CXF and WS-SecurityPolicy

Status: Version 1.2

Version: 17.07.2018

[Changelog 4](#_Toc519590792)

[1 Introduction 5](#_Toc519590793)

[1.1 Intended audience 5](#_Toc519590794)

[1.2 Prerequisites 5](#_Toc519590795)

[1.3 Apache CXF Version 5](#_Toc519590796)

[1.4 Disclaimer 5](#_Toc519590797)

[2 Building a Web Service Provider 6](#_Toc519590798)

[2.1 Hello World Service 6](#_Toc519590799)

[2.2 Reference Code 6](#_Toc519590800)

[2.3 Design Choices 6](#_Toc519590801)

[2.4 Security Requirements 7](#_Toc519590802)

[2.5 WS-SecurityPolicy Configuration 7](#_Toc519590803)

[2.5.1 The General Security Policy 7](#_Toc519590804)

[2.5.2 The Input Policy 8](#_Toc519590805)

[2.5.3 The Output Policy 9](#_Toc519590806)

[2.6 Configuring Keystores for the Service 9](#_Toc519590807)

[2.7 Support Classes 10](#_Toc519590808)

[2.7.1 Dealing with custom headers 10](#_Toc519590809)

[2.7.2 Pulling data from incoming token and storing for later access 12](#_Toc519590810)

[2.8 Additional Configuration of Apache CXF 15](#_Toc519590811)

[2.8.1 Validating Audience Restriction 15](#_Toc519590812)

[2.8.2 Configure TimeToLive 16](#_Toc519590813)

[2.8.3 Configure BSP 1.1 Compliance 16](#_Toc519590814)

[2.9 The bearer-token version of the service 17](#_Toc519590815)

[2.9.1 Modify the WS-SecurityPolicy 17](#_Toc519590816)

[2.9.2 Modify the truststore 18](#_Toc519590817)

[2.10 Other Considerations 18](#_Toc519590818)

[2.10.1 Replay detection 18](#_Toc519590819)

[2.10.2 WS-SecureConversation 18](#_Toc519590820)

[3 Building a Web Service Consumer 19](#_Toc519590821)

[3.1 Reference Code 19](#_Toc519590822)

[3.2 Design Choices 19](#_Toc519590823)

[3.3 Security Requirements 19](#_Toc519590824)

[3.4 WS-SecurityPolicy 19](#_Toc519590825)

[3.4.1 Configuring WS-SecurityPolicy for the STS 19](#_Toc519590826)

[3.4.2 Configuring WS-SecurityPolicy for the WSP 20](#_Toc519590827)

[3.5 Configuring Keystores for the WSC 20](#_Toc519590828)

[3.6 The STSClient Configuration and Implementation 21](#_Toc519590829)

[3.7 Support Classes 22](#_Toc519590830)

[3.8 Additional Configuration of Apache CXF 23](#_Toc519590831)

[3.8.1 Configure BSP 1.1 Compliance 23](#_Toc519590832)

[3.9 Other Considerations 23](#_Toc519590833)

[4 Typical Errors 24](#_Toc519590834)

[4.1 Java Strong Crypto Not Installed 24](#_Toc519590835)

[4.2 A .keystore File in Home Folder 24](#_Toc519590836)

[5 Summary 25](#_Toc519590837)

[6 References 26](#_Toc519590838)

# Changelog

17-07-2018 Initial release

# Introduction

This document is a technical appendix to the Java reference source code, that showcases how to use the Apache CXF framework to implement various identity based webservices, following either the Liberty Basic SOAP Binding profile, the OIO IDWS Profile or the XUA variant of the OIO IDWS Profile.

This document is derived from the original documentation for the NemLog-in2 reference implementation, but as there now exist several implemented profiles, the CXF and WS-SecurityPolicy specific parts of the implementation has been moved into this separate appendix, and the profile-specific documentation is left in the main documents.

In this document it is assumed that the reader already possesses some background knowledge about federated security, and knows what Web Service Consumers (WSC), Web Service Providers (WSP), Service Providers (SP), Identity Providers (IdP) and Security Token Services (STS) are, and how they correspond to each other. Furthermore, it is assumed that the reader is familiar with the concepts of holder-of-key and bearer-tokens.

## Intended audience

This document is written for developers, and while all configuration and customization of Apache CXF concerning security is dealt with, some experience with Apache CXF or a similar web service framework is recommended. The reader is also expected to have experience with Java development in general.

## Prerequisites

The source code uses Apache Maven 3 [MAVEN] as a build tool, and the source code requires at least Java 7 with Strong Crypto [CRYPTO] to compile and run. The reader is expected to have these tools available before using the reference source code. Maven handles all other dependencies.

## Apache CXF Version

The reference code is based on Apache CXF 3.0.16, but the same concepts outlined in this document can be applied to the 3.1.x and 3.2.x branches of CXF.

## Disclaimer

The Danish Agency for Digitisation provides the reference code as is and assumes no responsibility for the code by service providers. Service Providers should understand the limitations of the code and deal with these according to their own needs.

# Building a Web Service Provider

This chapter covers all the steps necessary to secure a web service using Apache CXF.

The reference source code is split into several subfolders, one for each profile. There exists a set of WSP reference implementations for each of these profiles. The approach to building them is identical, and there are only minor differences in the actual source/configuration.

