Java Reference Code for

NemLog-in STS Integration

(System User Scenario)

(Bootstrap Scenario)

(Signature Scenario)

Status: Version 1.1

Version: 09.03.20018

[Changelog 5](#_Toc508369270)

[1 Introduction 6](#_Toc508369271)

[1.1 Intended audience 7](#_Toc508369272)

[1.2 Prerequisites 7](#_Toc508369273)

[1.3 Apache CXF Version 7](#_Toc508369274)

[1.4 Disclaimer 8](#_Toc508369275)

[2 Building a Web Service Provider with Apache CXF 9](#_Toc508369276)

[2.1 Hello World Service 9](#_Toc508369277)

[2.2 Reference Code 9](#_Toc508369278)

[2.3 Design Choices 9](#_Toc508369279)

[2.4 Security Requirements 9](#_Toc508369280)

[2.5 Registration with NemLog-in 10](#_Toc508369281)

[2.6 WS-SecurityPolicy Configuration 11](#_Toc508369282)

[2.6.1 The General Security Policy 11](#_Toc508369283)

[2.6.2 The Input Policy 14](#_Toc508369284)

[2.6.3 The Output Policy 14](#_Toc508369285)

[2.7 Configuring Keystores for the Service 15](#_Toc508369286)

[2.8 Support Classes 16](#_Toc508369287)

[2.8.1 Framework Header 16](#_Toc508369288)

[2.8.2 Basic Privilege Profile parser 18](#_Toc508369289)

[2.9 Additional Configuration of Apache CXF 21](#_Toc508369290)

[2.9.1 Validating Audience Restriction 21](#_Toc508369291)

[2.9.2 Configure TimeToLive 22](#_Toc508369292)

[2.9.3 Configure BSP 1.1 Compliance 22](#_Toc508369293)

[2.9.4 The Full cxf-servlet.xml Configuration File 22](#_Toc508369294)

[2.10 The bearer-token version of the service 23](#_Toc508369295)

[2.10.1 Modify the WS-SecurityPolicy 23](#_Toc508369296)

[2.10.2 Modify the trust.jks truststore 24](#_Toc508369297)

[2.11 Deployment and Testing 24](#_Toc508369298)

[2.12 Encrypted Assertion Workarounds 24](#_Toc508369299)

[2.12.1 Issue 1 – KeyInfo Reference to EncryptedData 25](#_Toc508369300)

[2.12.2 Issue 2 – STR-Transform on EncryptedData 26](#_Toc508369301)

[2.13 Interoperability with .NET 27](#_Toc508369302)

[2.14 Other Considerations 28](#_Toc508369303)

[2.14.1 Replay detection 28](#_Toc508369304)

[2.14.2 WS-SecureConversation 28](#_Toc508369305)

[3 Building a Web Service Consumer with Apache CXF (System User Scenario) 30](#_Toc508369306)

[3.1 Reference Code 30](#_Toc508369307)

[3.2 Design Choices 30](#_Toc508369308)

[3.3 Security Requirements 30](#_Toc508369309)

[3.4 WS-SecurityPolicy 30](#_Toc508369310)

[3.4.1 Configuring WS-SecurityPolicy for the STS 31](#_Toc508369311)

[3.4.2 Configuring WS-SecurityPolicy for the Service 32](#_Toc508369312)

[3.5 Configuring Keystores for the Client 33](#_Toc508369313)

[3.6 The STSClient Configuration and Implementation 33](#_Toc508369314)

[3.7 Support Classes 36](#_Toc508369315)

[3.8 Additional Configuration of Apache CXF 37](#_Toc508369316)

[3.8.1 Configure BSP 1.1 Compliance 37](#_Toc508369317)

[3.8.2 The Full cxf.xml ConfigurationFile 37](#_Toc508369318)

[3.9 Encrypted Assertion Workarounds 38](#_Toc508369319)

[3.10 Using the Client 39](#_Toc508369320)

[3.11 Other Considerations 40](#_Toc508369321)

[4 Bootstrap scenario 41](#_Toc508369322)

[4.1 Reference Code 41](#_Toc508369323)

[4.2 Re-use from chapter 3 (The System User Scenario) 41](#_Toc508369324)

[4.3 Getting CXF and OIOSAML.Java to co-exist 42](#_Toc508369325)

[4.3.1 Integrating with NemLog-in SSO 43](#_Toc508369326)

[4.3.2 Getting a token and calling a service 44](#_Toc508369327)

[4.4 Using the reference code 45](#_Toc508369328)

[5 Signature scenario 47](#_Toc508369329)

[5.1 Reference Code 47](#_Toc508369330)

[5.2 Re-use from chapter 3 (The System User Scenario) 47](#_Toc508369331)

[5.3 The GUI for the desktop application 47](#_Toc508369332)

[5.4 Dealing with the user supplied keystore 48](#_Toc508369333)

[5.4.1 Providing the keystore 48](#_Toc508369334)

[5.4.2 Providing the password 49](#_Toc508369335)

[5.4.3 Providing the alias 49](#_Toc508369336)

[5.4.4 Interaction with the UI 50](#_Toc508369337)

[5.5 Using the reference code 51](#_Toc508369338)

[6 OIO IDWS Profile 1.1 53](#_Toc508369339)

[6.1 Update to SOAP 1.2 53](#_Toc508369340)

[6.2 Configuring a secure transport mechanism (TLS 1.2) 54](#_Toc508369341)

[6.3 Remove Framework header 55](#_Toc508369342)

[6.4 Changes for interoperability with .NET reference code 56](#_Toc508369343)

[6.5 Removed SOAP Fault example 58](#_Toc508369344)

[7 Example payloads 59](#_Toc508369345)

[7.1 WSC-TO-STS 59](#_Toc508369346)

[7.2 STS-TO-WSC 59](#_Toc508369347)

[7.3 WSC-TO-WSP 60](#_Toc508369348)

[7.4 WSP-TO-WSC 60](#_Toc508369349)

[8 Summary 61](#_Toc508369350)

[9 Typical Errors 62](#_Toc508369351)

[9.1 Java Strong Crypto Not Installed 62](#_Toc508369352)

[9.2 A .keystore File in Home Folder 62](#_Toc508369353)

[10 References 63](#_Toc508369354)

# Changelog

01-12-2015 Initial release

09-03-2018 Updated with OIO IDWS profile, and updated certificates

# Introduction

This document is a companion to the Java reference source code that showcase how to use the Apache CXF framework to implement

Either a Liberty Basic SOAP Binding solution

* A Web Service Provider (WSP) that requires clients to present a token issued by the NemLog-in STS, showing how to perform the following validations
  + Require the client to conform to the Basic Liberty SOAP Binding [LIBERTY]
  + Parse the issued Basic Privilege Profile [OIO-BPP] information in the presented token
  + Ensure that the request is no older than 5 minutes
* A Web Service Consumer (WSC) that can
  + Interact with the NemLog-in STS to get a token issued
  + Call the above WSP with the issued token
  + Validate that the response from the WSP conforms to the Basic Liberty SOAP Binding [LIBERTY]

Or a more modern OIO IDWS SOAP Profile 1.1 solution

* A Web Service Provider (WSP) that requires clients to present a token issued by the NemLog-in STS, showing how to perform the following validations
  + Require the client to conform to the OIO IDWS SOAP Profile 1.1 [OIOIDWS]
  + Parse the issued Basic Privilege Profile [OIO-BPP] information in the presented token
  + Ensure that the request is no older than 5 minutes
* A Web Service Consumer (WSC) that can
  + Interact with the NemLog-in STS to get a token issued
  + Call the above WSP with the issued token
  + Validate that the response from the WSP conforms to the OIO IDWS SOAP Profile 1.1 [OIOIDWS]

This document was originally written to cover the LBSB case, and the changes from LBSB to OIOIDWS are minor, so instead of duplicating most of the content of this document, an appendix with the required changes to implement OIO IDWS based on the LBSB code is added at the end of this document.

Note that the NemLog-in STS supports a number of different usage scenarios that differ in how authentication to the STS is performed, including bootstrap token case, local token case and signature case illustrated below:



This reference code only covers the following scenarios

1. The system user scenario where the WSC authenticates to the STS as a system user using a signature (e.g. a FOCES or VOCES certificate). Here, the WSC gets a token to act as the WSC (a system user).
2. The bootstrap token scenario, where the WSC is also a SAML 2.0 Service Provider, integrated with the NemLog-in SSO. Here the WSC authenticates to the STS using a signature (e.g. a FOCES or VOCES certificate) and exchanges a bootstrap token (representing the user) received during the Web SSO roundtrip. The WSC gets a token to act on behalf of the end-user.
3. The signature scenario where the WSC authenticates to the STS as a user using the users signature (e.g. a MOCES certificate). The WSC gets a token to act on behalf of the end-user.

Please refer to the NemLog-in documentation for descriptions of the other usage scenarios.

Since all relevant configuration and customization of Apache CXF is contained in this document, it is possible to read it as a stand-alone document.

It is still highly recommended to read this document together with the reference code, as the context of the configuration and customization increases the value.

## 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 3.0.6, but Apache CXF 2.7.17 can be used as well. If CXF 2.7.17 is used, note that the WSPasswordCallback class resides in a different package, and the following namespace

**import** org**.**apache**.**wss4j**.**common**.**ext**.**WSPasswordCallback**;**

must be changed to

**import** org**.**apache**.**ws**.**security**.**WSPasswordCallback**;**

Note that Apache CXF 3.1.0 was released after the development of the reference code, and it has not been tested.

## 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 with Apache CXF

This chapter covers all the steps necessary to secure a web service using Apache CXF. The reference source code is based on a very simple web service called HelloWorld, and all parts of the code is clearly packaged and commented, so those sections relevant to dealing with security is easily identified.

Please note that the documentation covers two different versions of the same service. The first service accepts holder-of-key tokens, and is used in the System User Scenario and the Bootstrap Scenario, and a service that accepts bearer-tokens, which is used in the Signature Scenario.

## Hello World Service

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 the root of 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 covers 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 – if a code based approach is used, the dependency on Spring can be removed.

## Security Requirements

The service must be configured to enforce validation of the client request according to the Basic Liberty SOAP Binding.

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.

