SICS AssertionServer

Ludwig Seitz
SICS

September 30, 2011

Abstract

The SICS AssertionServer is a lightweight SAML-based identity management tool. It's main purpose is to store and manage attribute information and issue SAML attribute assertions for the stored attribute information.

AssertionServer provides pluggable modules for authenticating and authorizing management of the attribute data it maintains. Example implementations of these modules are provided.

The SICS Assertion Server is not a stand-alone product, rather is should be exposed as a service through a user-friendly interface and tied into a security architecture, that consumer the assertions it produces.

1 Introduction

There a numerous commercial (and probably several open-source) solutions for identity management. However most of these (in fact all known to the author) are very heavy-weight with lots of overhead, making them difficult to use without extensive training.

Therefore we have decided to provide a lightweight identity management service targeted at managing attributes and issuing SAML [1] assertions for these attributes.

The architecture of the AssertionServer is shown in figure 1

The elements of the architecture are as follows:

- An API providing the main functions of the AssertionServer. This API is supposed to be exposed as a service (e.g. as a webservice).
- An attribute database. This database is managed via a pluggable database connector.

  AssertionServer currently provides an example MySQL connector.
- An authentication module. This is used to authenticate users that want to access the API. The module plugs into an extension point, so that new types of authentication can easily be plugged in. We provide an example module with password based authentication.

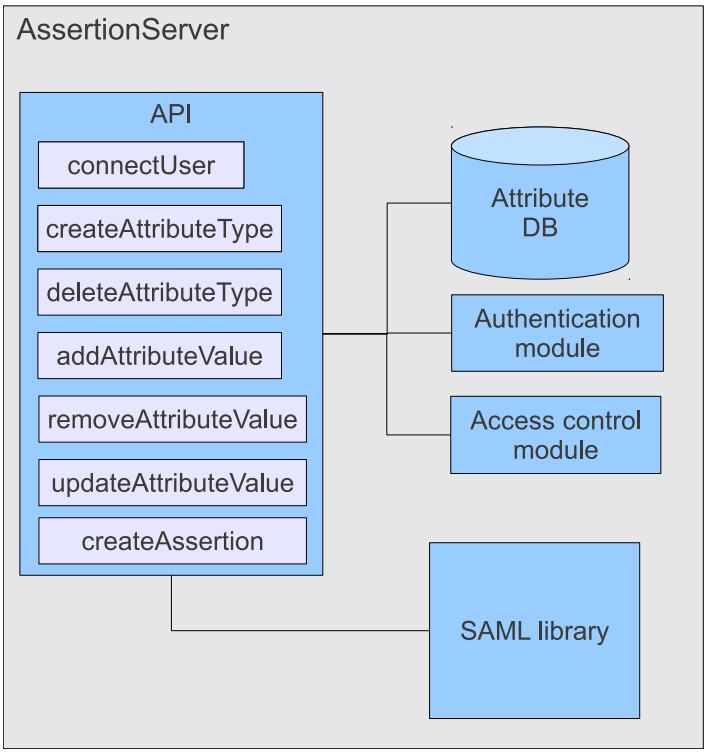

Figure 1: AssertionServer architecture

- An access control module. This is used to check the authorization of user that want to access the API. The module plugs into an extension point, so that new types of access control can easily be plugged in. We provide a simplistic example module where the owner of an attribute can perform any action, and other users can perfrom the *createAssertion* function on arbitrary attributes.
- The *SICS lighweight SAML library* (SlSl) is used to support the creation of attribute assertions.

2 Concepts

The following concepts are important in order to understand the purpose of AssertionServer. As an identity management tool, assertion server manages *Attributes* assigned to *Subjects*. Examples of subject attributes could be their age, security clearance, or role.

An *Attribute* is defined as a tuple of *Source of authority*, *attribute identifier*, *datatype* and *attribute value*.

The *Source of authority* (SOA) of an attribute defines the subject that is in control of this attribute (i.e. that can decide over the assignment of attribute values to other subjects).

The *attribute identifier* is the name of the attribute e.g. *Age*, or *SecurityClearance* or *Role*.

Each attribute has a specific *Datatype* that constraints the set of possible values for this attribute. For example age has the datatype integer (while it could even be further constrained to positive integers).

An *Attribute Value* is the value of an attribute assigned to a subject. E.g. For *subject = Ludwig* and the *attribute identifier = role* an attribute value could be *senior researcher*.

An *Attribute Definition* defines the SOA, attribute identifier, datatype, and optionally a set of allowed values for the attribute. Note that the attribute definition does not contain any assignments of attribute values to subjects.

The assertion server requires you to create an attribute definition, before you start to assign attribute values for this attribute.