Currently the following profiles are available

* **“lbsb-scenarios”**. This folder shows how to implement the Liberty Basic SOAP Binding using CXF and WS-SecurityPolicy
* **“oioidws-scenarios”**. This folder shows how to implement the OIO IDWS 1.1 profile using CXF and WS-SecurityPolicy
* **“oioidws-xua-scenarios”**. This folder shows how to implement the XUA variant of the OIO IDWS 1.1 profile using CXF and WS-SecurityPolicy

## Hello World Service

The reference source code is based on a very simple web service called HelloWorld, and all parts of the code are clearly packaged and commented, so those sections relevant to dealing with security are easily identified.

Please note that the documentation covers two different versions of the same service. The first service accepts holder-of-key tokens and the second accepts bearer-tokens. The documentation will outline the difference in configuration and implementation between these two variants.

The HelloWorld service has a single operation, that takes a text-string as input, and returns the corresponding “Hello [input]” text-string.

## Reference Code

The code for the Hello World service is found in the two folders “service-hok” and “service-bearer” found in each of the profile folders in the reference code distribution.

The “service-hok” folder contains the code and configuration for the version of the service that accepts holder-of-key tokens, and the “service-bearer” folder contains the code and configuration for the version of the service that accepts bearer-tokens.

The following chapters cover the steps necessary to build the holder-of-key version of the service, followed by a chapter that covers the set of modifications required to use bearer-tokens instead of holder-of-key tokens.

## Design Choices

In the reference source for the web service, a WSDL-first design approach has been chosen, as this is the recommended approach to building web services using Apache CXF.

XML-based configuration has been chosen over code-based configuration, which ensures separation between code and configuration. Apache CXF depends on Spring for XML-based configuration.

## Security Requirements

The service must be configured to enforce validation of the client request according to the chosen security profile (e.g. OIO IDWS 1.1).

Using Apache CXF this can be accomplished in two ways. The recommended way is to use WS-SecurityPolicy [WS-SEC-POL], and the alternate way is to use WSS4J Interceptors. We will use the recommended approach, and configure the service using WS-SecurityPolicy.

The service must also be configured to trust at least one STS, which requires a keystore containing the STS certificate(s) to be configured as a truststore.

## WS-SecurityPolicy Configuration

WS-SecurityPolicy is a contract-based approach for dealing with security requirements. The WSDL of the service is extended with a wsp:Policy section, that describes the exact security policy that both the client and service must follow.

This document is not a manual for using WS-SecurityPolicy, and it will only focus on describing the solution used in the reference source – for a more detailed description of WS-SecurityPolicy, please consult the standard [WS-SEC-POL].

A security policy is split into three parts

* A general security policy that describes the tokens used as well as the processing and validation rules.
* An input policy that describes the signing and encryption requirements that the client must follow when generating requests
* An output policy that describes the signing and encryption requirements that the service must follow when generating responses

### The General Security Policy

In all the profiles covered by the reference code, an AsymmetricBinding policy is used. This type of policy covers the use-cases where the communicating parties have access to some asymmetric secret (e.g. certificates). The policy specifies how these asymmetric secrets are to be used.

Besides the specification of the asymmetric binding, the policy can contain some generic settings (e.g. specify that WS-Adressing MUST be used), as well as information about any tokens that are to be used.

The context in which the policy section is placed in the WSDL can be found in the file HelloWorld-Hok.wsdl (or HelloWorld-Bearer.wsdl) found in the src/main/resources folder of the service projects.

If we look at the policies used in the reference code, they contain these three sections

<wsam:Addressing />

<sp:AsymmetricBinding />

<sp:[Signed]SupportingTokens />

The first element enables WS-Addressing and should be set on profiles that require WS-Addressing (and can be set on profiles that do not explicitly prohibit WS-Addressing). Setting this will require the WSC to send WS-Addressing headers in any request sent to the WSP.

The second element specifies the request/response tokens (the asymmetric secrets) to be used by the WSC and WSP, as well as a series of configuration settings that apply to this usage.

Below are listed the elements that are used in the profiles implemented in the reference code. There are more settings available, please see [WS-SEC-POL] for details.

<sp:InitiatorToken />

<sp:RecipientToken />

<sp:AlgorithmSuite />

<sp:Layout />

<sp:ProtectTokens />

<sp:IncludeTimestamp />

<sp:OnlySignEntireHeadersAndBody />

The InitiatorToken and RecipientToken element specify the type of token that the WSC and WSP must use. The InitiatorToken specifies the type of key-material used for request signature and response encryption, and similarly, the RecipientToken element specify the type of key-material used for response signature and request encryption.

Note that the term “token” covers both actual SAML tokens as well as certificates. The elements also specify whether the WSC/WSP should just use the token, or the token must also be included in the message sent to the other party.

In general, if the message-receiver needs the token to validate the message, the policy should specify that the token must be included. Note that the various profiles usually govern exactly when a token must be included in the message.

The AlgorithmSuite element describes which cryptographic algorithms the client and service must use when signing and encrypting data. Useful settings are (see [WS-SEC-POL] for a full list)

* **Basic256**. RSA-SHA1 for signatures, AES-256 for encryption.
* **Basic256Sha256**. RSA-SHA256 for signatures, AES-256 for encryption.

The Layout element describes how header elements must be added to the Security header. This value is usually set to Strict, but Lax is allowed by some profiles.

