java.lang.String" value="/stsKeystore.properties" />

    </bean>

    <!-- end of XKMS configuration -->

    <bean id="mySTSProviderBean"

        class="org.apache.cxf.sts.provider.DefaultSecurityTokenServiceProvider">

        <property name="stsProperties" ref="mySTSProperties" />

        <property name="services" ref="myServiceList" />

    </bean>

    <bean id="myServiceList" class="org.apache.cxf.sts.service.StaticService">

        <property name="endpoints" ref="wspAllowedEndpoints" />

    </bean>

    <util:list id="wspAllowedEndpoints">

        <value>http://localhost:8080/doubleit/services/doubleit.\*</value>

    </util:list>

    <bean id="mySTSProperties" class="org.apache.cxf.sts.StaticSTSProperties">         

        <property name="signatureUsername" value="mystskey" />

        <property name="callbackHandlerClass" value="sts.PasswordCallbackHandler" />        

        <property name="issuer" value="DoubleItSTSIssuer" />

        <property name="encryptionUsername" value="useEndpointAsCertAlias"/>

        <property name="encryptionCrypto" ref="xkmsCryptoProvider" />

        <property name="signatureCrypto" ref="xkmsCryptoProvider" />

    </bean>

    <jaxws:endpoint id="CXFSTS" implementor="#mySTSProviderBean"

        address="/STS" wsdlLocation="/WEB-INF/wsdl/DoubleItSTSService.wsdl"

        xmlns:ns1="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"

        serviceName="ns1:SecurityTokenService" endpointName="ns1:STS\_Port">

        <jaxws:properties>

            <entry key="ws-security.callback-handler" value="sts.PasswordCallbackHandler" />

            <entry key="ws-security.signature.properties" value="stsKeystore.properties" />

            <entry key="ws-security.signature.username" value="mystskey" />

            <!-- Below unused/unneeded if using UT auth between WSC and STS -->

            <entry key="ws-security.encryption.username" value="useReqSigCert" />          

        </jaxws:properties>

    </jaxws:endpoint>

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.

**Conclusion**

In this blog we have discussed use case and possible solution to use central certificates management infrastructure inside Security Token Service (STS) to validate and locate X509 certificates.