Introduction

This document provides a definition of the sample SAML application provided by Digital Insight as a reference implementation for software developers at any Financial Institution or Third Party Vendor who are integrating custom content into their online and mobile applications with Digital Insight products. Integration with Digital Insight requires that the partner adhere to SAML (Secure Access Markup Language) SSO (single signon) standards. This document provides usage instructions for a sample SAML application and its associated test driver, so that any partner can refer to the SAML processing used within the sample application and exercise any SAML end points using the test driver without the need for Digital Insight infrastructure.

For further details on the Digital Insight SAML Integration process, please refer to DI_SAML_SSO_Integration_Guide_v2.0.doc. This document does not attempt to reproduce the content specified within the integration guide and familiarity with the integration guide is assumed.

Sample Application

Overview

The sample application is a minimal example of how a third party app would communicate with Digital Insight's internet banking platform. The implementation has been tested against Di's PingFed server and is known to be compatible with it. In a Service Provider/Identity Provider system, this app is the former.

- The app will expose an endpoint that listens for incoming SAML message POSTs
- Upon receiving such a message, the application will use it to attempt to authenticate the user sending the POST
- If the SAML authentication is rejected the user will be redirected to some error page (either a session timeout page or a general login error page) and access to the site is restricted
- Otherwise the app will create a user session, granting the user due access up till some expiration date
- It then extracts user/auxiliary data contained in the <AttributeStatement> element of the SAML message
- The user will be redirected to a results page where the user data is displayed
- The results page will send regular ajax requests to the keep-alive URL provided by the SAML <AttributeStatement>
- If the user sends a request to the logout endpoint the user session will be invalidated and access to the site is once again restricted

SAML Validation

The sample app uses SAML to authenticate users according to the unsolicited response scenario; the sample app does not actively request any authentication from the IdP (Identity Provider). The flow is instead initiated by the IdP, which upon the end user’s request sends a SAML Response with all necessary authentication data to the sample app. This scenario is true to the actual FMIS flow, where a user action on the USP
will be what initiates any interaction with a TPV, causing FMIS to initiate the SSO sequence.

The sample app will listen on the `https://server/contextPath/validate` address for incoming SAML messages, expecting them to be submitted in `base64-encoding` via the form post parameter `SAMLResponse`. The app subsequently attempts to:

- Decode the base64 value from the SAMLResponse parameter into a string representation of an XML message
- Parse the XML message
- Marshal the XML into an OpenSAML Response object

If these steps succeed a validation process will commence, checking the SAML message against a set of criteria stipulated in the SAML specs. There are three main reference documents regarding this:

<table>
<thead>
<tr>
<th>Document</th>
<th>URL</th>
<th>Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAML 2.0 Core</td>
<td><a href="http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf">http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf</a></td>
<td>The SAML assertions and protocols specification defines the SAML assertions and request-response messages themselves.</td>
</tr>
<tr>
<td>SAML 2.0 Profiles</td>
<td><a href="http://docs.oasis-open.org/security/saml/v2.0/saml-profiles-2.0-os.pdf">http://docs.oasis-open.org/security/saml/v2.0/saml-profiles-2.0-os.pdf</a></td>
<td>The SAML profiles specification [SAMLProfile] defines specific usage patterns that reference both [SAMLCore] and bindings defined in this specification or elsewhere.</td>
</tr>
</tbody>
</table>

The sample app implements the Web Browser SSO profile (with unsolicited responses) and employs the HTTP POST binding, in addition to conforming to the Core specs. If the validation process passes, a Principal object with information about authentication status and expiration date will be attached to the user's session data.

**Assumptions**

When working with SAML authentications, the sample app operates under the following assumptions:

- The SAML message must be a well formed XML, following the SAML protocol .xsd schema (found [here](#)). Failure to comply with these constraints will cause the app to throw top level exceptions.
  - This means, among other things, that the root element must be `<samlp:Response>`, and not `<saml:Assertion>`.
- The attributes `Response.Destination` and `Response>Assertion>Subject>SubjectConfirmation>SubjectConfirmation>Data.Recipient` must both match the endpoint receiving the message.
- The values of the elements `Response>Issuer` and `Response>Assertion>Issuer` must both match the `ISSUER_ENTITY_ID` constant found in `com.ncrcoe.di.saml.validation.constants.SAMLConstants`.
- In the same class the constant `SERVICE_PROVIDER_URL` is declared. This must be present in the list of audiences located at `Response>Assertion>Conditions>AudienceRestrictions`.
- While praxis seems to be that each `Response` only carries a single `Assertion`, the sample app should work fine even if this isn't the case.

**Validators**

The above documents can be fairly daunting and an attempt has been made to divide the validation process into manageable, intuitive chunks. The details of this solution may be considered too low level for this document. If you're reading this to get a quick overview of the product you can skip this part, and needn't feel guilty about doing so.