Adding the ProtectTokens element, requires that the Token Protection rules are followed by the WSC and WSP. The Token Protection rules states that any token used for generating signatures, must also be covered by the generated signature.

The IncludeTimestamp element requires that both the WSC and the WSP must add a timestamp header to the messages being send.

Finally, the OnlySignEntireHeadersAndBody element requires that the WSC and WSP must sign entire header elements and the entire body (if required by the input/output policy) and cannot sign only a part of a header or the body.

The third element describes the supporting tokens (the token supplied by the STS) that the WSC must use when calling the service. Note that either a SignedSupportingToken or SupportingToken section can be used. SupportingToken(s) can be used in the Bearer case, otherwise a SignedSupportingToken is used. This section can be used to specify KeyType, TokenType and other attributes for the supporting token.

### The Input Policy

The input policy specifies which parts of the request generated by the WSC must be signed and which must be encrypted. The policy can reference specific headers as well as the body element, and specify individual rules for each header, as well as for the body element.

In the WSDL, the input policy can then be applied to the individual operations as needed.

The basic structure of the input policy contains a SignedParts and an EncryptedParts element, each indicating which elements should be signed or encrypted,

<sp:SignedParts />

<sp:EncryptedParts />

The different policies implemented in the reference code each have rules regarding which fields are to be signed and which (if any) that need to be encrypted.

### The Output Policy

The output policy is similar to the input policy, with the sole difference that it specifies the rules for the response generated by the WSP.

## Configuring Keystores for the Service

The service must be configured with 2 keystores. One containing the service’s own certificate and corresponding private key and another containing only the certificate(s) of the trusted STSs. The latter is referred to as a truststore.

Both JKS and PKCS12 keystore formats are supported by Apache CXF.

Apache CXF must be configured to use these keystores, which is done by creating a property file with the relevant information, and then pointing CXF towards this property file.

The property file uses the following settings – the values are examples taken from the reference code.

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

org.apache.ws.security.crypto.merlin.keystore.password**=**Test1234

org.apache.ws.security.crypto.merlin.keystore.alias**=**server

org.apache.ws.security.crypto.merlin.file**=**service.jks

org.apache.ws.security.crypto.merlin.truststore.type**=**jks

org.apache.ws.security.crypto.merlin.truststore.file**=**trust.jks

org.apache.ws.security.crypto.merlin.truststore.password**=**Test1234

CXF must know about this property file, so it should be referenced from the main CXF configuration file – in the WSP this is called cxf-servlet.xml and is found under the webapp/WEB-INF folder.

The “ws-security.signature.properties” property found I the cxf-servlet.xml file can be used to point to the keystore configuration file, as shown below.

<jaxws:endpoint>

<jaxws:properties>

<entry key=**"ws-security.callback-handler"**

value=**"service.callback.KeystorePasswordCallback"** />

<entry key=**"ws-security.signature.properties"**

value=**"serviceKeystore.properties"** />

</jaxws:properties>

</jaxws:endpoint>

Note the reference to a callback-handler (ws-security.callback-handler). This is a reference to a Java class whose sole purpose is to supply the password to the server keystore. The Apache CXF framework uses the keystore in various places, and sometimes it looks up the password directly in the property file, and other times it requires that the password is supplied through a callback handler. Hence the keystore password needs to be supplied both in the property file and the callback handler.

Below is shown the source from the callback handler in the reference code – in a real implementation, the password should not be hardcoded, but rather read from some external configuration file.

public class KeystorePasswordCallback **implements** CallbackHandler **{**

@Override

public void handle**(**Callback**[]** callbacks**)** **throws** IOException**,** UnsupportedCallbackException **{**

**for** **(**int i **=** 0**;** i **<** callbacks**.**length**;** i**++)** **{**

WSPasswordCallback pc **=** **(**WSPasswordCallback**)** callbacks**[**i**];**

int usage **=** pc**.**getUsage**();**

**if** **(**usage **==** WSPasswordCallback**.**DECRYPT **||**

usage **==** WSPasswordCallback**.**SIGNATURE**)** **{**

pc**.**setPassword**(**"Test1234"**);**

**}**

**}**

**}**

**}**

Once the property file and callback handler have been implemented, and the XML configuration updated, no further configuration is needed to ensure Apache CXF has access to the keystores.

## Support Classes

Apache CXF can deal with the standard cryptographic operations and SOAP headers like WS-Addressing, but custom headers, custom attributes or anything else that is not governed by the framework, requires additional code.

Some examples of this are shown below

### Dealing with custom headers

Some profiles might require specific SOAP headers to be sent by the WSC to the WSP, and if these headers contain information that needs to be parsed (or they simply have a mustUnderstand=”1” attribute set), then custom handlers must be implemented to deal with this.

If the WSC sends a header that the WSP must understand, an interceptor must be implemented that can deal with this header. The getUnderstoodHeaders() method should return the headers that the WSP can understand (the CXF framework will reject any incoming messages that contains headers with mustUnderstand=”1” that it does not understand).