SAML assertions are signed XML documents that bind an attribute to a subject. The issuer of the SAML assertion, asserts that the subject has the named attribute, e.g. *SICS (issuer) says that Ludwig (subject) has the role (attribute-identifier) "senior researcher" (attribute value).*

The SAML standard is defined by the OASIS standardization organization, and is described in numerous documents available on the web.

3 Configuration

In order to operate the SICS AssertionServer you need to perform a small number of configuration steps:

1. Install the MySQL database server and create a user for the AssertionServer. By default this user is called *saml-aa* in the AssertionServer code, but this value is configurable.
2. Get an instance of the `DBConnector` class and run the `init(String)` method in order to create the necessary tables. See 3.1 for details.
3. Select an authentication module and configure it. See 3.2 for details.
4. Select an access control module and configure it. See 3.2 for details.
5. Create a pair of assymetric keys for the AssertionServer. These will be used for signing SAML assertions.

After this you can create an instance of the `AssertionServer` class and use the functions provided by the API.

3.1 Database Connectors

The AssertionServer needs a database connector in order to store information on a database. The connector needs to implement the `DBConnector` interface.

We provide the class `SQLConnector` that uses the Interfaces from the `java.sql` package.

In order to make this work with a concrete database you need to put a JDBC-jar in the build path. We have tested SQLConnector with the *MySQL* database and the `mysql-connector-java.jar` JDBC library.

3.2 AAA-modules

AssertionServer supports pluggable modules for Authentication and Access Control. Furthermore Accounting is supported by configurable logging with the Log4j library.

The Authentication module needs to implement the `AuthenticationModule` interface, while the Access Control module needs to implement `AccessControlModule`. Both modules require that you run the `init(DBConnector)` method first, in order to create necessary database tables for the authentication or access control data.

The Access Control module trusts the previous authentication of a user by the Authentication module, and obtains the user-id of the currently connected user from the `AssertionServer` class.

We provide simplistic example modules and encourage the development of more sophisticated ones.

The `PasswordAuth` class implements password-based authentication. The authentication token passed to the `newUser()` and `doAuth()` methods of this class must be the raw password as a `String`.

The `BasicAccessControl` class implements a policy, where the source of authority (SOA) of an attribute can do any operation on this attribute, while other users can only retrieve SAML assertions.

3.3 Certificates

The AssertionServer requires a X.509 certificate and the associated secret key in order to sign SAML assertions it issues. The certificate and the corresponding private key is expected to be encoded in a file in *PKCS#12* format (i.e. a *.p12* file).

Furthermore you need to provide the root certificate(s) that was used to sign the CA certificate as a Base64 encoded DER certificate (i.e. a *.pem* file).

This information is used as parameters in the constructor of the `AssertionServer` class.

4 Functions

The AssertionServer API (from the `AssertionServer` class) supports the following functions:

- Connecting a User;
- Disconnecting a User;
- Creating an Attribute Definition;
- Deleting an Attribute Definition;
- Adding an Attribute Value;
- Removing an Attribute Value;
- Updating an Attribute Value;
- Getting a SAML Assertion.

They are described briefly in the following subsections.

4.1 Connecting a User

In order to perform any of the other functions of the AssertionServer a user needs to establish a connection (and in the process of doing so to authenticate). The method `connectUser(SAMLID, Object)` allows a user to connect to the AssertionServer. The SAMLID is expected to be the user identifier of the connecting user, while the Object can either contain an authentication token, or information necessary to perform an authentication protocol.

The exact contents of the Object parameter are up to the Authentication module (see 3.2).

After a user is connected, his identity is used to authorize any requests until the user is disonnected.

Note that the AssertionServer does not support multiple administrative users connected at the same time.

4.2 Disconnecting a User

The `disconnectUser()` method disconnects the current user.

4.3 Creating an Attribute Definition

The `createAttribute(AttributeDefinition)` method creates a new attribute definition. The `AttributeDefinition` parameter contains all relevant information for this.

4.4 Deleting an Attribute Definition

The `deleteAttribute(AttributeDefinition)` method deletes an existing attribute definition and all associated attributes. If the attribute definition given as parameter does not exist, the AssertionServer returns an error message.

4.5 Adding an Attribute Value

The `addAttributeValue(SAMLID, SAMLID, SAMLAttribute)` method adds a new attribute to a subject. The first SAMLID represents the subject, the second the SOA of the attribute and finally the attribute is provided in the SAMLAttribute parameter.

4.6 Removing an Attribute Value

The `removeAttributeValue(SAMLID, SAMLID, SAMLAttribute)` method removes an existing attribute from a subject. The first SAMLID represents the subject, the second the SOA fo the attribute. Note that the SAMLAttribute must specify the whole attribute (including the attribute value) in order for this method to succeed.

4.7 Updating an Attribute Value

The `updateAttributeValue(SAMLID, SAMLID, SAMLAttribute, String)` method updates an attribute. The first SAMLID represents the subject, the second the SOA fo the attribute. The SAMLAttribute parameter must contain the old attribute, while the String parameter must contain the String representation of the new attribute value.