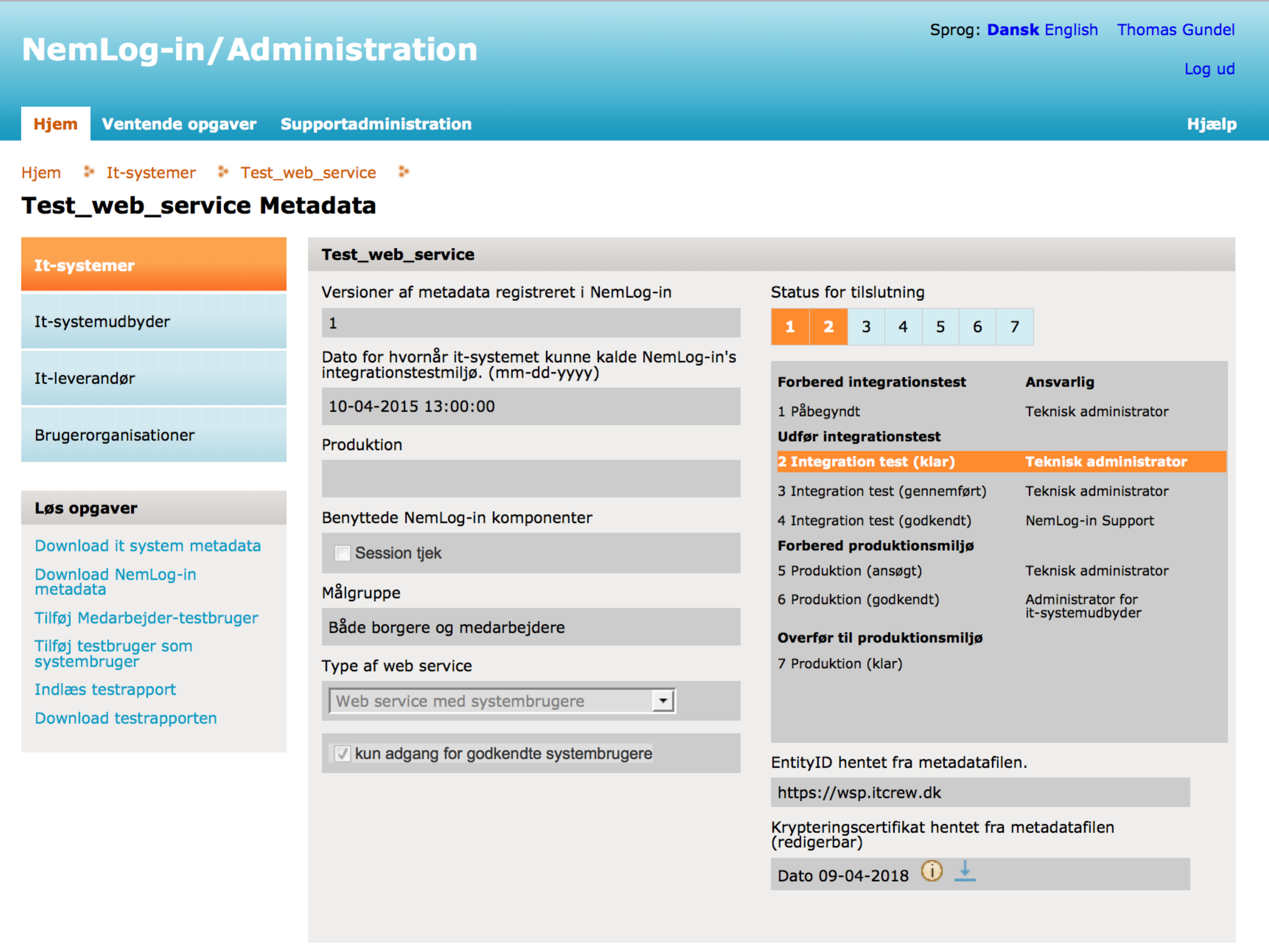
The service must also be configured to trust the STS, which requires a keystore containing the STS certificate to be configured as a truststore.

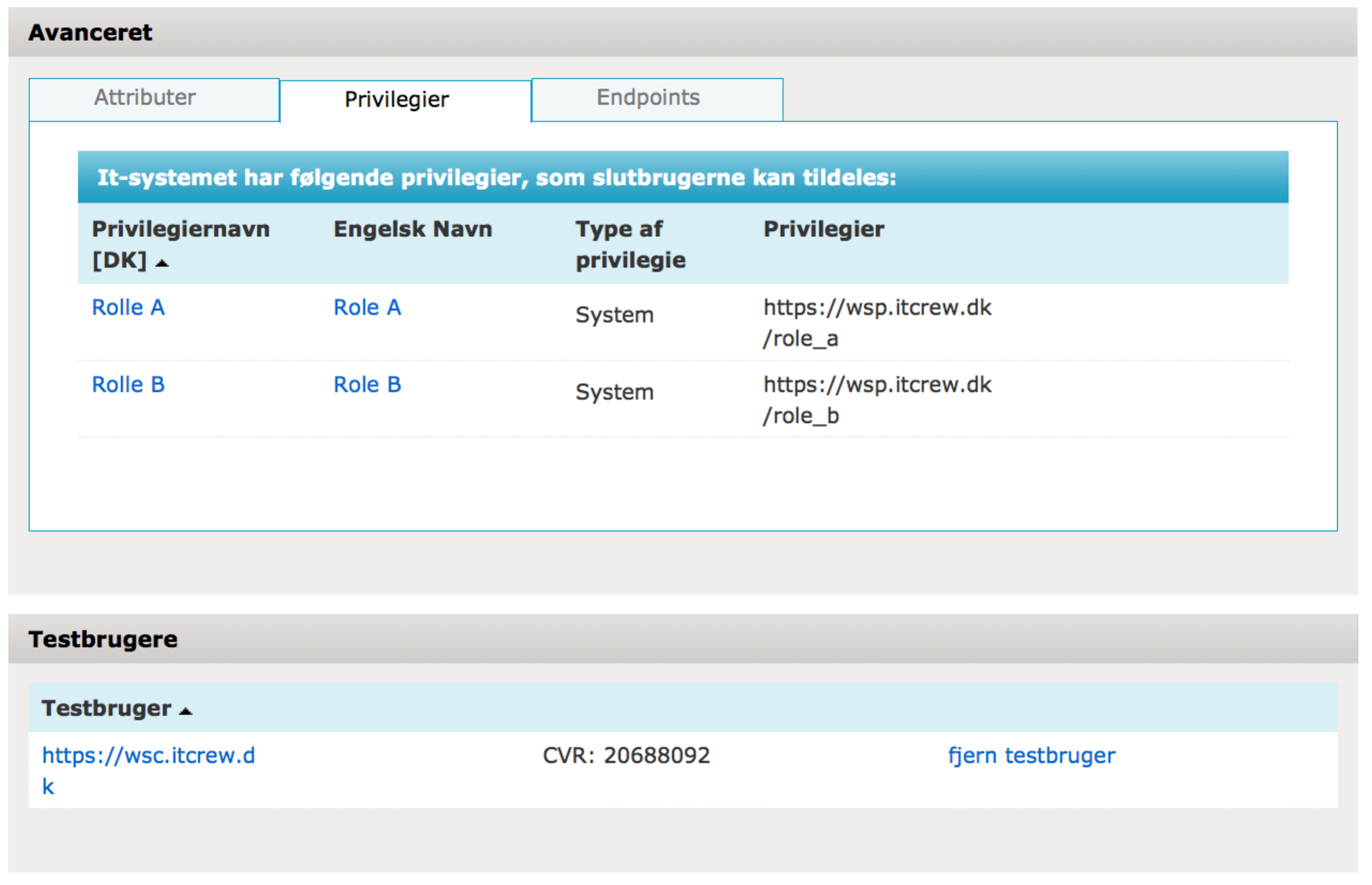
## Registration with NemLog-in

The example WSC and WSP have been registered with NemLog-in[[1]](#footnote-2) to make the example work:

* The client (WSC) is registered as a system user with EntityID “https://wsc.itcrew.dk”.
* The WSP is registered as a web service with the EntityID “<https://wsp.itcrew.dk>” with two privileges (roles). The client has been granted these two privileges by the administrator of the WSP such they will appear in tokens issued by the STS to the WSP.

The process of registering with NemLog-in is not covered in this guide, but a detailed process description with associated screenshots will be provided on Digitaliser.dk in the group[[2]](#footnote-3): A screen shot showing the registration details for the WSP is shown below:





## 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].

The 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
* An output policy that describes the signing and encryption requirements that the service must follow

### The General Security Policy

Below the full wsp:Policy section used by the reference code is shown. 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 project.

<wsp:Policy wsu:Id=**"HelloWorldBindingPolicy"**>

<wsp:ExactlyOne>

<wsp:All>

<wsam:Addressing wsp:Optional=**"false"**>

<wsp:Policy />

</wsam:Addressing>

<sp:AsymmetricBinding>

<wsp:Policy>

<sp:InitiatorToken>

<wsp:Policy>

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

<wsp:Policy>

<sp:WssSamlV20Token11 />

</wsp:Policy>

</sp:SamlToken>

</wsp:Policy>

</sp:InitiatorToken>

<sp:RecipientToken>

<wsp:Policy>

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

<wsp:Policy>

<sp:WssX509V3Token10 />

</wsp:Policy>

</sp:X509Token>

</wsp:Policy>

</sp:RecipientToken>

<sp:AlgorithmSuite>

<wsp:Policy>

<sp:Basic256 />

</wsp:Policy>

</sp:AlgorithmSuite>

<sp:Layout>

<wsp:Policy>

<sp:Strict />

</wsp:Policy>

</sp:Layout>

<sp:ProtectTokens />

<sp:IncludeTimestamp />

<sp:OnlySignEntireHeadersAndBody />

</wsp:Policy>

</sp:AsymmetricBinding>

<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>

<sp:WssSamlV20Token11 />

</wsp:Policy>

</sp:IssuedToken>

</wsp:Policy>

</sp:SignedSupportingTokens>

</wsp:All>

</wsp:ExactlyOne>

</wsp:Policy>

If we look at the policy from a top-level, it contains three main sections

<wsam:Addressing />

<sp:AsymmetricBinding />

<sp:SignedSupportingTokens />

The first element enables WS-Addressing, which is required by the Basic Liberty SOAP Binding, and will force the client to send the Addressing headers that is required by this profile (wsa:Action and wsa:MessageID).

The second element describes the ***primary*** tokens used by the client and service, and mandates that the tokens should be used in an asymmetric binding. This tells the client that it is required to sign the request, and the service that it is supposed to sign the response.

If we take a detailed look at the AsymmetricBinding element, it contains the following elements

<sp:InitiatorToken />

<sp:RecipientToken />

<sp:AlgorithmSuite />

<sp:Layout />

<sp:ProtectTokens />

<sp:IncludeTimestamp />

<sp:OnlySignEntireHeadersAndBody />

The InitiatorToken and RecipientToken elements describes the type of token that the client and service must use to sign the request, in this case a SAML token is configured for the client (the token issued by the STS) and an X.509 token is configured for the service.

Note that the IncludeToken attribute is set to Never on the InitiatorToken, as the service will not have a direct trust relationship established with the client. Instead the issued token is used to establish trust with the client. Likewise the IncludeToken attribute is set to AlwaysToInitiator on the RecipientToken, as this will ensure that the services certificate is returned to the client as part of the response, as this is required by the Basic Liberty SOAP Binding.

The AlgorithmSuite element describes which cryptographic algorithms the client and service must use when signing and encrypting data.

The Layout element describes how the header elements must be added to the SOAP header.

The ProtectTokens element requires that any tokens added to the request and response must be signed by the sender.

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

Finally, the OnlySignEntireHeadersAndBody element requires that the client and service 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 ***secondary*** token that the client must use when calling the service. In this case it requires that the client gets a SAML token from the STS and that it must be digitally signed by the client (sp:SignedSupportingToken as opposed to just sp:SupportingToken).

### The Input Policy

Below the full wsp:Policy section used by the reference code for securing the input is shown. Like the general policy, this too can be found in-context in the WSDL file for the service.