The example below flags the Framework header from the Liberty Basic SOAP Binding profile as a header it understands.

public class UnderstandFrameworkHeaderInterceptor **extends** AbstractSoapInterceptor **{**

private static final QName frameworkQName **=** **new**

QName**(**"urn:liberty:sb"**,** "Framework"**,** "sbf"**);**

public UnderstandFrameworkHeaderInterceptor**()** **{**

**super(**Phase**.**PRE\_PROTOCOL**);**

**}**

@Override

public void handleMessage**(**SoapMessage message**)** **throws** Fault **{**

Header framework **=** message**.**getHeader**(**frameworkQName**);**

**if** **(**framework **==** **null)** **{**

**throw** **new** XMLFault**(**"Missing framework header"**);**

**}**

**}**

@Override

public Set**<**QName**>** getUnderstoodHeaders**()** **{**

Set**<**QName**>** set **=** **new** HashSet**<>();**

set**.**add**(**frameworkQName**);**

**return** set**;**

**}**

**}**

The interceptor must be added in the cxf-servlet.xml configuration file, so CXF knows to use it. This is done by adding the interceptor as a configured bean, and then adding it to the bus element as shown below

<cxf:bus>

<cxf:inInterceptors>

<ref bean=**"UnderstandFrameworkHeaderInterceptor"** />

</cxf:inInterceptors>

</cxf:bus>

<bean id=**"UnderstandFrameworkHeaderInterceptor"**

class=**"service.interceptor.UnderstandFrameworkHeaderInterceptor"** />

If the WSP must respond with custom headers, then a similar interceptor must be implemented and configured in cxf-servlet.xml.

First generate a JAXB representation of the header, using the @Xml annotations in the Java language. This class can then be used to serialize and deserialize the XML headers.

An example of such a header representation is shown below – in this case the Framework Header required by the Liberty Basic SOAP Binding profile.

@XmlAccessorType**(**XmlAccessType**.**FIELD**)**

public class SbfFrameworkHeader **{**

@XmlAttribute**(**name**=**"version"**)**

private final String version **=** "2.0"**;**

@XmlAttribute**(**name**=**"profile"**,** namespace**=**"urn:liberty:sb:profile"**)**

private final String profile **=** "urn:liberty:sb:profile:basic"**;**

public String getVersion**()** **{**

**return** version**;**

**}**

public String getProfile**()** **{**

**return** profile**;**

**}**

**}**

An interceptor that deals with this specific header should then be implemented, where the handleMessage() method generates the required header and adds it to the output

public class FrameworkHeaderInterceptor **extends** AbstractSoapInterceptor **{**

public FrameworkHeaderInterceptor**()** **{**

**super(**Phase**.**PRE\_PROTOCOL**);**

**}**

@Override

public void handleMessage**(**SoapMessage message**)** **throws** Fault **{**

List**<**Header**>** headers **=** message**.**getHeaders**();**

**try** **{**

Header framework **=** **new** SoapHeader**(**

**new** QName**(**"urn:liberty:sb"**,** "Framework"**,** "sbf"**),**

**new** SbfFrameworkHeader**(),**

**new** JAXBDataBinding**(**SbfFrameworkHeader**.**class**));**

headers**.**add**(**framework**);**

**}**

**catch** **(**Exception ex**)** **{**

**throw** **new** XMLFault**(**ex**.**getMessage**());**

**}**

message**.**put**(**Header**.**HEADER\_LIST**,** headers**);**

**}**

**}**

Finally, we need to configure this interceptor, which is done in the cxf-servlet.xml file. The relevant part of the XML configuration file is shown below – note that the header can be configured to be added to both regular responses, as well as fault responses.

<cxf:bus>

<cxf:outInterceptors>

<ref bean=**"FrameworkHeaderInterceptor"** />

</cxf:outInterceptors>

<cxf:outFaultInterceptors>

<ref bean=**"FrameworkHeaderInterceptor"** />

</cxf:outFaultInterceptors>

</cxf:bus>

<bean id=**"FrameworkHeaderInterceptor"**

class=**"service.interceptor.FrameworkHeaderInterceptor"**/>

### Pulling data from incoming token and storing for later access

When the WSP receives a request from the WSC, CXF will validate the security of incoming request, up to and including the validity of the token.

What CXF will not do, is look in the attributes of the token, and make authorization decision based on these attributes.

If the supplied token is used for more than just access=yes/no, and contains additional information that the WSP needs to make authorization decisions, then the incoming token should be parsed, and the relevant information made available to the parts of the WSP code that makes these decisions.

The approach outlined below, parses the incoming token, and pulls the relevant information into a ThreadLocal, making the information globally accessible in the WSP code. Depending on the type of security framework used, it might make better sense to push the information into whatever security framework is used – the general concept outline below can be adapted to such a purpose, but that is out of scope for this document.

To implement this functionality, three support classes are needed. A class to hold the ThreadLocal (PriviligeHolder.java), a class to clean up the ThreadLocal once the request has been processed (SamlTokenFilter.java) and finally a class to parse the attribute in the token (SamlTokenValidator.java).

