SAML Web SSO

JAX-RS: SAML Web SSO

- Introduction
  - Typical Flow
- Maven dependencies
- Identity Provider
- Service Provider Security Filter
  - Redirect Binding Filter
  - POST Binding Filter
  - Signing SAML Authentication Requests
  - Filters and State Management
- Request Assertion Consumer Service
  - Dealing with signed SAML Responses
  - Signature Key Info Validation
  - Using RACS as Endpoint Filter
- SSO State Provider
- Distributed State Management
- Logout Service
- Metadata Service

Introduction

SSO is about a user having to sign in only once when interacting with a custom web application which may offer a number of individual endpoints.

CXF 2.6.1 introduces a comprehensive service provider (SP) support for the SAML Web SSO profile. This page also offers a good overview of the profile.

HTTP Redirect (via GET) and POST bindings are supported. The module has been tested against many IDP providers and is easily configurable.

The following components are required to get SSO supported:

- Identity Provider (IDP) supporting SAML SSO
- Request Assertion Consumer Service (RACS)
- Service Provider Security Filter
- SSO State Provider

The following sections will describe these components in more details

Typical Flow

Typically, the following flow represents the way SAML SSO is enforced:

1. User accesses a custom application for the first time
2. Service Provider Security Filter checks if the security context is available and redirects the user to IDP with a SAML SSO request
3. IDP challenges the user with the authentication dialog and redirects the user to Request Assertion Consumer Service (RACS) after the user has authenticated
4. RACS validates the response from IDP, establishes a security context and redirects the user to the original application endpoint
5. Service Provider Security Filter enforces that a valid security context is available and lets the user access the custom application.

Maven dependencies

```xml
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-rs-security-sso-saml</artifactId>
  <version>2.6.1</version>
</dependency>
```

Identity Provider

Identity Provider (IDP) is the service which accepts the redirect requests from application security filters, authenticates users and redirects them back to Request Assertion Security Service.
CXF does not offer its own IDP SAML Web SSO implementation but might provide it in the future as part of the Fediz project.

However, CXF has been tested against a number of popular IDP implementations which support SAML SSO and thus should be interoperable with whatever IDP is being used in the specific production environment. The interoperability tests have shown that some IDPs may process SAML request and produce SAML response data the way which may not be exactly specification-compliant and thus CXF Request Assertion Consumer Service (RACS) and Service Provider Security Filter implementations have a number of configuration properties for adjusting the way SAML requests to IDP are prepared and SAML responses from IDP are processed.

Service Provider Security Filter

SP Security Filter protects the application endpoints by checking that a valid SSO security context is available. If it is then the filter lets the request to continue, if not then it redirects the current user to IDP.

When a filter redirects a user to IDP, it creates a SAML Authentication Request, see this page for the example and appends it to the IDP Service URI or POSTed to IDP.

Additionally, a RelayState token pointing to the state of the current user request is also included which IDP will return to Request Assertion Consumer Service (RACS) after the user has authenticated.

CXF offers two SP Security filters, one for redirecting the user back to IDP via GET and another one - via POST.

Redirect Binding Filter

Redirect Binding Filter is implemented by org.apache.cxf.rs.security.saml.sso.SamlRedirectBindingFilter.

Here is an example of a typical filter protecting a custom JAX-RS endpoint:

```xml
<bean id="redirectGetFilter" class="org.apache.cxf.rs.security.saml.sso.SamlRedirectBindingFilter">
    <property name="idpServiceAddress" value="https://localhost:9443/idp"/>
    <!-- both relative and absolute URIs are supported -->
    <property name="assertionConsumerServiceAddress" value="/racs/sso"/>
    <property name="stateProvider" ref="stateManager"/>
</bean>

<bean id="stateManager" class="org.apache.cxf.rs.security.saml.sso.state.EHCacheSPStateManager">
    <constructor-arg ref="cxf"/>
</bean>
```

Note that at the very minimum the filter needs to have 3 properties set-up:
1. IDP service address
2. RACS address - it can be absolute or relative if RACS is collocated (shares the same web application context) with the application endpoint.
3. Reference to SSO State Provider.

The following optional properties affecting the created SAML request may also be set:

- String issuerId - it defaults to the base URI of the application endpoint protected by this filter, for example, "http://localhost:8080/services/app1".
- AuthnRequestBuilder authnRequestBuilder - A builder that constructs the SAML Request. It defaults to DefaultAuthnRequestBuilder.