<wsp:Policy wsu:Id=**"HelloWorldBinding\_HelloWorld\_Input\_Policy"**>

<wsp:ExactlyOne>

<wsp:All>

<sp:SignedParts>

<sp:Body />

<sp:Header Name=**"To"**

Namespace=**"http://www.w3.org/2005/08/addressing"** />

<sp:Header Name=**"From"**

Namespace=**"http://www.w3.org/2005/08/addressing"** />

<sp:Header Name=**"FaultTo"**

Namespace=**"http://www.w3.org/2005/08/addressing"** />

<sp:Header Name=**"ReplyTo"**

Namespace=**"http://www.w3.org/2005/08/addressing"** />

<sp:Header Name=**"MessageID"**

Namespace=**"http://www.w3.org/2005/08/addressing"** />

<sp:Header Name=**"RelatesTo"**

Namespace=**"http://www.w3.org/2005/08/addressing"** />

<sp:Header Name=**"Action"**

Namespace=**"http://www.w3.org/2005/08/addressing"** />

<sp:Header Name=**"Framework"**

Namespace=**"urn:liberty:sb"** />

</sp:SignedParts>

</wsp:All>

</wsp:ExactlyOne>

</wsp:Policy>

Usually the input policy contains two relevant elements, the sp:SignedParts and sp:EncryptedParts elements. In the reference source, the sp:EncryptedParts is left out, as the message is not encrypted. The sp:SignedParts controls which headers are signed, together with the body.

Note that the header elements included in the sp:SignedParts section are exactly the headers required by the Basic Liberty SOAP Binding. Other headers can be included as needed.

### The Output Policy

Below the full wsp:Policy section used by the reference code for securing the response from the service is shown. Note that it is very similar to the input policy, for the same reasons as mentioned above.

<wsp:Policy wsu:Id=**"HelloWorldBinding\_HelloWorld\_Output\_Policy"**>

<wsp:ExactlyOne>

<wsp:All>

<sp:SignedParts>

<sp:Body />

<sp:Header Name=**"To"**

Namespace=**"http://www.w3.org/2005/08/addressing"** />

<sp:Header Name=**"From"**

Namespace=**"http://www.w3.org/2005/08/addressing"** />

<sp:Header Name=**"FaultTo"**

Namespace=**"http://www.w3.org/2005/08/addressing"** />

<sp:Header Name=**"ReplyTo"**

Namespace=**"http://www.w3.org/2005/08/addressing"** />

<sp:Header Name=**"MessageID"**

Namespace=**"http://www.w3.org/2005/08/addressing"** />

<sp:Header Name=**"RelatesTo"**

Namespace=**"http://www.w3.org/2005/08/addressing"** />

<sp:Header Name=**"Action"**

Namespace=**"http://www.w3.org/2005/08/addressing"** />

<sp:Header Name=**"Framework"**

Namespace=**"urn:liberty:sb"** />

</sp:SignedParts>

</wsp:All>

</wsp:ExactlyOne>

</wsp:Policy>

## Configuring Keystores for the Service

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

Both JKS and PKCS12 keystore formats are supported by Apache CXF. The reference code comes pre-configured with two JKS keystores called service.jks and trust.jks, the latter being the truststore containing the STS certificate.

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 content of the propertyfile (serviceKeystore.properties) used in the reference code is shown

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

As mentioned in the Design Choices section, the reference code uses an XML-based configuration approach, and this property file is configured in the XML configuration file for CXF (cxf-servlet.xml found under src/main/resources). Below an excerpt from the cxf-servlet.xml file is shown, outlining the relevant configuration of the keystore files

<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. Below is shown the source from the callback handler in the reference code

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 has 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 requirements specific to the NemLog-in STS and the Liberty Basic SOAP Binding must be dealt with by custom code.

The reference code contains additional Java classes to deal with the mandatory Framework header, as well as the Basic Privilege Profile attributes issued by the STS.

### Framework Header

The Basic Liberty SOAP Binding requires that the service return an sbf:Framework header as part of the response. To do this, we need a Java class that can be marshalled to XML by the XML framework used by Apache CXF.

The reference code contains class with the correct attributes and annotations to ensure marshalling, and the code is shown below

@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**;**

**}**

**}**

To ensure that this header is added to the response created by the service, we need to implement and configure an Interceptor that can catch the outgoing response, and add the additional header. The Java code for this Interceptor is shown below

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**);**

**}**

**}**

Note that the super() call to the parent class constructor informs the CXF framework in which phase of the response creation that the interceptor is executed. By using the PRE\_PROTOCOL phase we are assured that we add the header before the response is signed, which is needed as the Framework header is one of the headers that must be signed.

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.

<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"**/>

### Basic Privilege Profile parser

The token issued by the STS, and delivered to the service by the client, contains a Basic Privilege Profile attribute, which must be parsed and potentially handled by the service. The reference source contains code that locates the relevant attribute in the token, marshal it to a Java class, and place it on a ThreadLocal so it is globally accessible. Finally, the reference code shows how to access this Java class from the business logic of the service, by printing the content of the attribute to the console.

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

The first class is an ordinary ThreadLocal holder, shown below for completeness:

public class AssertionHolder **{**

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, which 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 SamlFilter **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** **{**

AssertionHolder**.**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 is shown below for completeness

<filter>

<filter-name>**SamlFilter**</filter-name>

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

</filter>

<filter-mapping>

<filter-name>**SamlFilter**</filter-name>

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

</filter-mapping>

The final class does most of the work, and is shown below

public class DigstSamlAssertionValidator **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**));**

AssertionHolder**.**set**(**

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

**}**

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

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

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

**}**

**}**

**}**

**}**

**}**

**}**

**}**

**return** vCredential**;**

**}**

**}**

The above code extracts the attribute with the name “Privileges” from the set of attributes in the token, and then takes the “raw” string value of that attribute, base64 decodes it and finally uses the JAXB framework to marshal it into a Java class.

The reference code contains Java classes that model the Basic Privilege Profile schema, which can be used for this purpose. They are generated using the JAXB command-line tool (xjc), based on the official BPP schema files, and are located in the service.bpp package in the source.

The DigstSamlAssertionValidator 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 is shown below

<jaxws:endpoint>

<jaxws:properties>

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

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

</jaxws:properties>

</jaxws:endpoint>

Once the above classes has been implemented and configured to use, the Basic Privilege Profile attribute in the incoming token will automatically be processed, and can be accessed anywhere in the code like this:

// get the Privileges from the presented token

PrivilegeListType privilegeListType **=** AssertionHolder**.**get**();**

// print the privileges

**if** **(**privilegeListType **!=** **null)** **{**

**for** **(**PrivilegeGroupType privilegeGroup **:**

privilegeListType**.**getPrivilegeGroup**())** **{**

System**.**out**.**println**(**"scope: " **+** privilegeGroup**.**getScope**());**

**for** **(**String privilegeStr **:** privilegeGroup**.**getPrivilege**())** **{**

System**.**out**.**println**(**"privilege: " **+** privilegeStr**);**

**}**

**}**

**}**

## 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 5 minutes, and finally we need to configure Apache CXF to accept that the token issued by the STS is not 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 tokens Audience is validated against.

This code is added to the DigstSamlAssertionValidator found above, by adding a hardcoded field in the class that contains the EntityId registered with the STS (<https://wsp.itcrew.dk/>),

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

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

**{**

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

**}**

**};**

And then adding this value to the requestData before performing the validation in the validate() method

@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, and no actual configuration is needed. For completeness sake, the configuration value that controls this setting is shown below. It is added the the cxf-servlet.xml file, and the value is written in seconds (300 seconds = 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. Because of this we have to disable BSP 1.1 compliance checking. This is done in the apache-cxf.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 Full cxf-servlet.xml Configuration File

The full cxf-servlel.xml configuration file is shown below for completeness

<?xml version=**"1.0"** encoding=**"UTF-8"**?>

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

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

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

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

xsi:schemaLocation=**"**

**http://www.springframework.org/schema/beans**

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

**http://cxf.apache.org/core**

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

**http://cxf.apache.org/jaxws**

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

<cxf:bus>

<cxf:features>

<cxf:logging />

</cxf:features>

<cxf:outInterceptors>

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

</cxf:outInterceptors>

<cxf:outFaultInterceptors>

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

</cxf:outFaultInterceptors>

</cxf:bus>

<bean id=**"FrameworkHeaderInterceptor"**

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

<jaxws:endpoint

id=**"helloworld"**

implementor=**"service.HelloWorldPortTypeImpl"**

wsdlLocation=**"WEB-INF/wsdl/HelloWorld.wsdl"**

address=**"/helloworld"**>

<jaxws:properties>

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

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

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

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

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

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

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

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

</jaxws:properties>

</jaxws:endpoint>

</beans>

## The bearer-token version of the service

Modifying the above service so it accepts bearer tokens instead of holder-of-key tokens requires two minor modifications. The source found in the “service-bearer” folder is a copy of the “service-hok” folder with these two modifications applied.

### Modify the WS-SecurityPolicy

As the web service consumer calling the service only supplies a bearer-token, we need to ensure that the client supplies its certificate, 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 web service consumer to supply its certificate in the request to the service.

We also need to remove the inner wsp:Policy section from the SignedSupportingToken element to avoid both parties attempting to use the bearer-token

<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 trust.jks truststore

Since the bearer-token is not used for validating the signature on the request, the service must establish trust towards the web service consumer directly. This is solved by adding the CA certificate that has issued the web service consumers FOCES certificate into the trust.jks truststore. The trust.jks truststore in the source comes preconfigured with the CA certificate from the NemID test environment.

## Deployment and Testing

To build and deploy the service, perform the following commands from the command-line. Make sure to execute the command from the root of the source folder (where the main pom.xml file is located)

$ mvn clean install

This will compile both versions of the service.

Once the project has been compiled, go into either of the service folders, and start that specific service by running the following command (please note that both services cannot run at the same time)

$ mvn tomcat7:run-war

This will launch the service on the following URL

<http://localhost:8080/HelloWorld/services/>

Testing that the service works will require a web service consumer. The following chapter covers how to build different types of web service consumers that can call the services.

## Encrypted Assertion Workarounds

The Apache CXF framework does not fully support all use cases with EncryptedAssertion elements, and especially it does not support the Holder-of-key use-case used by the Liberty Basic SOAP Binding profile.

The root cause is the way that Apache CXF deals with encrypted elements. Whenever Apache CXF decrypts an element, it will replace the encrypted value in the DOM with the corresponding decrypted value, so the token issued by the STS, will go from this

<EncryptedAssertion>

<EncryptedData id=**"encryptedDataId"**>

<EncryptionMethod />

<KeyInfo />

<CipherData />

</EncryptedData>

</EncryptedAssertion>

to this

<Assertion id=**"decryptedDataId"**>

<Issuer />

<Signature />

<Subject />

<Conditions />

<AttributeStatement />

</Assertion>

Note that only the service can decrypt this token, so the client calling the service has no way of knowing what the ID of the <Assertion> element is, which leads us to the first of two issues

### Issue 1 – KeyInfo Reference to EncryptedData

The client has signed the request according to the Liberty Basic SOAP Binding, and the service must validate this signature. The client has, correctly, referenced the token using a SecurityTokenReference element, using the ID of the EncryptedData element inside the EncryptedAssertion. When the service needs to validate the signature, the ID referenced by the SecurityTokenReference no longer exists (as the element has been decrypted), and the lookup fails, causing the signature validation to fail.

To solve this issue, the code in CXF that is responsible for dereferencing the element has been modified slightly. The workaround does by allowing any lookup that attempts to dereference an element with the ID “encryptedassertion” to match the first decrypted Assertion element in the DOM structure. As the ID of the EncryptedData element is ALWAYS set to “encryptedassertion” by the NemLog-in STS, this will work.