The first class is an ordinary ThreadLocal holder, which stores instances of the class PrivilgeListType (a class used to represent an OIO-BPP set of privilege issued by the NemLog-in STS). This holder can be adapted to store whatever information is relevant about the WSC.

public class PriviligeHolder **{**

private static final ThreadLocal**<**PrivilegeListType**>** privileges **=** **new** ThreadLocal**<>();**

public static void set**(**PrivilegeListType privilege**)** **{**

privileges**.**set**(**privilege**);**

**}**

public static PrivilegeListType get**()** **{**

**return** privileges**.**get**();**

**}**

public static void clear**()** **{**

privileges**.**remove**();**

**}**

**}**

The second class is a very simple servlet filter, whose sole purpose is to ensure that the above ThreadLocal holder is cleared once the request has been processed. It is shown below for completeness:

public class SamlTokenFilter **implements** Filter **{**

@Override

public void init**(**FilterConfig filterConfig**)**

**throws** ServletException **{** **}**

@Override

public void doFilter**(**ServletRequest request**,**

ServletResponse response**,**

FilterChain chain**)**

**throws** IOException**,** ServletException **{**

**try** **{**

chain**.**doFilter**(**request**,** response**);**

**}**

**finally** **{**

PriviligeHolder**.**clear**();**

**}**

**}**

@Override

public void destroy**()** **{** **}**

**}**

As with all servlet filters, it must be configured to intercept requests to the application. As the reference source code is packaged as a WAR file, this is done in the web.xml file in the ordinary way. The relevant parts of the web.xml file are shown below for completeness.

<filter>

<filter-name>**SamlTokenFilter**</filter-name>

<filter-class>**service.saml.SamlTokenFilter**</filter-class>

</filter>

<filter-mapping>

<filter-name>**SamlTokenFilter**</filter-name>

<url-pattern>**\***</url-pattern>

</filter-mapping>

The final class extends the CXF SamlAssertionValidator class, and must implement the validate() method – it receives the incoming SAML token after CXF is done parsing it, and we can then use the validate() method to pull any relevant information from the SAML token. In the example below, the code attempts to parse the Privileges attribute in the SAML token (the attribute used by the NemLog-in STS to issue roles to a WSC).

public class SamlTokenValidator **extends** SamlAssertionValidator **{**

@SuppressWarnings**(**"unchecked"**)**

@Override

public Credential validate**(**Credential credential**,**

RequestData data**)**

**throws** WSSecurityException **{**

Credential vCredential **=** **super.**validate**(**credential**,** data**);**

SamlAssertionWrapper samlAssertion **=**

credential**.**getSamlAssertion**();**

**if** **(**samlAssertion**.**getSaml2**()** **!=** **null)** **{**

Assertion saml2 **=** samlAssertion**.**getSaml2**();**

**for** **(**AttributeStatement attributeStatement **:**

saml2**.**getAttributeStatements**())** **{**

**for** **(**Attribute attribute **:**

attributeStatement**.**getAttributes**())** **{**

**if** **(**"Privileges"**.**equals**(**

attribute**.**getFriendlyName**()))** **{**

**for** **(**XMLObject attributeValue **:**

attribute**.**getAttributeValues**())** **{**

**if** **(!**attributeValue.isNil()) **{**

String privilege **=**

attributeValue**.**getDOM**().**getTextContent**();**

byte**[]** privilegeBytes **=**

Base64**.**decodeBase64**(**privilege**);**

**try** **{**

JAXBContext context **=**

JAXBContext**.**newInstance**(**ObjectFactory**.**class**);**

Unmarshaller unmarsheller **=**

context**.**createUnmarshaller**();**

JAXBElement**<**PrivilegeListType**>** privilegeList **=** **(**JAXBElement**<**PrivilegeListType**>)** unmarsheller**.**unmarshal**(new** ByteArrayInputStream**(**privilegeBytes**));**

PriviligeHolder**.**set**(**

privilegeList**.**getValue**());**

**}**

**catch** **(**Exception ex**)** **{**

**throw** **new** WSSecurityException**(**

WSSecurityException**.**ErrorCode**.**FAILURE**,** "invalidSAMLsecurity"**,** ex**);**

**}**

**}**

**}**

**}**

**}**

**}**

**}**

**return** vCredential**;**

**}**

**}**

The SamlTokenValidator class must be configured to be used by the Apache CXF framework. This is done in the cxf-servlet.xml configuration file. The relevant parts of the configuration file are shown below

<jaxws:endpoint>

<jaxws:properties>

<entry key=**"ws-security.saml2.validator"**

value=**"service.saml.SamlTokenValidator"** />

</jaxws:properties>

</jaxws:endpoint>

At any given point in the WSP code, the PriviligeHolders get() method can then be used to access the stored information about the WSC.

## Additional Configuration of Apache CXF

There are a few more configuration steps that are needed before the service is ready to be deployed. The Audience Restriction of the token must be properly validated, it should be validated that the clients request is no older than say 5 minutes, and finally we may need to configure Apache CXF to accept that the token issued by the STS is not Basic Security Profile (BSP) 1.1 compliant.

### Validating Audience Restriction

The token issued by the STS contains an element, called the Audience, that tells which service this token is intended for (an EntityId value). This element must be validated, to ensure that the client (or some other party) is not using a token intended for another service. By default, the Apache CXF framework will validate the value found in the token against the hostname that the service is deployed on as well as the QName of the service. As this is not always the EntityId that the service is registered under, the reference code contains code that adds an additional EntityId to the list that the token’s Audience is validated against.

The SamlTokenValidator found above can be used for performing this validation. We need a List of allowed audiences, so for instance, if we allow tokens issued to <https://wsp.itcrew.dk> to be used, then the code below could be used to create such a list.

private List**<**String**>** audienceRestrictions **=** **new** ArrayList**<**String**>()** **{**

private static final long serialVersionUID **=** 1L**;**

**{**

add**(**"https://wsp.itcrew.dk"**);**

**}**

**};**

Then before we call super.validate() in our own class’s validate() method, we set this list as the allowed set of audiences, overwriting CXFs default validation.