The IDP address is where filters will redirect users to and the RACS address is where users will be redirected by IDP to. RACS will set up a security context and redirect the user back to the original application address by using the RelayState token which is included by the filters when users are initially redirected to IDP.

POST Binding Filter

POST Binding Filter is implemented by org.apache.cxf.rs.security.saml.sso.SamlPostBindingFilter.

Here is an example of a typical filter protecting a custom JAX-RS endpoint:
Note that the POST binding filter has the same 3 required properties as org.apache.cxf.rs.security.saml.sso.SamlRedirectBindingFilter has but also sets a "useDeflateEncoding" property for getting a SAML request deflated. Some IDPs might not be able to process deflated SAML requests with POST binding redirects thus the compression may be optionally disabled.

What is actually different in this case from the GET-based redirect is that the filter prepares an instance of SAMLRequestInfo which is subsequently bound to an XHTML view via a JSP filter. The view will typically have a Java Script handler which will actually redirect the user to IDP when it is loaded into the browser. The data to view binding is facilitated by org.apache.cxf.jaxrs.provider.RequestDispatcherProvider, please see this page for more information.

One may prefer using the POST binding filter in cases where having SAML request to IDP encoded as a URI parameter prohibited.

Here is a typical JSP handler for binding org.apache.cxf.rs.security.saml.sso.SAMLRequestInfo to the view:

```jsp
<%@ page import="javax.servlet.http.HttpServletRequest,org.apache.cxf.rs.security.saml.sso.SamlRequestInfo" %>
<% 
    SamlRequestInfo data = (SamlRequestInfo)request.getAttribute("samlrequestinfo"); 
%>
<html xmlns="http://www.w3.org/1999/xhtml">
<body onload="document.forms[0].submit()" method="POST">
    <form action="<%= data.getIdpServiceAddress() %>">
        <input type="hidden" name="SAMLRequest" value="<%= data.getSamlRequest() %>">
        <input type="hidden" name="RelayState" value="<%= data.getRelayState() %>">
        <div>
            <input type="submit" value="Continue"/>
        </div>
    </form>
</body>
</html>

Signing SAML Authentication Requests
The filters may optionally sign SAML requests, the following configuration properties can be set-up:

- boolean signRequest - Whether to sign the AuthnRequest or not. The default is false.
- String signatureUsername - The keystore alias to use to sign the AuthnRequest.
- Crypto signatureCrypto - A WSS4J Crypto object if the SAML AuthnRequest is to be signed.
- String signaturePropertiesFile - This points to a properties file that can be used to load a Crypto instance if the SAML AuthnRequest is to be signed.
- CallbackHandler callbackHandler - A CallbackHandler object to retrieve the private key password used to sign the request.
- String callbackHandlerClass - A class name that is loaded for use as the CallbackHandler object.

Either the "signatureCrypto" or "signaturePropertiesFile" properties must be set if "signRequest" is set to true. Similarly, either "callbackHandler" or "callbackHandlerClass" must be configured.

Example:

```xml
<bean id="ssoSignedRedirectPOST" class="org.apache.cxf.rs.security.saml.sso.SamlPostBindingFilter">
    <property name="idpServiceAddress" value="https://localhost:9443/idp"/>
    <property name="assertionConsumerServiceAddress" value="/racs/sso"/>
    <property name="stateProvider" ref="stateManager"/>

    <property name="signRequest" value="true"/>
    <property name="callbackHandlerClass" value="org.apache.cxf.samlp.sso.SSOCallbackHandler"/>
    <property name="signatureUsername" value="myservicekey"/>
    <property name="signaturePropertiesFile" value="serviceKeystore.properties"/>
</bean>

<bean id="stateManager" class="org.apache.cxf.rs.security.saml.sso.state.EHCacheSPStateManager">
    <constructor-arg ref="cxf"/>
</bean>
```

Filters and State Management

The following properties affect the way filters manage the SSO state:

- SPStateManager stateProvider
- long stateTimeToLive - default is 2 minutes (in milliseconds).
- String webAppDomain.
- boolean addWebAppContext - default is true.
- boolean boolean addEndpointAddressToContext - default is false.

The 'stateProvider' refers to a custom SPStateManager implementation and is used for filters and RACS coordinating with the filters persisting the current user request state. RACS validating it and persisting the current security context state and filters getting the information about the context. Filters and RACS use a 'RelayState' token to work with the current request state. RACS persists the security context and the filters retrieve and validate it using the cookie which RACS also sets to point to this security context.

Note that a 'stateTimeToLive' property can be used to control how long the current security context can be valid for.

Both filters and RACS use opaque cookies to refer to the original request and security context state and 'webAppDomain', 'addWebAppContext' and 'addEndpointAddressToContext' affect the way these cookies can be shared between multiple SP custom applications.

For example, here is a typical Set Cookie request issued by a web application to the browser:

```
Set-Cookie: value; Domain=mydomain; Path=/accounts; Expires=Wed, 13-Jan-2021 22:23:01 GMT;
```

By default, CXF will get a Cookie 'Path' property set to something like "/services", where 'services' is the actual name of the war archive. The 'addEndpointAddressToContext' property can be further restrict this path to something like "/services/app1", "/services/app2", where "/app1" and "app2" are jaxrs:endpoint addresses, this can be handy for testing, with every jaxrs:endpoint within a single war having its own security context.

If the custom SP application is 'spread' across multiple containers with different application context names, then the 'addWebAppContext' can be set to 'false' leading to Cookie 'Path' parameters set to '?' and the 'webAppDomain' property set to some shared value.

Note that the stateTimeToLive property affects a Cookie 'Expires' property but also used by filters and RACS to enforce that the internal state has not expired.

Request Assertion Consumer Service

Request Assertion Consumer Service receives a SAML Authentication Response and RelayState token from IDP, uses the token to validate the response against the data available in the original SAML Authentication Request, creates a security context if it does not already exists for the current user, persists it and redirect the user back to the original endpoint.
The RACS processes the SAML Response, and validates it in a number of ways:

- The `SAMLProtocolResponseValidator` validates the Response against the specifications and checks the signature of the Response (if it exists), as well as doing the same for any child Assertion of the Response. It validates the status code of the Response as well.
- The `SAMLSSOResponseValidator` validates the Response according to the Web SSO profile.

Here is a typical RACS configuration:

```xml
<bean id="consumerService" class="org.apache.cxf.rs.security.saml.sso.RequestAssertionConsumerService">
    <property name="stateProvider" ref="stateManager"/>
    <!-- responses are expected to be deflated by default
    <property name="supportDeflateEncoding" value="false"/>
    -->
    <!-- responses are expected to be base64 encoded by default
    -->
    <property name="supportBase64Encoding" value="false"/>
</bean>

<bean id="stateManager" class="org.apache.cxf.rs.security.saml.sso.state.EHCacheSPStateManager">
    <constructor-arg ref="cxf"/>
</bean>

<jaxrs:server address="/racs">
    <jaxrs:serviceBeans>
        <ref bean="consumerService"/>
    </jaxrs:serviceBeans>
</jaxrs:server>
```

RACS is implemented as a JAX-RS server endpoint. It needs a reference to the SSO State Manager and by default it expects that SAML Response is deflated and Base64 encoded which can be changed. It shares the same 'stateTimeToLive' property with the filters which can be used to restrict the time the security context state is kept for.

The following properties may also be set up:

- boolean `enforceKnownIssuer` - Whether the Issuer of the Response (and child Assertions) is "known" to the RACS. This value is compared against the IDP URL configured on the filter. The default value is true.
- `TokenReplayCache replayCache` - A TokenReplayCache implementation to store Assertion IDs for the POST binding to guard against replay attacks. The default uses an implementation based on EhCache.

### Dealing with signed SAML Responses

RACS can be setup to support verifying signed Responses, or signed Assertions contained in a Response. Similarly, either "callbackHandler" or "callbackHandlerClass" must be configured if you wish to support decrypting encrypted Assertions. For example:

```xml
<bean id="consumerService" class="org.apache.cxf.rs.security.saml.sso.RequestAssertionConsumerService">
    <property name="stateProvider" ref="stateManager"/>
    <property name="supportBase64Encoding" value="false"/>
    <property name="signaturePropertiesFile" value="serviceKeystore.properties"/>
    <property name="enforceAssertionsSigned" value="false"/>
    <property name="callbackHandlerClass" value="org.apache.cxf.samlp.sso.SSOCallbackHandler"/>
</bean>
```

In this example the "enforceAssertionsSigned" enforcing that signed Assertions are contained in a Response is disabled by default and RACS will only verify that the actual Responses are signed.

### Signature Key Info Validation

By default `ds:Signature` is expected to contain `ds:KeyInfo` element.

Setting a "keyInfoMustBeAvailable" property to false will lead to a default store alias being used to load the certificate for validating the signature.

### Using RACS as Endpoint Filter

As you can see from the documentation above, RACS is typically represented as an independent service endpoint or service bean: in such cases RACS redirects the requestor back to the the actual endpoint.
Starting from CXF 3.0.0 it is possible to set it up as the target endpoint filter, simply add org.apache.cxf.rs.security.saml.sso.
RequestAssertionConsumerFilter to the list of other endpoint providers.

In this case the authentication filters do not have to set their "assertionConsumerServiceAddress" property

SSO State Provider

SP Security Filters and RACS depend on the custom SPStateManager implementation for persisting the current request and security context state.

CXF ships a basic MemorySPStateProvider and an EhCache-based implementation which is memory based with an option to overflow to the disk. Users can customize the EhCache provider or register their own custom SPStateProvider implementations if required.

For example, by default, the EhCache provider will overflow the data to the system temp directory and will not persist the data across restarts. The default configuration can be seen here. To change the default configuration, simply copy this file to a custom location and make whatever changes are required.
Assuming this configuration is saved in WEB-INF/ehcache.xml, the EhCache provider can be configured as follows:

<bean id="stateManager" class="org.apache.cxf.rs.security.saml.sso.state.EHCacheSPStateManager">
    <constructor-arg value="/WEB-INF/ehcache.xml"/>
</bean>

Distributed State Management

If you have a complex application supported by a number of wars deployed into different containers, one has to decide whether to have a single RequestAssertionConsumerService (RACS) endpoint which IDP will redirect to when processing the user authentication requests or have a separate RACS endpoint per every web application which all form a bigger application.

For example, assume you have server1, server2 and server3 which all support a bigger application. One can have a serverRacs web application which will host a RACS endpoint. Next, server1, server2 and server3 SSO filters will all point to this standalone RACS endpoint when redirecting the user to IDP and IDP will eventually redirect the user to RACS which in turn will redirect the user to the original target URI supported by server or server2 or server3.

In this case, one has to decide how the state between SSO security filters protecting the individual servers and RACS will be shared.
One approach is to setup the Ehcache provider to use Terracotta or RMI with the multicast all.

CXF offers a simple HTTPSPStateManager provider which can be used to simplify the task of setting up the distributed state cache, which can be used for simple distributed web applications or to support the more advanced applications at the proof-of-concept stage.

For example, the following jaxrs:endpoint can be deployed alongside the RACS endpoint running in its own web application:

Note that the RACS bean itself directly uses HTTPSPStateManager which is also available as an HTTP endpoint for all the SSO security filters to work with.
Here is an example of how the SPStateManagers at the individual SSO filter end can use this HTTP endpoint:
<jaxrs:client id="stateManager"
    address="https://localhost:${racs.port}/racs"
    serviceClass="org.apache.cxf.rs.security.saml.sso.state.HTTPSPStateManager"/>

<bean id="ssoRedirectURI" class="org.apache.cxf.rs.security.saml.sso.SamlRedirectBindingFilter">
    <property name="idpServiceAddress" value="${idp.address}"/>
    <property name="assertionConsumerServiceAddress" value="https://localhost:${racs.port}/racs/sso"/>
    <property name="stateProvider" ref="stateManager"/>
    <property name="addWebAppContext" value="false"/>
</bean>

Note that a JAX-RS Client proxy to the HTTPSPStateManager endpoint is used as SPStateManager reference.

The alternative to having a distributed state cache be set up is to simply have a RACS endpoint collocated with every individual web application constituting the bigger application, see the earlier section describing SSO filters on how this can be easily set up. One possible downside of it is that there will be no centralized store managing the state required by different filters and RACS which in turn can make it more difficult to audit and log all the SSO-related activities spanning across all the bigger application.

Logout Service

CXF 3.0.0 introduces LogoutService. It will remove the SSO state for the logged-in user, and can be registered as an independent endpoint or service bean.

It returns LogoutResponse bean which is expected to be processed by the View handler.

For example, one can imagine a user getting HTML page confirming the logout has been successful and linking to the application front page.

Metadata Service

A new MetadataService is available in CXF 3.1.0 and 3.0.5, which can publish SAML SSO Metadata for a given service. Similar to the Logout Service, it is registered as an independent endpoint or service bean. A sample spring configuration is available here in the CXF system tests.