IdPAddAttribute

Define and Release a New Attribute in an IdP

An attribute passes through four main steps from source system to release to an SP: it’s pulled from the system of record, massaged within the provider, given a set of protocol-specific encoders, and then filtered for release. Every attribute has a unique attribute ID which is used to refer to it consistently through this process. We’ll track an example through the process.

- **Before You Begin**
  - 1. Pull in Raw Attributes
  - 2. Prepare the Attributes
    - Attribute Definition
    - Attribute Encoding
  - 3. Release the Attribute
  - 4. Test the Attribute
  - Examples

**Before You Begin**

The configuration files dealing with attribute resolution and filtering have a few basic patterns in common and understanding them will help you in writing proper configuration files.

First, within each configuration file similar configurable components all use the same XML element tag name. For example, within the attribute resolver configuration all attribute definitions start with the XML element tag name `AttributeDefinition` and all data connectors with the tag name `DataConnector`. A type attribute, `xsi:type`, is then used to determine the type of the component (e.g. a database, an LDAP data connector). The type also determines the content, or configuration options, of that component.

Second, all data within an attribute object (i.e. it’s ID, values, etc.) is case sensitive regardless of what created the object. This may be confusing at first because some data sources treat some, or all, of their information as case insensitive. However, in order to avoid general confusion about when a particular piece of data is case sensitive or not, the simple rule of “all information is case sensitive” is followed.

**1. Pull in Raw Attributes**

Shibboleth doesn’t store any attributes about users itself; it relies on external data stores to supply user information to be released. These attributes are pulled from data sources using data connectors. You may need to modify your existing data connectors or add a new one if this attribute isn’t available through any existing connector. Data connectors may produce more than one attribute and each attribute may have many values. Attributes produced by data connectors are never released outside of the attribute resolver.

- The static data connector is used to add statically attributes and values to every person served by the identity provider. An example usage of this connector would be to add an entitlement attribute that everyone in your organization receives.
- The computed ID data connector is used to construct unique identifiers by hashing together some information.
- The stored ID data connector is used to construct and persist identifiers by means of a database.
- The relational database connector is used to pull attributes from a relational database by executing some configured SQL.
- The LDAP data connector is used to pull attributes from an LDAP directory by executing an LDAP filter on a specific branch.

### Example Data Connector

```xml
<resolver:DataConnector xsi:type="dc:RelationalDatabase"
 xmlns="urn:mace:shibboleth:2.0:resolver:dc"
 id="MyDatabase">
  <ApplicationManagedConnection
    jdbcDriver="org.hsqldb.jdbcDriver"
    jdbcURL="jdbc:hsqldb:res:/data/database/shibdb"
    jdbcUserName="sa"/>
  <QueryTemplate>
    <![CDATA[
      SELECT * FROM PEOPLE WHERE netid='${principal}'
    ]]>  
  </QueryTemplate>
</resolver:DataConnector>
```

**2. Prepare the Attributes**
There are two rounds of transformation each attribute will undergo as it passes through the IdP. The first set of changes allow the IdP to transform attributes (merge, split, reformat, etc.) using other attributes to get a complete data definition. The second set of changes transform the attribute from the internal representation to one appropriate for the protocol the IdP will be communicating with, a process known as attribute encoding.

Attribute Definition

First, you'll need an attribute definition that gives an ID and value to the attribute. That step alone will be enough most of the time, but if additional transformation is needed, you can use additional attribute definitions dependent on the primary attribute definition. Click below for some short examples of a few ways to do that.

- The simple attribute definition is used to handle flat attributes with a simple name and values.
- The scoped attribute definition is used to handle flat attributes with a simple name and scoped values.
- The principal name attribute definition creates an attribute whose value is the user's principal name.
- The principal authentication method attribute definition creates an attribute whose value is the user's authentication method.
- The regex based split attribute definition creates an attribute whose values are the split of a set of input values.
- The transient ID creates an attribute whose value is a short-lived identifier.
- The SAML 1 NameIdentifier attribute definition creates an attribute whose values are SAML 1 NameIdentifiers.
- The SAML 2 NameID attribute definition creates an attribute whose values are SAML 2 NameIDs.
- The script attribute definition builds an attribute's values based on an ECMAScript script.
- The mapped attribute definition is used to map an attribute's values to different values using regular expressions.
- The template attribute definition uses the VelocityTemplateLanguage to construct values by combining other attribute values.