The method that has been overwritten is displayed below, and it is found within the class WSDocInfo, located in the ***wss4j-ws-security-dom*** module used by CXF.

public WSSecurityEngineResult getResult**(**String uri**)** **{**

String id **=** uri**;**

**if** **(**id **==** **null)** **{**

**return** **null;**

**}** **else** **if** **(**id**.**charAt**(**0**)** **==** '#'**)** **{**

id **=** id**.**substring**(**1**);**

**}**

**if** **(**resultsList **!=** **null)** **{**

**for** **(**WSSecurityEngineResult result **:** resultsList**)** **{**

**if** **(**result **!=** **null)** **{**

// START DIGST WORKAROUND

**if** **(**"encryptedassertion"**.**equals**(**id**))** **{**

Object samlAssertion **=** result**.**get**(**

WSSecurityEngineResult**.**TAG\_SAML\_ASSERTION**);**

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

**return** result**;**

**}**

**}**

// END DIGST WORKAROUND

String cId **=** **(**String**)**result**.**get**(**WSSecurityEngineResult**.**TAG\_ID**);**

**if** **(**id**.**equals**(**cId**))** **{**

**return** result**;**

**}**

**}**

**}**

**}**

**return** **null;**

**}**

Note that this workaround has been applied to version 2.0.5 of this module in the reference code. If another version of the wss4j module is used, the override should be applied to that version to ensure compatibility.

The reference code also contains a distribution of the wss4j module that has been precompiled with this override.

### Issue 2 – STR-Transform on EncryptedData

The client must sign the token, which is done using a SecurityTokenReference, and the STR-Transform algorithm. The STR-Transform requires that the referenced element has a default namespace, and if one does not exist, it will either inherit the namespace from the parent element, or emit an empty default namespace (xmlns="").

As the EncryptedData element does not contain a default namespace, it should inherit the value from the parent element. In this case the EncryptedAssertion element. This element has a default namespace with the value (xmlns="urn:oasis:names:tc:SAML:2.0:assertion").

The client will correctly compute this value, but the service will not. Due to the way Encrypted elements are replaced by the corresponding decrypted values in the DOM, this means that the EncryptedData element will be orphaned, and thus will not have a parent. This will cause the STR-Transform algorithm to emit an empty default namespace when performing the Transform.

The end-result is that the client and the service will not compute the same digest for the SecurityTokenReference, and the signature check will fail.

To solve this issue, the code in CXF that is responsible for computing the STR-Transform has been modified slightly. In the very specific case, where the element to compute the STR-Transform on has the ID “encryptedassertion”, and the element does not have a parent element, code has been added to create a dummy parent element, where this parent element contains the expected default namespace (xmlns="urn:oasis:names:tc:SAML:2.0:assertion").

The method that has been overridden is displayed below, trimmed to display only the relevant sections of the method. The method itself can be found in the class STRTransform located in the ***wss4j-ws-security-dom*** module used by CXF (the same module as in issue 1).

private Data transformIt**(**Data data**,** XMLCryptoContext xc**,** OutputStream os**)** **{**

**...** snip **...**

**if** **(**dereferencedToken **!=** **null)** **{**

**...** snip **...**

**}**

// BEGIN DIGST WORKAROUND

boolean parentNodeIsNull **=** **(**dereferencedToken**.**getParentNode**()** **==** **null);**

boolean encryptedAssertionId **=**

**(**dereferencedToken**.**getAttribute**(**"wsu:Id"**) !=** **null** **&&**

dereferencedToken**.**getAttribute**(**"wsu:Id"**).**equals**(**"encryptedassertion"**));**

**if** **(**parentNodeIsNull **&&** encryptedAssertionId**)** **{**

DocumentBuilderFactory factory **=** DocumentBuilderFactory**.**newInstance**();**

DocumentBuilder documentBuilder **=** factory**.**newDocumentBuilder**();**

Document document **=** documentBuilder**.**newDocument**();**

Element encryptedAssertion **=** document**.**createElement**(**"EncryptedAssertion"**);**

encryptedAssertion**.**setAttributeNS**(**"http://www.w3.org/2000/xmlns/"**,**

"xmlns"**,**

"urn:oasis:names:tc:SAML:2.0:assertion"**);**

dereferencedToken **=** **(**Element**)** document**.**importNode**(**

dereferencedToken**,** **true);**

encryptedAssertion**.**appendChild**(**dereferencedToken**);**

dereferencedToken = encryptedAssertion;

NodeList securityTokenReferences =

dereferencedToken.getElementsByTagName("o:SecurityTokenReference");

if ((securityTokenReferences.getLength() > 0) &&

((securityTokenReferences.item(0) instanceof Element)))

{

Element securityTokenReference = (Element)

securityTokenReferences.item(0);

securityTokenReference.setAttributeNS(

"http://www.w3.org/2000/xmlns/",

"xmlns:o",

"http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd");

**}**

**}**

// END DIGST WORKAROUND

//

// C14n with specified algorithm. According to WSS Specification.

//

buf **=** canon**.**canonicalizeSubtree**(**dereferencedToken**,** "#default"**,** **true);**

**...** snip **...**

**}**

This workaround has been applied to the same version of the wss4j module as in issue 1, and is part of the precompiled distribution that comes with the reference code.

## Interoperability with .NET

The Apache CXF framework is not fully interoperable with .NET when it comes to dealing with encrypted assertions, so another modification is required to ensure that CXF generates the expected references as expected by .NET.

The method that has been overwritten is displayed below, and it is found within the class AbstractBuildingBuilder, located in the ***cxf-rt-ws-security*** module used by CXF.

protected WSSecSignature getSignatureBuilder**(**AbstractToken token**,**

boolean attached**,** boolean endorse**)** **throws** WSSecurityException **{**

// ... snip ...

// remove this line and replace it with the code below

// SecurityTokenReference secRef = new

// SecurityTokenReference(cloneElement(ref), new BSPEnforcer());

// replace with this

SecurityTokenReference secRef **=** **null;**

**if** **((**id **!=** **null)** **&&** **(**id**.**equals**(**"encryptedassertion"**)))** **{**

Document doc **=** ref**.**getOwnerDocument**();**

secRef **=** **new** SecurityTokenReference**(**doc**);**

secRef**.**addTokenType**(**"http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0"**);**

Element keyId **=** doc**.**createElementNS**(**"http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"**,** "wsse:KeyIdentifier"**);**

keyId**.**setAttributeNS**(null,** "ValueType"**,** "http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLID"**);**

keyId**.**appendChild**(**doc**.**createTextNode**(**id**));**

Element elem **=** secRef**.**getElement**();**

elem**.**appendChild**(**keyId**);**

**}**

**else** **{**

secRef **=** **new** SecurityTokenReference**(**cloneElement**(**ref**),** **new** BSPEnforcer**());**

**}**

// ... snip ....

**}**

## 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 repays against the same server instance. The following configuration key is used to configure the implementation class for the cache

ws-security.timestamp.cache.instance

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

### WS-SecureConversation

The reference code requires that the client must send a token on each request, which is then validated by the service. In scenarios where a client calls the service 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 client establishes a secure session with the service, 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 with Apache CXF (System User Scenario)

This chapter covers all the steps necessary to build a Web Service Consumer (client) capable of calling a web service 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 Basic Liberty SOAP Binding.

This chapter will focus on the System user scenario, and the following chapters will cover the details for the other consumer implementations.

Note that the reference code is written explicitly to call the holder-of-key token version of the HelloWorld service in the previous chapter, but the code that is service-specific is easily isolated, and the rest of the code can either be reused directly or work as inspiration for calling other services that are secured in a similar way.

## Reference Code

The code for the System User Scenario is found in the folder “system-user-scenario” in the root of the reference code distribution. The project is self-contained, and does not directly depend on any of the other modules in the reference code distribution (to test the code, it does require the service module to be running).

## Design Choices

As with the service in the previous chapter, the reference code for the client 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 client must be configured to follow the security requirements of both the STS and the service it wants to call. It must also ensure that it follows the requirements detailed by the Basic Liberty SOAP Binding [LIBERTY].

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

Finally the STS must be aware of the client, so it can issue token to the client. This is outside the scope of this document, and the reference source for this client has already been registered with the STS under the following EntityId: <https://wsc.itcrew.dk>

## WS-SecurityPolicy

In a perfect world, both the STS and the service would present WSDLs that contains 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 service WSDL files to ensure that Apache CXF will handle the security requirements correctly.

### Configuring WS-SecurityPolicy for the STS

The WSDL of the NemLog-in STS contains a wsp:Policy section, but unfortunately the policy section is not representative of the actual security requirements for calling the STS, so we will have to create our own.

Also, the WSDL of the NemLog-in STS does not contain the operations usually found in a WSDL for a WS-Trust compliant STS. For this reason we will create a WSDL from scratch for the STS, and completely ignore the WSDL presented by the NemLog-in STS.

The reference code contains a WSDL (src/main/resources/sts.wsdl) that has the Issue operation exposed by the NemLog-in STS, and contains the required wsp:Policy section. The WSDL is modelled after the WSDL found in the WS-Trust 1.4 specification [TRUST-WSDL].

Please note that the WSDL has the URL of the test-STS hardcoded, and this should be changed to point to the URL of the production STS before using this in production.

Below is shown the required wsp:Policy section that has been added to the WSDL to ensure that Apache CXF will call the STS correctly.

<wsp:Policy wsu:Id=**"X509\_policy"**>

<wsp:ExactlyOne>

<wsp:All>

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

<wsp:Policy>

<sp:InitiatorToken>

<wsp:Policy>

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

<wsp:Policy>

<sp:WssX509V3Token10 />

</wsp:Policy>

</sp:X509Token>

</wsp:Policy>

</sp:InitiatorToken>

<sp:RecipientToken>

<wsp:Policy>

<sp:X509Token sp:IncludeToken=**"http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/Never"**>

<wsp:Policy>

<sp:WssX509V3Token10 />

<sp:RequireIssuerSerialReference />

</wsp:Policy>

</sp:X509Token>

</wsp:Policy>

</sp:RecipientToken>

<sp:AlgorithmSuite>

<wsp:Policy>

<sp:Basic256 />

</wsp:Policy>

</sp:AlgorithmSuite>

<sp:Layout>

<wsp:Policy>

<sp:Strict />

</wsp:Policy>

</sp:Layout>

<sp:ProtectTokens />

<sp:IncludeTimestamp />

<sp:OnlySignEntireHeadersAndBody />

</wsp:Policy>

</sp:AsymmetricBinding>

</wsp:All>

</wsp:ExactlyOne>

</wsp:Policy>

<wsp:Policy wsu:Id=**"Input\_policy"**>

<wsp:ExactlyOne>

<wsp:All>

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

<sp:Body />

<sp:Header Name=**"To"**

Namespace=**"http://www.w3.org/2005/08/addressing"** />

<sp:Header Name=**"From"**

Namespace=**"http://www.w3.org/2005/08/addressing"** />

<sp:Header Name=**"FaultTo"**

Namespace=**"http://www.w3.org/2005/08/addressing"** />

<sp:Header Name=**"ReplyTo"**

Namespace=**"http://www.w3.org/2005/08/addressing"** />

<sp:Header Name=**"MessageID"**

Namespace=**"http://www.w3.org/2005/08/addressing"** />

<sp:Header Name=**"RelatesTo"**

Namespace=**"http://www.w3.org/2005/08/addressing"** />

<sp:Header Name=**"Action"**

Namespace=**"http://www.w3.org/2005/08/addressing"** />

</sp:SignedParts>

</wsp:All>

</wsp:ExactlyOne>

</wsp:Policy>

The policy is similar to the one described in the previous chapter, that was used to describe the security requirements of the service. The two major changes are with the ***primary*** and ***secondary*** tokens.