@SuppressWarnings**(**"unchecked"**)**

@Override

public Credential validate**(**Credential credential**,** RequestData data**)** **throws** WSSecurityException **{**

// Set the valid audiences for this request

data**.**setAudienceRestrictions**(**audienceRestrictions**);**

Credential validatedCredential **=** **super.**validate**(**credential**,** data**);**

SamlAssertionWrapper samlAssertion **= ...**

### Configure TimeToLive

By default, the Apache CXF framework will not accept messages that are older than 5 minutes, but if a different time period is required, this can be configured in the cxf-servlet.xml configuration file by setting the following property (the value is in seconds, so 300 = 5 minutes)

<jaxws:endpoint>

<jaxws:properties>

<entry key=**"ws-security.timestamp.timeToLive"** value=**"300"** />

</jaxws:properties>

</jaxws:endpoint>

### Configure BSP 1.1 Compliance

The Apache CXF framework enforces Basic Security Profile 1.1 compliance by default, and as CXF does not have full support for encrypted tokens, this causes some validation errors to occur during BSP 1.1 compliance checking if such are used. If needed, BSP validation can be disabled; this is done in the cxf-servlet.xml configuration file, and the relevant settings are shown below.

<jaxws:endpoint>

<jaxws:properties>

<entry key=**"ws-security.is-bsp-compliant"** value=**"false"** />

</jaxws:properties>

</jaxws:endpoint>

## The bearer-token version of the service

A bearer token simply grants the WSC the identity and privileges indicated by the token, without requiring the WSC to prove that it is the rightful bearer of the token (possession of the token is enough).

The implementation notes above all cover the case for implementing a WSP that accepts holder-of-key tokens, but the changes required to accept bearer tokens are minimal and are covered in this section.

Before making the changes, consider whether the WSC should still sign the message, or is the bearer token enough on its own? The WSC can still sign the message (allowing for message integrity and non-repudiation), but the supplied token is not used in the signature validation process, so the identity supplied by the token is not associated with the signature validation process (which is considered a feature in some cases).

This choice depends on the implementation profile, and for instance the Liberty Basic SOAP Binding profile requires a signature when using bearer tokens.

Assuming the previous steps in this chapter have been implemented, the following modifications are required to implement bearer tokens.

### Modify the WS-SecurityPolicy

As the WSC calling the WSP only supplies a bearer-token, we need to ensure that the client supplies its certificate (if the message needs to be signed, or the response needs to be encrypted), this is done by changing this section of the WSDL file

<sp:SamlToken sp:IncludeToken=**"http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/Never"**>

<wsp:Policy>

<sp:WssSamlV20Token11 />

</wsp:Policy>

</sp:SamlToken>

To the following

<sp:X509Token sp:IncludeToken=**"http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRecipient"**>

<wsp:Policy>

<sp:WssX509V3Token10 />

</wsp:Policy>

</sp:X509Token>

This will require the WSC to supply its certificate in the request to the service.

We also need to empty the inner wsp:Policy section from the SignedSupportingToken element, otherwise CXF will attempt to use the token instead of the certificate.

Note that if the profile does not mandate that the bearer token should be signed by the WSC, then a SupportingTokens section can be used instead of a SignedSupportingTokens section.

<sp:SignedSupportingTokens xmlns:sp=**"http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"**>

<wsp:Policy>

<sp:IssuedToken sp:IncludeToken=**"http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRecipient"**>

<sp:RequestSecurityTokenTemplate />

<wsp:Policy />

<!--

<wsp:Policy>

<sp:WssSamlV20Token11 />

</wsp:Policy>

-->

</sp:IssuedToken>

</wsp:Policy>

</sp:SignedSupportingTokens>

### Modify the truststore

If the WS-SecurityPolicy requires that the WSC message is signed, the WSP must have some way of validating this signature. The bearer token (unlike the holder-of-key token) does not contain any information about the WSCs certificate, so trust must be established either by adding the WSCs certificate to the WSP truststore or adding one (or more) CA certificates to the WSP truststore – allowing a WSC to use any certificate issued by one of these CAs.

## Other Considerations

By default, a service using the Apache CXF framework is decently secured, but depending on the deployment methods used, some of the default settings used by Apache CXF should probably be tweaked.

### Replay detection

The default setting for Apache CXF is to use a memory cache (EHCache) to detect replayed messages. In a deployment scenario, where multiple servers host the same service, this would only detect replays against the same server instance. The following configuration key is used to configure the implementation class for the cache

ws-security.timestamp.cache.instance

and can be used to supply a different cache implementation.

This document does not cover how to implement a caching strategy to detect replays, but it is something one should consider in a multi-server deployment scenario.

### WS-SecureConversation

The reference code requires that the WSC must send a token on each request, which is then validated by the WSP. In scenarios where a WSC calls the WSP multiple times, this can be a big overhead, and in these scenarios, it might be worth enabling WS-SecureConversation.

When WS-SecureConversation is enabled, the initial request by the WSC establishes a secure session with the WSP, which is used for further requests. This session has a lower overhead than validating the token on each request but comes with the cost of an increased overhead on the first request.

This document does not cover how to enable WS-SecureConversation, and the CXF documentation should be consulted for this

<http://cxf.apache.org/docs/ws-secureconversation.html>

# Building a Web Service Consumer

This chapter covers all the steps necessary to build a WSC capable of calling a WSP secured as detailed in the previous chapter. This includes calling the STS and getting a token, as well as securing the request to the service in a way compliant with the security profile specified by the WSP.