4.8 Getting a SAML Assertion

The `getAssertions(AssertionRequest)` method generates SAML assertions for existing attributes. Note that the `AssertionRequest` parameter allows to set the attribute value to `null` in order to retrieve all attribute values for a specific attribute.

5 Code example

```
//Use default values for connecting to the database
SQLConnector db =  new SQLConnector(null, null, null);

AccessControlModule pdp = new BasicAccessControl();

//Use the default hash algorithm on the passwords
PasswordAuth auth = new PasswordAuth(null);

List<X509Certificate> certs = new ArrayList<X509Certificate>();
File rootCertFile = new File("resources/testData/Certificates/cacert.pem");

CertificateFactory certFact = CertificateFactory.getInstance("X.509");

BufferedInputStream bis = new BufferedInputStream(
   new FileInputStream(rootCertFile));

X509Certificate rootCert
   = (X509Certificate) certFact.generateCertificate(bis);
certs.add(rootCert);

AssertionServer sas = new AssertionServer(db, pdp, auth,
   "resources/testData/Certificates/aserver.p12",
   "password", certs, 1);

HashSet<String> allowedValues = new HashSet<String>();
allowedValues.add("member");
allowedValues.add("administrator");

SAMLNameID subject = new SAMLNameID("ludwig@sics.se");
String soaStr = "testSOA";
SAMLNameID soa = new SAMLNameID(soaStr);
String id = "testID";
String dataType = SAMLAttribute.xmlStringId;
```

```
//Add required XML-attributes for the SAML-XACML profile
HashMap<String, String> otherXMLAttrs = new HashMap<String, String>();
otherXMLAttrs.put("xmlns:" + SAMLAttribute.xacmlprofNSPrefix,
   SAMLAttribute.xacmlprofNS);
otherXMLAttrs.put(SAMLAttribute.xacmlprofNSPrefix + ":" +
   "DataType", dataType);

ArrayList<SAMLAttributeValue> values
   = new ArrayList<SAMLAttributeValue>();
values.add(new SAMLAttributeValue(null, "member"));
SAMLAttribute valueMember = new SAMLAttribute(
   id, SAMLAttribute.xacmlNameFormat, null,
   otherXMLAttrs, values);

values.clear();
values.add(new SAMLAttributeValue(null, "administrator"));
SAMLAttribute valueAdmin = new SAMLAttribute(
   id, SAMLAttribute.xacmlNameFormat, null,
   otherXMLAttrs, values);

AttributeDefinition ad = new AttributeDefinition(soa, id,
   dataType, allowedValues);

//Create the database
db.init("password");
auth.init(db);

//Create a new user
auth.newUser(ad.getSOA().getName(), "password", db);

//Authenticate the user
RequestResult result = sas.connectUser(ad.getSOA(), "password");
if (!result.success()) {
   System.out.println(result.getFailureReasons());
}

//Create an attribute
result = sas.createAttribute(ad);

if (!result.success()) {
   System.out.println("..." + result.getFailureReasons());
}

//Adding an attribute value
result = sas.addAttributeValue(subject, soa, valueMember);
if (!result.success()) {
   System.out.println("..." + result.getFailureReasons());
}
```

```
//Update an attribute value
result = sas.updateAttributeValue(subject, soa, valueMember,
   "administrator");
if (!result.success()) {
   System.out.println("..." + result.getFailureReasons());
}

//Get assertion
AssertionRequest request = new AssertionRequest(subject, soa, valueAdmin);
result = sas.getAssertions(request);
SignedSAMLAssertion assertion = result.getResults().get(0);
String assertionXML = assertion.toString(new Indenter());
System.out.println(assertionXML);

//Parse an assertion
XMLInputParser parser = new XMLInputParser(null, null);
Document doc = parser.parseDocument(assertionXML);
SignedSAMLAssertion parsedAssertion
   = SignedSAMLAssertion.getInstance(doc.getDocumentElement());

//Test validity of conditions
if (!parsedAssertion.getConditions().checkValidityIntervall()) {
   // Failed
} else {
  // Success
}

//Delete an attribute value
result = sas.removeAttributeValue(subject, soa, valueAdmin);
if (!result.success()) {
   System.out.println(result.getFailureReasons());
}

//Delete an attribute
result = sas.deleteAttribute(ad);

if (!result.success()) {
   System.out.println("..." + result.getFailureReasons());
}

//Delete a user
auth.deleteUser(ad.getSOA().getName(), db);
```

References

[1] S. Cantor., J. Kemp., R. Philpott., E. Maler., eds.: Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) v2.0. Standard, Organization for the Advancement of Structured Information Standards (OASIS) (2005) http://www.oasis-open.org.