1. Unlike the policy for the service, the primary token is a X.509 token, and it must be included in the request.
2. Unlike the policy for the service, the STS does not require a secondary token (though it should be noted that the Local Token Case scenario supported by the STS DOES require a secondary token).

With the above policy 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. Note that we will have to write some supporting code for the request, as the STS requires some specific addressing headers; this is covered later in this document.

### Configuring WS-SecurityPolicy for the Service

The reference code is written to call the HelloWorld service created in the previous chapter, so the WSDL exposed by that service is ready to use with the client. In the case that a client must be written for a service that does not expose a correct wsp:Policy section, the wsp:Policy section from the service in the previous chapter can be used. Simply copy the relevant parts of the policy section from the HelloWorld.wsdl file into the WSDL for the service that must be called.

## Configuring Keystores for the Client

Like the service, the client must be configured with 2 keystores. The first keystore contains the clients certificate and corresponding private key. The second keystore is a truststore, containing the public certificate of the STS as well as the service(s) that the client 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 the OCES CA certificate, and trust would then be established to all OCES certificates issued by that CA. Note that by having a keystore, that only contains the certificates of the services that the client is communicating with, we have a much better control over who the client trusts.

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

The XML configuration file for the client is called cxf-xml, and the relevant parts for configuring keystores is 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 the communication towards the service. 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 normally be configured to call an STS, without the need for additional code. Unfortunately the STSClient is not fully compliant with the NemLog-in STS, so we will have to write our own STSClient by extending the existing class.

The three issues are:

1. The NemLog-in STS requires that the @Context attribute on the RequestSecurityToken element contains a unique value for each request. The Apache CXF framework allows for configuring a static value.
2. The NemLog-in STS requires that the WS-SecurityPolicy namespace used in the AppliesTo element is the WS-SecurityPolicy 1.1 namespace. The Apache CXF framework uses a later version.
3. The NemLog-in STS requires some specific addressing headers. The STSClient implementation does not offer an easy way to modify the addressing headers, so an interceptor is implemented to deal with this.

The reference code contains a class called DigstSTSClient that includes the needed code to deal with these three issues. It is shown below:

public class DigstSTSClient **extends** STSClient **{**

public DigstSTSClient**(**Bus b**)** **{**

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

**this.**out**.**add**(new** STSAddressingInterceptor**());**

createUniqueContextAttribute**();**

**}**

@Override

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

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

createUniqueContextAttribute**();**

**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**();**

**}**

**}**

private void createUniqueContextAttribute**()** **{**

**this.**context **=** "urn:uuid:" **+** UUID**.**randomUUID**().**toString**();**

**}**

**}**

Note that the class depends on an Interceptor implementation that can add the expected addressing headers. The code for this interceptor is shown below

public class STSAddressingInterceptor **extends** AbstractSoapInterceptor **{**

public STSAddressingInterceptor**()** **{**

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

**}**

@Override

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

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

ObjectFactory wsAddressingFactory **=** **new** ObjectFactory**();**

AttributedURIType to **=** **new** AttributedURIType**();**

AttributedURIType messageId **=** **new** AttributedURIType**();**

AttributedURIType action **=** **new** AttributedURIType**();**

to**.**setValue**(**"https://signature.sts.nemlog-in.dk/"**);**

action**.**setValue**(**"http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue"**);**

messageId**.**setValue**(**"uuid:" **+** UUID**.**randomUUID**().**toString**());**

JAXBElement**<**AttributedURIType**>** createTo **=** wsAddressingFactory**.**createTo**(**to**);**

JAXBElement**<**AttributedURIType**>** msgId **=** wsAddressingFactory**.**createMessageID**(**messageId**);**

JAXBElement**<**AttributedURIType**>** newAction **=** wsAddressingFactory**.**createAction**(**action**);**

JAXBDataBinding jaxbDataBinding **=** **null;**

**try** **{**

jaxbDataBinding **=**

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

**}**

**catch** **(**JAXBException ex**)** **{**

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

**}**

Header toHeader **=** **new** Header**(**

**new** QName**(**"http://www.w3.org/2005/08/addressing"**),**

createTo**,** jaxbDataBinding**);**

Header msgIdHeader **=** **new** Header**(**

**new** QName**(**"http://www.w3.org/2005/08/addressing"**),**

msgId**,** jaxbDataBinding**);**

Header actionHeader **=** **new** Header**(**

**new** QName**(**"http://www.w3.org/2005/08/addressing"**),**

newAction**,** jaxbDataBinding**);**

headers**.**add**(**actionHeader**);**

headers**.**add**(**msgIdHeader**);**

headers**.**add**(**toHeader**);**

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

**}**

**}**

Note that the wsa:To header above is set to the value used by the STS in the Signature Scenario. The STS supports other scenarios, and this value should be changed if one of the other scenarios is used.

Finally, the DigstSTSClient class must be configured in the cxf.xml configuration file, which is done as shown below

<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.DigstSTSClient"**>

<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 client properties contain an entry for the value ws-security.sts.applies-to, this is needed to ensure that the STS knows which kind of token to issue, as it informs the STS which service the client intend to call with the token. In the reference code this contains the EntityId of the service created in the previous chapter.

Note that the STSClient configuration points to the WSDL file, which contains the wsp:Policy section that the STSClient must follow when making the request to the STS.

There is a set of property values that disables a range of settings, as the STSClient by default sends a lot of additional parameters to the STS (renewing, entropy, etc). As the STS does not support these parameters, they have been disabled.

Finally, the keystore values are configured, and note that the STS requires only property files and no callback handlers, though it does require two property files, one for the client certificate and another for the truststore.

## Support Classes

The client must support the sbf:Framework header, and for that purpose the same two supporting classes found in the service project is used in the client project.

FrameworkHeaderInterceptor is used to inject the Framework header in outgoing requests.

SbfFrameworkHeader is the model class that is marshalled into XML by the above interceptor.

## Additional Configuration of Apache CXF

### Configure BSP 1.1 Compliance

The Apache CXF framework enforces Basic Security Profile 1.1 compliance by default, which contains the following rule

**R5412**

Any SIG\_TRANSFORMS MUST contain as its last child a SIG\_TRANSFORM with an Algorithm attribute with a value of "http://www.w3.org/2001/10/xml-exc-c14n#" or "http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform" or "http://docs.oasis-open.org/wss/oasis-wss-SwAProfile-1.1#Attachment-Content-Signature-Transform" or "http://docs.oasis-open.org/wss/oasis-wss-SwAProfile-1.1#Attachment-Complete-Signature-Transform".

But the response from the STS does not conform to this rule, causing the CXF framework to reject the response. To avoid this, BSP 1.1 compliance must be disabled like this

<jaxws:client>

<jaxws:properties>

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

</jaxws:properties>

</jaxws:client>

### The Full cxf.xml ConfigurationFile

Besides the BSP 1.1 setting, no further configuration is needed to configure the Apache CXF framework. The full cxf.xml configuration file is shown below for completeness

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

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

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

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

xsi:schemaLocation=**"**

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

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

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

<cxf:bus>

<cxf:features>

<cxf:logging />

</cxf:features>

</cxf:bus>

<bean id=**"FrameworkHeaderInterceptor"**

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

<jaxws:client

name=**"{http://www.example.org/contract/HelloWorld}HelloWorldPort"**

createdFromAPI=**"true"**>

<jaxws:outInterceptors>

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

</jaxws:outInterceptors>

<jaxws:properties>

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

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

<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"** />

<entry key=**"ws-security.is-bsp-compliant"**

value=**"false"** />

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

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

<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>

</beans>

## Encrypted Assertion Workarounds

As with the Web Service Provider case, the client must be able to deal correctly with encrypted assertions. The changes in the client case is minimal, and is simply a matter of getting the correct element when the assertion is referenced.

This is done by modifying the same method as in the Web Service Provider case, though slightly simpler.

The method that has been overridden is displayed below, trimmed to display only the relevant sections of the method. The method itself can be found in the class STRTransform located in the ***wss4j-ws-security-dom*** module used by CXF (the same module as in issue 1).

private Data transformIt**(**Data data**,** XMLCryptoContext xc**,** OutputStream os**)** **{**

**...** snip **...**

**if** **(**dereferencedToken **!=** **null)** **{**

**...** snip **...**

**}**

// BEGIN DIGST WORKAROUND

boolean parentNodeIsNull **=** **(**dereferencedToken**.**getParentNode**()** **==** **null);**

boolean encryptedAssertionId **=**

**(**dereferencedToken**.**getAttribute**(**"wsu:Id"**) !=** **null** **&&**

dereferencedToken**.**getAttribute**(**"wsu:Id"**).**equals**(**"encryptedassertion"**));**

**if** **(!**parentNodeIsNull **&&** encryptedAssertionId**)** **{**

dereferencedToken **= (**Element**)**dereferencedToken**.**getParentNode**();**

**}**

// END DIGST WORKAROUND

//

// C14n with specified algorithm. According to WSS Specification.

//

buf **=** canon**.**canonicalizeSubtree**(**dereferencedToken**,** "#default"**,** **true);**

**...** snip **...**

**}**

This workaround has been applied to the digstclient version of the JAR that comes with the reference code.

## Using the Client

The reference code for the client consists of a single self-contained project, which is located in the folder “system-user-scenario”. The project can be compiled using Maven with the following command. Make sure that the command is issued from the directory that contains the pom.xml for this project

$ mvn clean install

This will compile the client project as a command-line application. The client can be executed by using the following Maven command

$ mvn exec:exec

This will execute the client, the implementation being located in the WSClient.java class. The class simply calls the method HelloWorld on the HelloWorld service, and the Apache CXF configuration will ensure that the STS is called first, and the result of the call to the STS cached, so the token is reused on the two following calls to the HelloWorld service.

It is assumed that the holder-of-key version of the HelloWorld service is running.

Also note that tracing is enabled on both the service and client projects, so the console will contain the full request and response payloads towards both the service and the STS.

## 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 WSClient.java source, the 2nd call to the service uses the cached token only because the same “port”-field is used.

# Bootstrap scenario

This chapter covers the steps necessary to combine the OIOSAML.Java framework with Apache CXF to build a Web Service Consumer that is also a SAML 2.0 Service Provider.

The purpose is to showcase the following scenario:

1. A user signs on to the Service Provider, through the NemLog-in SSO. The Service Provider is presented with a user-specific token (SAML 2.0 Assertion) from the NemLog-in SSO.
2. The Service Provider, acting as a Web Service Consumer, calls the NemLog-in STS to get a token, where a bootstrap Assertion embedded in the Web SSO Assertion is presented as part of the request.
3. The NemLog-in STS issues a token to the Service Provider (aka WSC), containing the user’s identity.
4. The Service Provider then calls Web Service Provider from chapter 2 using this token

## Reference Code

The code for the Bootstrap Scenario is found in the folder “bootstrap-scenario” in the root of the reference code distribution. The project is self-contained, and does not directly depend on any of the other modules in the reference code distribution (to test the code, it does require the service found in the folder “service-hok” to be running).

## Re-use from chapter 3 (The System User Scenario)

The Bootstrap Scenario is very similar (with regards to CXF) to the previous System User Scenario, and only a few minimal changes are required to modify the existing code to work with this scenario. These changes are listed below

1. The WSClient class is no longer needed

The WSClient class is the “main” class in the System User Scenario, and will be replaced by a webpage that drives the flow, so it is deleted from this module.