## Reference Code

The reference code for the different security profiles implements many different scenarios, where the steps required to build a WSC slightly differ depending on the scenario.

This chapter will outline the general approach for using CXF to build a WSC, and the documentation for the specific profiles will cover the scenario specific details.

## Design Choices

As with the WSP in the previous chapter, the reference code for the WSC uses an XML-based configuration approach. It is possible to setup the same configuration using code, though this is not covered by this document.

## Security Requirements

The WSC must be configured to follow the security requirements of both the STS and the WSP it wants to call.

Using Apache CXF this can be accomplished in two ways. The recommended way is to use WS-SecurityPolicy [WS-SEC-POL], and the alternate way is to use WSS4J Interceptors. We will use the recommended approach, and configure the client using WS-SecurityPolicy.

The WSC must also be configured to trust the WSP, as the response from the WSP is signed. This requires that we setup a keystore that contains the public certificate of the WSP and configure this as a truststore in Apache CXF.

Finally, the STS must be aware of the WSC, so it can issue token to the WSC. This is outside the scope of this document, and the reference source for the various profiles comes with pre-established configurations in the STS components involved.

## WS-SecurityPolicy

In a perfect world, both the STS and the WSP would present WSDLs that contain the wsp:Policy sections that are needed by Apache CXF. As this is not always the case, the following two sections will outline how to modify the STS and WSP WSDL files to ensure that Apache CXF will handle the security requirements correctly.

### Configuring WS-SecurityPolicy for the STS

If the STS does not publish a WSDL with a useful WS-SecurityPolicy section, then such a policy can be applied to the WSDL manually. Download the WSDL file, modify it and then point CXF to the local (modified) copy instead of the online version.

It is recommended to read chapter 2.5 where WS-SecurityPolicy for WSPs is documented – it is the same approach that is used for the STS.

The reference code contains a WSDL file for an STS, which can be used as a starting point. The security policy in the WSDL should match the requirements set by the STS that the WSC needs to call to get a token. For this information, reference the STSs documentation.

Once the required policy is in place, the Apache CXF framework will be able to create a valid RequestSecurityToken request towards the STS and sign it in the expected way.

It may be required to add additional customization to the call to the STS – this depends on the STS in question, and the reference code contains such modifications in the scenarios where the NemLog-in STS is used. For more information about this, see the documentation for the individual security profiles.

### Configuring WS-SecurityPolicy for the WSP

As with the STS, if the WSP does not publish a WSDL with a valid security profile section, then one should be added by downloading the WSDL, modifying it, and then pointing CXF to the locally stored version.

In the reference code, all the WSPs publish WSDL files with valid security policy sections, which can be used as inspiration for other WSDL files.

## Configuring Keystores for the WSC

Like the WSP, the WSC must be configured with two keystores. The first keystore contains the client’s certificate and corresponding private key. The second keystore is a truststore, containing the public certificate of the STS as well as the WSPs that the WSC wishes to call.

In the reference code these two keystores are called client.jks and trust.jks, and the configuration file called client.properties is similar to the serviceKeystore.properties described in the previous chapter.

Note that it is also possible to have a truststore that contains CA certificates that are trusted. For instance, the trust.jks keystore could just contain OCES CA certificates, and trust would then be established to all OCES certificates issued by these CAs.

Note that by having a keystore, that only contains the certificates of the WSPs and STS that the WSC is communicating with, we have a much better control over who the WSC trusts.

As mentioned in the previous chapter, Apache CXF sometimes uses the password configured in the properties file, and sometimes requires a callback handler to give it the password. In the reference code, a callback class like the one found in the service project has been added to deal with this.

The XML configuration file for the WSC is called cxf.xml, and the relevant parts for configuring keystores are shown below

<jaxws:client>

<jaxws:properties>

<entry key=**"ws-security.signature.username"**

value=**"client"** />

<entry key=**"ws-security.signature.properties"**

value=**"client.properties"** />

<entry key=**"ws-security.callback-handler"**

value=**"client.callback.ClientCallbackHandler"** />

</jaxws:properties>

</jaxws:client>

Note that the above section of the configuration file only deals with communication with the WSP. The configuration for communicating with the STS (including keystore configuration) is covered below

## The STSClient Configuration and Implementation

The Apache CXF framework has a class called STSClient, which can be configured to call an STS.

If any modifications are required to the call to the STS, this can be done by extending the STSClient class, and then adding the customization in the extension class. Below we show how to extend (and configure) the STSClient class.