△ More on the structuring of the validators

**Validator structure**

Most of the validators (with the notable exception of the Assertions Core validator) verify a subset of the constraints given by some section of the profile specifications. Thus, some sections are divided up between several validators. The relevant profile sections are the following:

<table>
<thead>
<tr>
<th>Section</th>
<th>Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.4.2</td>
<td><code>&lt;Response&gt;</code> Usage</td>
<td>Quite possibly not needed (?)</td>
</tr>
<tr>
<td>4.1.4.3</td>
<td><code>&lt;Response&gt;</code> Message Processing Rules</td>
<td>Checklist for the Web SSO profile</td>
</tr>
</tbody>
</table>
4.1.4.5 POST-Specific Processing Rules Checklist for the use of HTTP POST binding in a Web SSO context

4.1.5 Unsolicited Responses Rules for IdP initiated flows

SAML 2.0 Profile Validators

<table>
<thead>
<tr>
<th>Validator name</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assertion Issuer and Statement</td>
<td>4.1.4.2</td>
</tr>
<tr>
<td>Bearer Assertions</td>
<td>4.1.4.2</td>
</tr>
<tr>
<td>Issuer Matching IdP</td>
<td>4.1.4.2</td>
</tr>
<tr>
<td>No Assertions Within Error Response</td>
<td>4.1.4.2</td>
</tr>
<tr>
<td>POST Processing Rules</td>
<td>4.1.4.5</td>
</tr>
<tr>
<td>Response Message Processing Rules</td>
<td>4.1.4.3</td>
</tr>
<tr>
<td>Signature</td>
<td>4.1.4.3</td>
</tr>
<tr>
<td>Subject Confirmation</td>
<td>4.1.4.2</td>
</tr>
<tr>
<td>Unsolicited Response</td>
<td>4.1.5</td>
</tr>
</tbody>
</table>

SAML 2.0 Core Validators

Those SAML Core specs that are being verified are done so in one single validator.

<table>
<thead>
<tr>
<th>Validator name</th>
<th>Sections</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assertions Core</td>
<td>2.3.3 2.4.1.2 2.5.1.2 2.7.2</td>
</tr>
</tbody>
</table>

Implementation

The validation is carried out by various validators that take as input the SAML Response, HTTP Request and/or the unmarshalled XML Document - depending on what is to be validated - and outputs a set of validation errors.

These validators are then wrapped in validation steps. These are created with a set of (possibly empty) validation errors, and have a method nextStep which takes a validator and returns a new validation step containing the caller's set of errors combined with those resulting from the validator's validation process.

Authentication

If the validation process passes a Principal object with information about authentication status and expiration date will be attached to the user's session data, granting them access to the site. Otherwise the filters AuthenticationFilter and CacheFilter work together to restrict interaction with the third-party service.

User Data Processing

In addition to the data necessary for user authentication, SAML messages from FMIS carry the following user data (under Response>Assertion>AttributeStatement):
<table>
<thead>
<tr>
<th>Field name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FI_ID</td>
<td>Alphanumeric</td>
<td>This field is the DI internal ID for the subject Financial Institution.</td>
</tr>
<tr>
<td>USER_ID</td>
<td>Alphanumeric</td>
<td>The ID of the subject user on the Financial Institution’s host processing system.</td>
</tr>
<tr>
<td>USER_ACCOUNTS</td>
<td>XML Document</td>
<td>An XML document containing the list of accounts, and associated account nickname, for the user. The integration guide claims that this should be base64 encoded, but in reality appears to be a string of CDATA.</td>
</tr>
<tr>
<td>KEEP_ALIVE_URL</td>
<td>URL</td>
<td>This URL can be used by the FI or 3rd party app to keep the main online banking session alive while the end user is interacting with the 3rd party app. See below for an example of how to use the URL in an application.</td>
</tr>
<tr>
<td>XML_DATA</td>
<td>XML Document</td>
<td>This attribute is optional, and the structure of the document is determined during development and certification jointly by the third party and DI. The document can contain any additional user data needed for the functioning of the application that is not already provided by the standard attributes listed above. Note that this is base64 encoded and must be decoded by the third party.</td>
</tr>
<tr>
<td>SESSION_ID</td>
<td>Alphanumeric</td>
<td>The online banking session Id for the logged in user. This attribute is optional.</td>
</tr>
</tbody>
</table>

If the SAML validates, this data will be extracted by a SAMLProcessor and put in a DiAssertion object - a domain-specific wrapper for OpenSAML Assertions with an interface exposing the user data.

**XML Data Processing**

The USER_ACCOUNTS field is being parsed by UserAccountsProcessor#getAccountsFromXml(String xml), which returns a set of AccountEntry objects that facilitate access to various account data (account type, ID, mnemonic names &c.).