1. STSAdressingInterceptor must be modified

This class is responsible for setting the wsa:To header field in the soap call to the STS, and since the Bootstrap Scenario requires a different wsa:To header value, this line needs to be modified to

to**.**setValue**(**"https://bootstrap.sts.nemlog-in.dk/"**);**

1. DigstSTSClient needs to be extended

The DigstSTSClient is the class that is responsible for calling the STS to get a token. As the major difference between the Bootstrap Scenario and the System User Scenario is how the token is retrieved from the STS, this is where the largest modification is required.

We need to set the ActAs value to the request, and the easiest way to do this, is to override the issue() method found in the STSClient base class.

@Override

protected STSResponse issue**(**String appliesTo**,** String action**,**

String requestType**,** String binExc**)** **throws** Exception **{**

UserAssertion userAssertion **=** UserAssertionHolder**.**get**();**

UserAttribute attribute **=** userAssertion**.**getAttribute**(**

"urn:liberty:disco:2006-08:DiscoveryEPR"**);**

byte**[]** rawAsrt **=** Base64**.**decodeBase64**(**attribute**.**getValue**());**

setActAs**(new** String**(**rawAsrt**,** Charset**.**forName**(**"UTF-8"**)));**

**return** **super.**issue**(**appliesTo**,** action**,** requestType**,** binExc**);**

**}**

The above code uses the OIOSAML UserAssertionHolder class to get access to the UserAssertion that was supplied by the end-user when he signed on to the web-application. From this class we can extract the “urn:liberty:disco:2006-08:DiscoveryEPR” attribute, which contains a base64 encoded SAML 2.0 Assertion (bootstrap token), which we can supply to the STS as an ActAs argument in the request for a token.

With the above modifications, the CXF code is ready to be used in conjunction with OIOSAML.Java, which will be responsible for supplying the UserAssertion element.

## Getting CXF and OIOSAML.Java to co-exist

Disclaimer: This document is not intended as a guide on how to use OIOSAML.Java, nor will it go into details about best practices when it comes to using that framework. It will cover the necessary steps to get OIOSAML.Java to peacefully co-exist with Apache CXF, and the reference code contains an integration of OIOSAML.Java.

To reduce custom configuration of OIOSAML.Java, the reference code uses the demo service provider project found in the OIOSAML.Java distribution, this constitutes the following resource files from the demo project

├── oiosaml-config

│   ├── oiosaml-sp.log4j.xml

│   └── oiosaml-sp.properties

├── src

│   └── main

│   └── webapp

│   ├── configure.jsp

│   ├── head.jsp

│   ├── index.jsp

│   ├── oiosaml.gif

│   ├── postlogin.jsp

│   ├── sp

│   │   ├── logout.jsp

│   │   ├── post.jsp

│   │   ├── priv1.jsp

│   │   └── query.jsp

│   └── WEB-INF

│   ├── classes

│   │   └── log4j.properties

│   └── web.xml

As well as the OIOSAML.Java library itself. As the OIOSAML.Java library is distributed as a stand-alone Jar file, with documented dependencies on external libraries like OpenSAML, a maven dependency file (pom) has been created for easy integration into Maven based build projects. The OIOSAML.Java Jar file and corresponding pom file are located in the “libs” folder like this

├── libs

│   └── dk

│   └── itst

│   └── oiosaml.java

│   └── 21188

│   ├── oiosaml.java-21188.jar

│   ├── oiosaml.java-21188.jar.sha1

│   ├── oiosaml.java-21188.pom

│   └── oiosaml.java-21188.pom.sha1

Adding both the CXF and OIOSAML.Java dependencies to the Bootstrap Scenarios Maven pom file results in only a single conflict. The two frameworks disagree on which version of xmlsec to use. CXF requires the highest version of this library (version 2.0.4), and OIOSAML.Java can use this version without any issues, so a direct dependency to this specific version of xmlsec has been added to the project.

### Integrating with NemLog-in SSO

Integration with the NemLog-in SSO is covered by the NemLog-in documentation, and is not covered in any detail in this document[[3]](#footnote-4). The reference code comes with a working integration, which constitutes the following files in the reference code

├── oiosaml-config

│   ├── certificate

│   │   └── keystore

│   ├── metadata

│   │   ├── IdP

│   │   │   └── IdPMetadata.xml

│   │   └── SP

│   │   └── SPMetadata.xml

├── ssl-keystore

1. The keystore file is a FOCES certificate, and it is a requirement that this FOCES certificate is the same certificate as the one used to call the STS (which CXF requires is located in /src/main/resources). The client.jks file located here is an exact copy of the keystore file in the oiosaml-config/certificate folder
2. The IdPMetadata.xml file is copied from the NemLog-in webpage, and contains the metadata for the TEST environment of the NemLog-in SSO.
3. The SPMetadata.xml is generated by the OIOSAML.Java library, and trust towards this set of metadata has been established in the NemLog-in SSO
4. As the NemLog-in SSO requires all Service Providers to use TLS/SSL, a self-signed SSL certificate has been generated, and it is located in the ssl-keystore file.

As the SPMetadata.xml file contains the URLs that the NemLog-in SSO must send SAML-Responses to, a fully qualified URL has been chosen for this purpose.

<https://cxf-sp:8095/cxf-sp-ws-consumer/>

Please note that for the bootstrap scenario to work, the metadata that is exchanged with the NemLog-in SSO must contain the following RequestedAttribute element (it triggers issuance of bootstrap tokens):

<md:RequestedAttribute Name=**"urn:liberty:disco:2006-08:DiscoveryEPR"**

NameFormat=**"urn:oasis:names:tc:SAML:2.0:attrname-format:basic"**

isRequired=**"false"**/>

The metadata in the reference source has this property set. Without it the NemLog-in SSO will not return the attribute we need to extract a valid bootstrap token.

### Getting a token and calling a service

In the System User Scenario, we had a main method in the WSClient class that was responsible for executing the flow, but since we deleted that class, we need some other way to trigger the flow.

After a successful login, the end-user is presented with the webpage priv1.jsp. In the reference code, this JSP page has been modified to contain a link to a new page (call\_service.jsp). This new JSP page contains the code from WSClient, and accessing it will force CXF to call the STS to get a token, and then use the token to call the configured service.

**<%@**page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"**%>**

**<%@**page import="org.example.contract.helloworld.HelloWorldPortType"**%>**

**<%@**page import="org.example.contract.helloworld.HelloWorldService"**%>**

**<%**

HelloWorldService service = **new** HelloWorldService();

HelloWorldPortType port = service.getHelloWorldPort();

**String** serviceResponse = port.helloWorld("John");

**%>**

<jsp:include page="/head.jsp" />

<div>

<h1>**Result from calling service**</h1>

<span>

**<%** out.println(serviceResponse); **%>**

</span>

</div>

</body>

</html>

It is not a requirement to put the code inside the JSP page, nor is it a requirement to use JSP. It is simply convenient as the OIOSAML.Java demo project already uses JSP. The same code could be placed in a Java controller class.

## Using the reference code

The reference code for the combined Service Provider and Web Service Consumer resides in the “bootstrap-scenario” folder, and is a web-application that can be deployed either as a WAR file in a running Tomcat, or stand-alone using the Maven Tomcat Plugin as described below.

Please note that it comes preconfigured with working SAML metadata, which assumes that the Service Provider will be running on the following address

<https://cxf-sp:8095/cxf-sp-ws-consumer/>

A local DNS entry must be made for the domain name cxf-sp, so it points to 127.0.0.1 (if it is running locally, adjust for the IP address of the server that the application is deployed to).

**Linux/Unix variants (including OS X) instructions**

Add the following line to the /etc/hosts file (the file must be modified as the root user)

127.0.0.1 cxf-sp

**Windows instructions**

Add the following line to the c:\windows\system32\drivers\etc\hosts file (the file must be modified as the Windows Administrator)

127.0.0.1 cxf-sp

The project can be compiled using Maven with the following command. Make sure that the command is issued from the directory that contains the pom.xml for the “bootstrap-scenario” project

$ mvn clean install

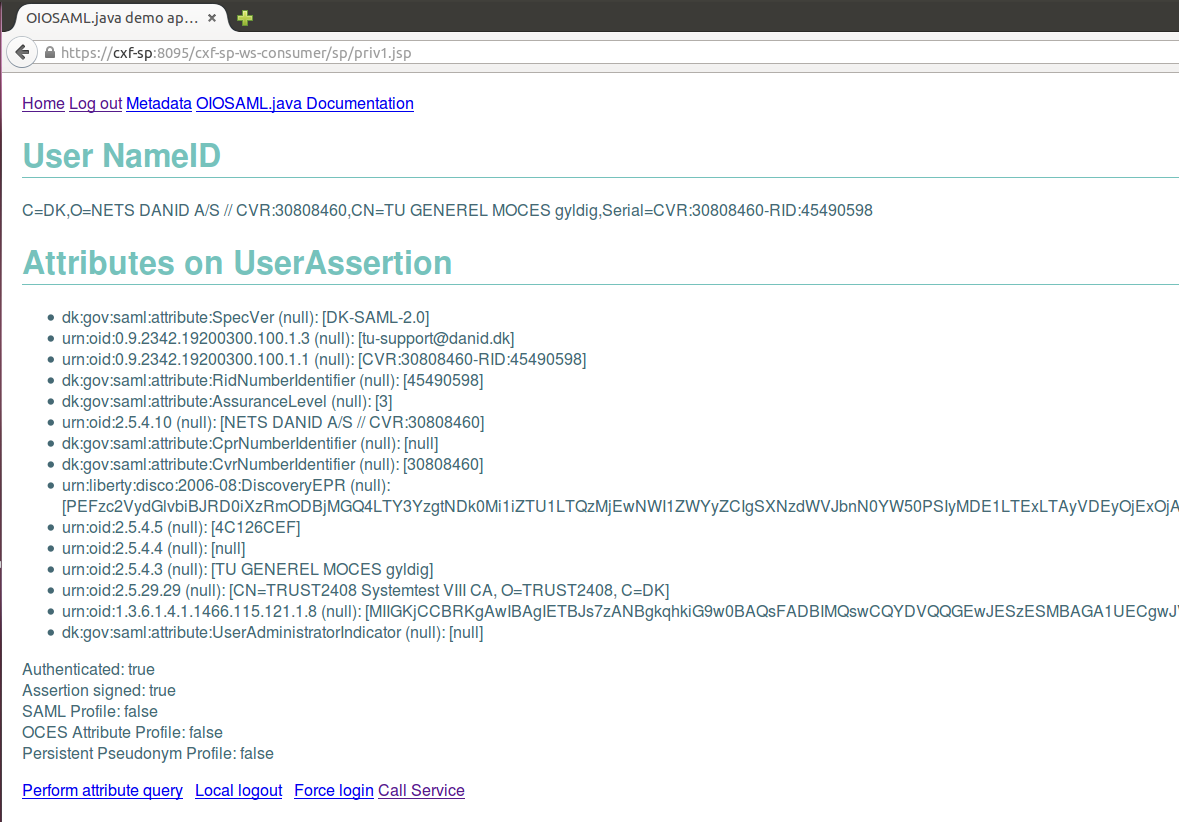
This will compile the project as a WAR file ready for deployment. The project can be started by using the following Maven command

$ mvn tomcat7:run-war

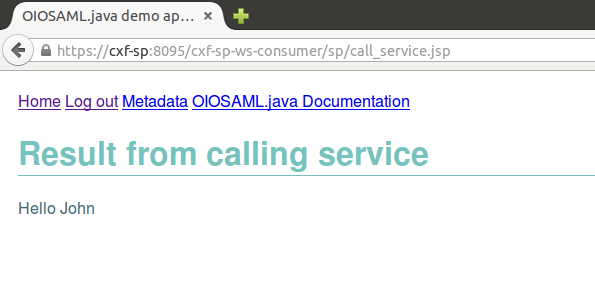
1. This will start the project running on port https/8095, using a self-signed SSL certificate. Access the running application by opening at browser and going to

<https://cxf-sp:8095/cxf-sp-ws-consumer/>

1. Once the page has been accessed, click on the link “Page requiring login”, and perform a login using the NemLog-in SSO. A valid MOCES certificate has been supplied with the reference code, it resides in root folder and is called test-moces.pfx. The password for the file is Test1234.
2. After a successful login, the following screen is presented in the browser. Click on the link “Call Service” to start the flow that calls the STS for a token and uses the token to call the Web Service Provider



1. After a short period, the following success screen should be displayed in the browser



# Signature scenario

This chapter covers the steps necessary to build a desktop/GUI application, that can retrieve a user-specific token from the STS, and use it as a bearer-token when calling a web service.