Example Attribute Definition using defined Data Connector

```xml
<resolver:AttributeDefinition xsi:type="ad:Simple" id="uid" sourceAttributeID="NETID">
  <Dependency ref="MyDatabase" />
</resolver:AttributeDefinition>

<resolver:DataConnector xsi:type="dc:RelationalDatabase" xmlns="urn:mace:shibboleth:2.0:resolver:dc" id="MyDatabase">
  <ApplicationManagedConnection jdbcDriver="org.hsqldb.jdbcDriver" 
                                 jdbcURL="jdbc:hsqldb:res:/data/database/shibdb" 
                                 jdbcUserName="sa" />
  <QueryTemplate>
    <![CDATA[
      SELECT * FROM PEOPLE WHERE netid='${principal}'
    ]]> 
</QueryTemplate>
</resolver:DataConnector>
```

Attribute Encoding

Once the attribute has the internal name and value you want, one more transformation has to take place to give it the right format on the wire. While this process occurs after attribute filtering, described next, you need to provide the configuration here. This is done by attaching attribute encoders to the attribute definitions. Each attribute needs one encoder per protocol it will be sent on. You should always communicate with your partners to make sure they're expecting the names you're sending.

- SAML 1 String Attribute Encoder is used to encode an attribute as a SAML 1 <Attribute> with simple strings for values.
- SAML 1 Scoped String Attribute Encoder is used to encode an attribute as a SAML 1 <Attribute> with scoped strings for values.
- SAML 1 Base64 Attribute Encoder is used to encode an attribute as a SAML 1 <Attribute> with a byte array for a value.
- SAML 1 XMLObject Attribute Encoder is used to encode an attribute, with XMLObjects as values, into a SAML 1 <Attribute>.
- SAML 1 String NameIdentifier Encoder is used to encode an attribute, having a simple string value, into a SAML 1 Subject <NameIdentifier>.
- SAML 2 String Attribute Encoder is used to encode an attribute as a SAML 2 <Attribute> with simple strings for values.
- SAML 2 Scoped String Attribute Encoder is used to encode an attribute as a SAML 2 <Attribute> with scoped strings for values.
- SAML 2 Base64 Attribute Encoder is used to encode an attribute as a SAML 2 <Attribute> with a byte array for a value.
- SAML 2 XMLObject Attribute Encoder is used to encode an attribute, with XMLObjects as values, into a SAML 2 <Attribute>.
- SAML 2 String NameID Encoder is used to encode an attribute, having a simple string value, into a SAML 2 Subject <NameID>.
Example attribute encoder for a defined attribute definition

```xml
<resolver:AttributeDefinition xsi:type="ad:Simple" id="uid" sourceAttributeID="NETID">
  <resolver:AttributeEncoder xsi:type="enc:SAML2String"
    name="urn:oid:1.3.6.1.4.1.5923.1.1.1.6"
    friendlyName="eduPersonPrincipalName" />

  <resolver:Dependency ref="MyDatabase" />
</resolver:AttributeDefinition>

<resolver:DataConnector xsi:type="RelationalDatabase" xmlns="urn:mace:shibboleth:2.0:resolver:dc"
  id="MyDatabase"
  validationQuery="SELECT 1;">
  <ApplicationManagedConnection jdbcDriver="org.hsqldb.jdbcDriver"
    jdbcURL="jdbc:hsqldb:res:/data/database/shibdb"
    jdbcUserName="sa" />

  <QueryTemplate><![CDATA[
  SELECT * FROM PEOPLE WHERE netid='${principal}'
  ]]]></QueryTemplate>
</resolver:DataConnector>
```

3. Release the Attribute

Newly defined attributes are not released to service providers until you define an attribute filter policy for that attribute. Such policies describe which service providers, under which conditions, receive which attributes.

Example filter that always releases the UID attribute we defined

```xml
(AttributeRule attributeID="uid">
  <PermitValue xsi:type="basic:ANY" />
</AttributeRule>
```

4. Test the Attribute

If you want, you can test your attribute using the attribute authority command line interface (AACLI). This will check the resolver, the filters, and also the metadata, so you can know exactly what will happen in any given situation.

Examples

Additional examples are also available. These provide more complete examples and are contributed by users of the software.

Also see IdPNameIdentifier for additional examples of adding name identifiers.

| Example 1 | Pulls a users email from an LDAP directory, encodes it as a name identifier, and releases it to Google for use with Google Apps. |