The XML_DATA field is currently not processed, but pretty-printed on the results page.

**Keep Alive**

When accessing a third party app from an internet bank the user runs the risk of having their original banking session timing out. To prevent this, the authenticating SAML message will provide a keep-alive URL for the third party site to call every so often through a short piece of javascript. This keep-alive URL is contained in an <Attribute> element, under Assertion>AttributeStatement.

**Logout**

Sending a GET request to https://servername/contextpath/logout will invalidate the user's session, restricting any further access to the site.

**Running the Sample Application**

The README.md file at the root of the sample application project provides full instructions on building, deploying, running and debugging the sample app.

**Security**

**Secure Communication**
TLS

The Tomcat plugin for maven allows for TLS encryption, although with limited configuration options. The plugin defaults to the TLS 1.0 protocol, with ECDHE-RSA key exchange and AES-128 CBC encryption. For any configuration more advanced than setting the certificate private key path a server.xml file has to be specified; the <Connector> element exposes all relevant TLS/SSL settings - protocol versions, cipher suites etc.

Certificate

The sample app comes with a self signed SHA 256 certificate (keystore located at /resources/tls/keystore.jks). Generating a new one is easily done with OpenSSL or some such tool; the only reconfiguration needed is to set the <keystoreFile> and <keystorePass> fields in the pom.xml

Encryption

PingFed gives the option to encryption assertions, essentially replacing the <Assertion> elements with <EncryptedAssertion>. com.ncrc oe.di.encryption.Cipher can then be used to decipher these.

To set up this you need to:

- Create a Java keystore somewhere in src/main/resources
- Edit encryption.properties to suit your keystore
- Export the public key certificate from the keystore and add this to your PingFed connection

As of July 2015, the encryption seems to interfere with the signing of the message.

Signature

The Web SSO specifications demand that each SAML <Assertion> is signed.

The application assumes that when the Signature is validated, the KeyInfo carried alongside the Signature will act as a public key. The sample application does not require separate access to the public key that is paired with the private key used when signing the SAML assertion.

Base64 Encoding

The sample app expects the SAML message to be base64 encoded and sent in a POST field named SAMLResponse of type application/x-www-form-urlencoded.

Filters

Two filters ensure that only authenticated users can access the contents of the site: A cache filter, that forces the browser to reload and revalidate each requested page, and an authentication filter that looks for a Principal object that has to be attached to the user's session data. If no such object can be found, or if its expiration date has passed, the request is redirected to a suitable error page.

SAML Test Driver

Overview

The SAML test driver is a lightweight java web application, the sole purpose of the application is to submit a SAML POST request to a user-specified SAML service provider. In the context of the sample application the test driver acts as the Identity Provider (IdP) within the SAML flow and the sample application acts as the Service Provider (SP).

The test driver is a simple html form that allows the following fields to be specified:

- SAML Assertion - a single SAML Assertion element. Note that whilst the sample application can handle multiple assertions, the test driver cannot.
- SAML Response - one or more SAML Response elements. Note the the Response element will be wrapped round the Assertion after the Assertion is signed and therefore should not contain any Assertions of it's own.
- Target URL - the SAML service provider to which the full content of the SAML Response should be posted.

On submission of the form the following execution flow occurs:

- The submitted Assertion is modified so that the dates contained within the XML body reflect the current date and time
- The Assertion is signed
  - The private key used during signing is stored within the test driver project (src/main/resources/com/ncr/di/genenveloped/http/privatekey_pcks8)
  - The public key carried within the KeyInfo element of the Signature is stored within the test driver project
The submitted Response is modified to contain a unique id
The Response is modified so that the dates contained within the XML body reflect the current date and time
The signed Assertion is added as a child element on the Response
The Response document is submitted to the target URL as an HTTP POST request
The body of the HTTP Response is written to the response output stream
The browser renders the contents of the response output stream

Running the Test Driver

The README.md file at the root of the test driver project provides full instructions on building, deploying, running and debugging the test driver application.

Modifying Public and Private Key

The public and private key combination provided with the test driver should not need to be modified, but if you wish to use your own public and private keys then the following files should be replaced:

- private key - src/main/resources/com/ncr/di/genenveloped/http/privatekey_pcks8
- public key - src/main/resources/com/ncr/di/genenveloped/http/publickey.cer

Once the files have been replaced the project must be rebuilt and then redeployed.

If you want to generate your own public and private keys then the following openssl commands should be issued and the generated files copied in place of the existing private and public keys

```
> openssl genrsa -out privatekey.pem 1024
> openssl req -new -x509 -key privatekey.pem -out publickey.cer -days 1825
> openssl pkcs8 -topk8 -in privatekey.pem -out privatekey_pcks8 -outform DER -nocrypt
```