The purpose is to showcase the following scenario

1. An end-user starts a desktop application.
2. The end-user triggers some event, that requires the desktop application to call a service.
3. The desktop application prompts the end-user for a valid MOCES keystore (and corresponding password).
4. The desktop application calls the STS (on behalf of the user), using the supplied MOCES certificate to sign the request and get back a bearer-token.
5. The desktop application calls the service, supplying the bearer-token, and signs the request with the applications own FOCES certificate.

## Reference Code

The code for the Signature Scenario is found in the folder “signature-scenario” in the root of the reference code distribution. The project is self-contained, and does not directly depend on any of the other modules in the reference code distribution (to test the code, it does require the service found in the folder “service-bearer” to be running).

## Re-use from chapter 3 (The System User Scenario)

The Signature Scenario is, like the Bootstrap Scenario, very similar (with regards to CXF) to the System User Scenario, and only a few minimal changes are required to modify the existing code to work with this scenario. These changes are listed below

1. The WSClient class is no longer needed

The WSClient class is the “main” class in the System User Scenario, and will be replaced by a desktop GUI that drives the flow, so it is deleted from this module.

1. The keystore required to call the STS is supplied at runtime

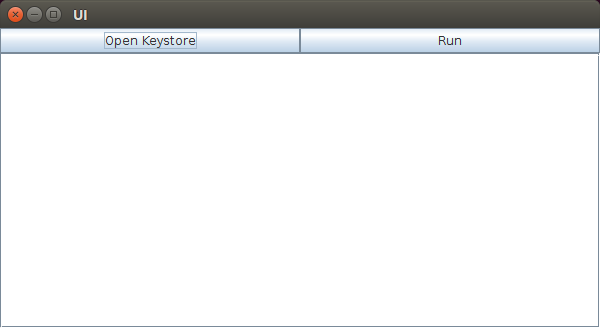
The System User Scenario has a very simple approach to handling keystores, as they are pre-configured in a property file. In this scenario we need to write some code that deals with supplying the keystore at runtime.

1. The client will call a different service

The client is configured to call the bearer-token version of the Hello World service, and will use the WSDL for that service instead of the holder-of-key version used in the System User Scenario

## The GUI for the desktop application

This document is not a tutorial in writing desktop applications, and for the purpose of implementing this scenario, an extremely simple Java Swing application has been written, containing two buttons and a textarea to write log-output to. A screenshot is shown below.



Here the user must first press the “Open Keystore” button to supply the MOCES certificate, and then click the “Run” button to trigger the call to the STS and the web service.

## Dealing with the user supplied keystore

Apache CXF requires three pieces of information about a keystore for it to work

1. The actual keystore
2. The password for the keystore
3. The alias of the certificate in the keystore to use

### Providing the keystore

Apache CXF by default uses the following class for dealing with keystores

org.apache.wss4j.common.crypto.Merlin

and we can configure CXF to use a different class. We will use this feature by extending the above Merlin class, and making the required modifications to this new class.

The keystore handler class is created by CXF when it is needed (e.g. at the time when CXF is calling the STS), so we can write a very simply extension of the class, that just overrides the keystore in the Merlin class with the user supplied keystore when the class is created

public class UserCertificateStore **extends** Merlin **{**

private static KeyStore ks**;**

public UserCertificateStore**(**Properties properties**,**

ClassLoader loader**,**

PasswordEncryptor passwordEncryptor**)**

**throws** WSSecurityException**,** IOException **{**

**super(**properties**,** loader**,** passwordEncryptor**);**

**this.**setKeyStore**(**UserCertificateStore**.**ks**);**

**}**

public static void setKs**(**KeyStore keyStore**)** **{**

ks **=** keyStore**;**

**}**

**}**

When the user clicks the button “Open Keystore” and supplies a keystore, we must simply call the static setKs method on the above class, and later when CXF needs an instance of this class, it will automatically be supplied with the users keystore.

To configure CXF with this new class, we need to create a property file, and then point to this property file in the cxf.xml configuration file. The content of the property file (called user.properties in the reference code) should be the following

org.apache.ws.security.crypto.provider=client.crypto.UserCertificateStore

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

The first line tells CXF to use our custom class for dealing with keystores, and the following three lines configures a trust-store for the interactions with the STS. The trust.jks truststore contains the certificate used by the STS.

Note that unlike the System User Scenario, we are not providing any keystore for the client here, as the UserCertificateStore is responsible for supplying it on runtime.

### Providing the password

The ClientCallbackHandler class from the System User Scenario was responsible for providing the password when needed. It had a hardcoded password (“Test1234”), which we will replace with a user-supplied password

public class ClientCallbackHandler **implements** CallbackHandler **{**

private static String password**;**

@Override

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

UnsupportedCallbackException **{**

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

**if** **(**callbacks**[**i**]** **instanceof** WSPasswordCallback**)** **{**

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

**if** **(**pc**.**getUsage**()** **==** WSPasswordCallback**.**DECRYPT **||**

pc**.**getUsage**()** **==** WSPasswordCallback**.**SIGNATURE**)** **{**

pc**.**setPassword**(**password**);**

**}**

**}**

**}**

**}**

public static void setPassword**(**String password**)** **{**

ClientCallbackHandler**.**password **=** password**;**

**}**

**}**

When the user clicks on the “Open Keystore” button and supplies a keystore, the user is also prompted for a password, which will be passed to the new setPassword method in the ClientCallbackHandler.

### Providing the alias

While it would be possible to scan the keystore for all entries, and list them for the user, so he or she can pick which certificates to use, the reference code makes the assumption that the user is happy if we just use the first entry in the keystore.

The STSClient is the class that requires the alias, so we let the UI extract the alias when the keystore is supplied, and then set the alias on the STSClient implementation. As the STSClient is already extended by our custom DigstSTSClient class, we only need to add a single method

public class DigstSTSClient **extends** STSClient **{**

**...** existing code

public void setAlias**(**String alias**)** **{**

ctx**.**put**(**SecurityConstants**.**SIGNATURE\_USERNAME**,** alias**);**

**}**

**}**

Unlike with the keystore and the password, we cannot solve this with a static method, so we need to access the exact instance of this class that CXF is using. As the DigstSTSClient is defined as a bean in cxf.xml, we can access it using the ApplicationContext. A simple helper class is created for this purpose, and used by the UI classes.

public class ApplicationContextProvider

**implements** ApplicationContextAware **{**

private static ApplicationContext ctx**;**

@Override

public void setApplicationContext**(**ApplicationContext appContext**)**

**throws** BeansException **{**

ApplicationContextProvider**.**ctx **=** appContext**;**

**}**

public static ApplicationContext getApplicationContext**()** **{**

**return** ctx**;**

**}**

**}**

### Interaction with the UI

The entire UI is constructed from three classes, all found in the client.ui package. The first is the UI class itself, which is constructs the UI panel and adds the buttons, the other two classes are the handlers, that deals with what happens when the two buttons are clicked.

The KeystoreButtonHandler simply asks the user to supply a file and a password, and no further action is taken.

The RunButtonHandler performs all of the interesting stuff, as shown below

public class RunButtonHandler **implements** ActionListener **{**

private UI ui**;**

public RunButtonHandler**(**UI ui**)** **{**

**this.**ui **=** ui**;**

**}**

@Override

public void actionPerformed**(**ActionEvent e**)** **{**

// set the password on the password callback handler

ClientCallbackHandler**.**setPassword**(**ui**.**getPassword**());**

// set the keystore on the UserCertificateStore

KeyStore keystore **=** getKeystore**();**

UserCertificateStore**.**setKs**(**keystore**);**

// set the alias on the STSClient

String alias **=** keystore**.**aliases**().**nextElement**();**

DigstSTSClient stsClient **=** ApplicationContextProvider

**.**getApplicationContext**().**getBean**(**DigstSTSClient**.**class**);**

stsClient**.**setAlias**(**alias**);**

// call the service a couple of times

ui**.**getPort**().**helloWorld**(**"John"**);**

ui**.**getPort**().**helloWorld**(**"Jane"**);**

**}**

private KeyStore getKeystore**()** **{**

KeyStore ks **=** KeyStore**.**getInstance**(**"PKCS12"**);**

ks**.**load**(new** FileInputStream**(**ui**.**getFile**()),**

ui**.**getPassword**().**toCharArray**());**

**return** ks**;**

**}**

**}**

## Using the reference code

The reference code for Signature Scenario is in the “signature-scenario” folder, and is a desktop application that can be started as described below.

The project can be compiled using Maven with the following command. Make sure that the command is issued from the directory that contains the pom.xml for the “signature-scenario” project

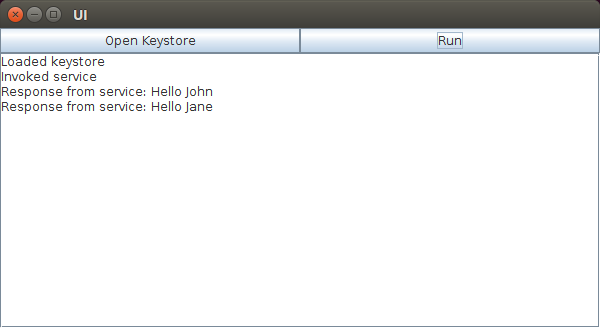
$ mvn clean install

The project can be started by using the following Maven command

$ mvn exec:exec

This will start the desktop application. To test the application follow these steps

1. Make sure the bearer-token version of the service is running.
2. Click the button “Open Keystore” and supply the test-moces.pfx file found in the root of the reference code folder. The password for the file is “Test1234”.
3. Click on the “Run” button. CXF will print a trace of the communication to the console, and the UI will show the result of the actual web service call in the log panel of the UI.
4. A successful run should result in the following



# OIO IDWS Profile 1.1

The Liberty Basic SOAP Binding (LBSB) profile has been replaced by the Danish profile OIO IDWS Profile version 1.1, which is at its core a revision of LBSB.