An example of such a customization is shown below

public class MySTSClient **extends** STSClient **{**

public MySTSClient**(**Bus b**)** **{**

**super(**b**);**

**}**

@Override

protected void addAppliesTo**(**XMLStreamWriter writer**,**

String appliesTo**)** **throws** XMLStreamException **{**

**if** **(**appliesTo **!=** **null** **&&** addressingNamespace **!=** **null)** **{**

writer**.**writeStartElement**(**"wsp"**,** "AppliesTo"**,**

"http://schemas.xmlsoap.org/ws/2002/12/policy"**);**

writer**.**writeNamespace**(**"wsp"**,**

"http://schemas.xmlsoap.org/ws/2002/12/policy"**);**

writer**.**writeStartElement**(**"wsa"**,** "EndpointReference"**,**

addressingNamespace**);**

writer**.**writeNamespace**(**"wsa"**,** addressingNamespace**);**

writer**.**writeStartElement**(**"wsa"**,** "Address"**,**

addressingNamespace**);**

writer**.**writeCharacters**(**appliesTo**);**

writer**.**writeEndElement**();**

writer**.**writeEndElement**();**

writer**.**writeEndElement**();**

**}**

**}**

**}**

The customized STSClient overrides the addAppliesTo() method, that sets the requested audience for the token. The code is copied from the reference code that calls the NemLog-in STS (because the NemLog-in requires a different namespace for this attribute than the one STSClient generates by default) and is an example on how to customize the STS client.

Depending on the STS that the WSC needs to call, different customizations may be required – this is usually done by overriding one of the many methods found in the base class.

The customized STSClient must be configured in the cxf.xml configuration file, which is done as shown below – note that it points to a locally stored WSDL file in the wsdlLocation property, allowing us to add policy modifications if needed to the local copy.

<jaxws:client>

<jaxws:properties>

<entry key=**"ws-security.sts.applies-to"**

value=**"https://wsp.itcrew.dk"** />

<entry key=**"ws-security.sts.client"**>

<bean class=**"client.sts.MySTSClient"**>

<constructor-arg ref=**"cxf"** />

<property name=**"wsdlLocation"**

value=**"src/main/resources/sts.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/}STS\_Port"** />

<property name=**"sendRenewing"** value=**"false"** />

<property name=**"sendKeyType"** value=**"false"** />

<property name=**"requiresEntropy"** value=**"false"** />

<property name=**"tokenType"**

value =**"http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0"** />

<property name=**"properties"**>

<map>

<entry key=**"ws-security.signature.username"**

value=**"client"** />

<entry key=**"ws-security.signature.properties"**

value=**"client.properties"** />

<entry key=**"ws-security.encryption.username"**

value=**"sts"** />

<entry key=**"ws-security.encryption.properties"**

value=**"sts.properties"**/>

</map>

</property>

</bean>

</entry>

</jaxws:properties>

</jaxws:client>

The STSClient accepts a range of properties, and depending on the STS, these properties may need to be set to specific values. The reference code contains a valid set of properties for calling the NemLog-in STS.

Note also that the keystores to be used for calling the STS is configured in this section. The WSCs own certificate is configured through the ws-security.signature.\* properties, and the truststore containing the STS certificate is configured through the ws-security.encryption.\* properties.

## Support Classes

As with the WSP, specific SOAP headers might be required in the request, and the WSC might be required to understand specific SOAP headers in the response.

The approach to dealing with this is identical to the one documented in chapter 2.7.1.

## Additional Configuration of Apache CXF

### Configure BSP 1.1 Compliance

As with the WSP, the WSC will by default attempt to be BSP 1.1 compliant. This may cause issues when dealing with encrypted tokens, and if this is the case, BSP compliance can be disabled using the following setting in cxf.xml

<jaxws:client>

<jaxws:properties>

<entry key=**"ws-security.is-bsp-compliant"** value=**"false"** />

</jaxws:properties>

</jaxws:client>

## Other Considerations

By default, the Apache CXF framework will cache the token issued by the STS and reuse it when calling the service for which the token was issued. The configuration key that controls which implementation is used for the token cache is

org.apache.cxf.ws.security.tokenstore.TokenStore

By setting this configuration key manually, it is possible to override specific settings in the TokenStore, or even implement a custom TokenStore. This can be useful if a more fine-grained control over time-to-live and renewal of the token is needed.

Note that the cached tokens are tied to the instance of the “Service” class, so if a new instance of the “Service” class is created each time the service is called, the cached tokens will not be used. In the reference code, the source file WSClient.java attempts to perform two calls to the WSP, and the second call to the WSP uses the cached token because the service instance is reused.

# Typical Errors

## Java Strong Crypto Not Installed

Getting an exception of the following type is usually a strong indication that the Unlimited Strength Jurisdiction Policy Files [CRYPTO] is not installed correctly

java.security.InvalidKeyException: Illegal key size

## A .keystore File in Home Folder

When running the reference code from the command-line (or Eclipse), the following exception might be thrown

java.security.UnrecoverableKeyException: Password must not be null

This is because Apache CXF will look for a .keystore file (note the . in the beginning of the filename) in the executing users home folder when setting up the SSL connection. The exception does not prevent the reference code from running, but to get rid of it, either supply a password to the file using the javax.ssl.net.keyStorePassword property, or remove the keystore from the home folder.

# Summary

This document has covered the steps needed to take an existing Apache CXF based WSP and secure it so WSCs must present a valid token from a trusted STS.

In combination with the profile specific documentation, and the bundled reference code, this can be used either as a template for creating WSPs and WSCs or as inspiration for creating such projects from scratch.

The document has also touched lightly on some of the security related issues that should be considered before deploying to production.

# References

[MAVEN] Apache Maven Build Tool v 3.x

<https://maven.apache.org/download.cgi>

[CRYPTO] Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files

**Java 7**

<http://www.oracle.com/technetwork/java/javase/downloads/jce-7-download-432124.html>

**Java 8** <http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html>

[WS-SEC-POL] WS-Security Policy 1.2

<http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/ws-securitypolicy-1.2-spec-os.html>