The major changes are

* Requires a secure transport mechanism (i.e. TLS 1.2 or better)
* Is based on SOAP 1.2 instead of SOAP 1.1
* No longer requires the use of the Framework SOAP header
* Does not govern the use of SOAP Faults

The change to SOAP version, and the omission of the Framework header will affect service contract (WSDL) of a service that migrates from LBSB to OIOIDWS, so please keep this in mind when upgrading.

The code that accompanies this document implements the scenarios mentioned in the previous chapters, but the code is split into two sub-folders

* lbsb-scenarios
* oioidws-scenarios

which contains the same scenario implementations, just with different security profile implementations. This chapter will outline the modifications made to the code in the oioidws-scenarios folder, compared to the original code in the lbsb-scenarios folder.

This also means that this chapter can be read as a migration guide for moving from LBSB to OIOIDWS.

## Update to SOAP 1.2

The example code is written as contract-first, so the WSDL governs how CXF performs SOAP calls, including the version of SOAP used.

The following change was made to all the WSDL files used in the example code

Replace this

xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"

with this

xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap12/"

We also need to tell the WSP implementation about this, so add this @BindingType annotation on top of the Java class (HelloWorldPortType.java) that implements the service endpoint (in the service-hok and service-bearer projects)

@BindingType(value = javax.xml.ws.soap.SOAPBinding.SOAP12HTTP\_BINDING)

public class HelloWorldPortTypeImpl implements HelloWorldPortType {

No changes are required on the WSC, it should just be pointed at the updated WSDL file, and it will start using SOAP 1.2.

## Configuring a secure transport mechanism (TLS 1.2)

While existing LBSB implementations in production will already use a secure transport mechanism (SSL/TLS), the LBSB example code run on HTTP without any transport layer security.

As the OIO IDWS Profile 1.1 mandates the use of a secure transport mechanism, the example code has been updated to illustrate how this could be done using Tomcat.

In a production setting, this is likely handled by whatever applicationserver, proxy or loadbalancer that hosts the application.

**The following changes were made to the WSP projects**

1. An SSL certificate was added to the project

A self-signed certificate was used, stored in the keystore filed named ‘ssl-keystore’.

1. The Maven plugin in the pom.xml was configured to use the keystore

<configuration>

<port>0</port>

<httpsPort>8443</httpsPort>

<keystoreFile>${project.basedir}/ssl-keystore</keystoreFile>

<keystorePass>Test1234</keystorePass>

</configuration>

Note that the Tomcat plugin used by Maven does not allow configuring the TLS version, but this is doable by setting the following properties directly in Tomcat in a production setting

<sslEnabledProtocols>TLSv1.2</sslEnabledProtocols>

<sslProtocol>TLSv1.2</sslProtocol>

1. The endpoint in the WSDL file was updated to the following value to reflect the changed Tomcat settings

<soap:address location="https://localhost:8443/HelloWorld/services/helloworld" />

**The following changes were made to the WSC projects**

The WSC projects need to use the updated WSDL, so they know about the changes to the address, and then they need to trust the SSL certificate used by the WSP.

In a production setting, the SSL certificate used will likely be trusted by the WSC, but as the example code uses a self-signed certificate, a trust-store called ‘ssl-trust.jks’ is added to the WSC projects, which contains the SSL certificate used by the WSP.

The following hardcoded trust is then added to the startup of the code (not recommended for production settings – use the application-servers truststore or used SSL certificates issued by a trusted CA)

public static void main (String[] args) {

System.setProperty("javax.net.ssl.trustStore",

"src/main/resources/ssl-trust.jks");

System.setProperty("javax.net.ssl.trustStorePassword",

"Test1234");

// … the rest of the code

}

This will ensure that the WSC can establish an SSL connection to the WSP without complaining about not trusting the certificate.

## Remove Framework header

As the Framework header from the LBSB profile is not a part of the OIOIDWS profile, we simply remove the configuration and code that deals with this header from both the WSC and WSP

**The following changes are made to the WSP**

The following Java classes are removed from the codebase

* FrameworkHeaderInterceptor
* UnderstandFrameworkHeaderInterceptor
* SbFrameworkHeader

The first two classes are responsible for adding the Framework header on responses, and validating the Framework header on requests, and the last is just the model class for marshelling to/from XML.

As we have removed the classes, we need to remove them from the cxf-servlet.xml file, where they are configured to be used – so remove these interceptor configuration sections

<cxf:inInterceptors>

<ref bean="UnderstandFrameworkHeaderInterceptor" />

</cxf:inInterceptors>

<cxf:outInterceptors>

<ref bean="FrameworkHeaderInterceptor" />

</cxf:outInterceptors>

<cxf:outFaultInterceptors>

<ref bean="FrameworkHeaderInterceptor" />

</cxf:outFaultInterceptors>

And these two bean definitions

<bean id="FrameworkHeaderInterceptor"

class="service.interceptor.FrameworkHeaderInterceptor" />

<bean id="UnderstandFrameworkHeaderInterceptor"

class="service.interceptor.UnderstandFrameworkHeaderInterceptor" />

**The following changes are made to the WSC**

Similar changes are made to the WSC projects, where the following Java classes are deleted

* FrameworkHeaderInterceptor
* SbFrameworkHeader

As they are used for setting the Framework header in the SOAP requests.

And the cxf.xml configuration file is updated, by removing these interceptors and beans in a similar fashion as the updates performed on the WSP

<jaxws:outInterceptors>

<ref bean="FrameworkHeaderInterceptor" />

</jaxws:outInterceptors>

<bean id="FrameworkHeaderInterceptor"

class="service.interceptor.FrameworkHeaderInterceptor" />

## Changes for interoperability with .NET reference code

During interoperability testing with the .NET reference code for OIO IDWS, a few changes were made to the WS-SecurityPolicy section of the WSDL for the HelloWorld service.

These changes make it possible for the .NET frameworks SVCUTIL tool to parse the policy section of the WSDL without errors, without changing the behaviour of the Java reference implementations.

These changes are as follows

1. Declare the following namespace in the top of the WSDL

xmlns:wsap="http://www.w3.org/2006/05/addressing/wsdl"

1. Change the <InitiatorToken> specification

The SVCUTIL from the .NET framework does not understand the <SamlToken> element, so it is replaced with a <Issuedtoken> element, with a <RequestSecurityTokenTemplate> that sets the type of the issued token to SAML

The original configuration is changed from

<sp:InitiatorToken>

<wsp:Policy>

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

<wsp:Policy>

<sp:WssSamlV20Token11 />

</wsp:Policy>

</sp:SamlToken>

</wsp:Policy>

</sp:InitiatorToken>

To the following value

<sp:InitiatorToken>

<wsp:Policy>

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

<sp:RequestSecurityTokenTemplate>

<wsap:KeyType>

http://docs.oasis-open.org/ws-sx/ws-trust/200512/Symmetric

</wsap:KeyType>

<wsap:TokenType>

http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0

</wsap:TokenType>

</sp:RequestSecurityTokenTemplate>

<wsp:Policy/>

</sp:IssuedToken>

</wsp:Policy>

</sp:InitiatorToken>

1. Change the <SignedSupportingTokens> specification

In a similar way, the SignedSupportingTokens element is updated, so the SVCUTIL tool can parse the policy, this is done by replacing this value

<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>

<sp:WssSamlV20Token11 />

</wsp:Policy>

</sp:IssuedToken>

</wsp:Policy>

</sp:SignedSupportingTokens>

With this value

<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>

<wsap:KeyType>

http://docs.oasis-open.org/ws-sx/ws-trust/200512/Symmetric

</wsap:KeyType>

<wsap:TokenType>

http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0

</wsap:TokenType>

</sp:RequestSecurityTokenTemplate>

<wsp:Policy/>

</sp:IssuedToken>

</wsp:Policy>

</sp:SignedSupportingTokens>

## Removed SOAP Fault example

As LBSB also covers SOAP Faults, the LBSB example code includes an example on how to deal with SOAP Faults. OIOIDWS does not govern how SOAP Faults are handled, so this is an implementation detail that is left up to the individual it-solution.

To not give the impression that SOAP Faults must be handled in a specific way, the OIOIDWS example code is stripped of SOAP Fault handling.

When migrating an existing LBSB based solution to OIOIDWS, it is safe to keep whatever methods are used for dealing with SOAP Faults, as OIOIDWS does not set any requirements.

# Example payloads

The reference code has trace logging enabled on both the client and the service, so by running the reference code, it is possibly to recreate example payloads mentioned in this chapter.

Full traces can be found in the “traces” folder inside the “doc” folder (note there are separate folders for the OIOIDWS and LBSB cases). There are separate folders for each scenario, with a full trace for each request and corresponding response.

The following files are available for inspection

├── bootstrap-scenario

│   ├── 1-WSC-TO-STS.XML

│   ├── 2-STS-TO-WSC.XML

│   ├── 3-WSC-TO-WSP.XML

│   └── 4-WSP-TO-WSC.XML

├── signature-scenario

│   ├── 1-WSC-TO-STS.XML

│   ├── 2-STS-TO-WSC.XML

│   ├── 3-WSC-TO-WSP.XML

│   └── 4-WSP-TO-WSC.XML

└── system-user-scenario

├── 1-WSC-TO-STS.XML

├── 2-STS-TO-WSC.XML

├── 3-WSC-TO-WSP.XML

└── 4-WSP-TO-WSC.XML

In each scenario, 4 xml files are available.

## WSC-TO-STS

These xml files contain the requests to the STS, and the main difference between the files is the contents of the soap-body. The soap-headers are almost identical, differing only in the wsa:To field, that decides which scenario on the STS is being called.

The soap-body for the bootstrap scenario contains an ActAs element not found in the other scenarios, and in the signature scenario, the request is signed with a MOCES certificate rather than a FOCES as in the other scenarios.

## STS-TO-WSC

These xml files contains the responses from the STS, and are for all practical purposes identical, as the Assertion element is encrypted, and it is the content of the Assertion element that differs.

## WSC-TO-WSP

These xml files contains the requests to the web service provider, and have identical soap-bodies – the only difference lies in the soap-headers, where the signature scenario differs from the bootstrap and system user scenarios.

The signature scenario uses a bearer-token, hence the request contains a BinarySecurityToken element, containing the WSC’s x509 certificate, which is not present in the other scenarios.

## WSP-TO-WSC

These xml files contains the responses from the web service provider, and are identical across the scenarios.

# Summary

This document has covered the steps needed to take an existing Apache CXF based web service, and secure it so clients must present a valid token from the NemLog-in STS.

The document is bundled with reference code that has implemented these steps on a very simple web service using WS-SecurityPolicy. The reference code can potentially be used as a template when creating a new web service from scratch, or simply as inspiration when modifying an existing web service.

In the same way, the document has covered the steps needed to create several different clients, also using Apache CXF, to call a secured service in the correct way, including how to get a token from the STS. As with the service, reference code is available that can be used either as a template for creating clients for other services, or as inspiration for modifying existing clients.

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

# 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.

# References

[LIBERTY] Liberty Basic SOAP Binding v 1.0

<https://digitaliser.dk/resource/414852>

[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>

[OIO-BPP] OIOSAML Basic Privilege Profile 1.0

<https://digitaliser.dk/resource/2377872>

[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>

[OIOIDWS] OIO IDWS SOAP Profile 1.1

<https://digitaliser.dk/resource/3457606>

1. https://administration.nemlog-in.dk [↑](#footnote-ref-2)
2. https://digitaliser.dk/group/2848479 [↑](#footnote-ref-3)
3. https://digitaliser.dk/resource/2598479 [↑](#footnote-ref-4)