SAM J2EE Implementation Guide - JAAS Login Modules and SAML Web Services

39 A2 02LT Rev01

Secure Access Manager J2EE Implementation Guide

JAAS Login Modules and SAML Web Services

Subject	This guide describes how to use the JAAS login modules with J2EE EJB platforms or with Java applications, and how to use SAML Web Services.
Intended Readers	• System integrators. • Programmers.
Software/Hardware Required	SAM J2EE 7.0 and later versions. For more information about the versions of the required operating systems and other software solutions mentioned in this guide, please refer to *SAM Web and SAM J2EE Release Notes*.
Supported Operating Systems	SAM J2EE runs on the following systems: • Solaris. • Linux. Note: Refer to *SAM Web and SAM J2EE Release Notes* for more informat on the supported operating systems versions.
Directory Paths	Not applicable
Date	June 2006.
Typographical Conventions	**Bold**: Identifies the following: • Interface objects such as menu names, labels, buttons and icons. • File, directory and path names. • Keywords to which particular attention must be paid. *Italics*: Identifies references to other guides. `Monospace`: Identifies portions of program codes, command lines, or messages displayed in command windows. Capitalization: Identifies application specific objects (in addition to standard capitalization rules).

If you have any comments or questions related to this documentation, please mail us at **Institute@Evidian.com**

39 A2 02LT Rev01

39 A2 02LT Rev01

Contents

1. Introduction

2. Installing SAML Login Module

3. Implementing SAML Login Module With a Standalone Java Application

4. Developing your Own Principal and Credential Classes

5. SAML Web Services - Resources Location

6. WebLogic Platform

7. JBoss Platform

8. WebSphere Platform

9. Oracle Application Server Platform

10. Login Modules on SDM (Security Data Manager)

1. Introduction

This section is an overview of AccessMaster SAM J2EE functionalities.

1.1 Presentation

1.1.1 AccessMaster SAM J2EE

AccessMaster SAM J2EE is a solution based on SAML technology. It supports authentication of users, and provides SAML assertions for them. SAML assertions are documents that contain information about a user and the authentication phase.

SAM J2EE is designed to be seamlessly integrated in J2EE platforms. It provides for user authentication, and the contents of the SAML assertion is used to determine the user rights.

But SAM J2EE is a misleading name as the product can be used in Java applications that do not run in a J2EE environment. In addition it can also be used in non-Java environments, even with scripting languages such as JavaScript in a Web browser, through the Web Service Interface.

1.1.2 SAML Provider

A trusted authority, the SAML provider, delivers SAML assertions. In the SAM J2EE architecture, the provider is the authentication server. An assertion is delivered as a result of a positive authentication.

The SAML provider signs assertions. The assertion consumers must verify the signature and trust the provider. This verification is automatically done when the SAM J2EE JAAS LoginModule is used on the assertion consumer side (see 1.3.5 “SAM J2EE Login methods”)

1.1.3 Interfaces

Applications can authenticate users and request SAML assertions through two interfaces:

Java API: JAAS;

Web Service: SOAP protocol.

1.2 SAML

1.2.1 What is SAML?

Security Assertion Markup Language (SAML) is an OASIS standard. A SAML assertion is an XML document that contains security information relating to a user.

An assertion is made of statements. Assertions delivered by SAM J2EE contain two statements:

An authentication statements;

An attributes statement.

The attributes statement contains user attributes. These attributes may be:

User identities (name, email address, phone number etc.); or

Properties that can be shared by several users (job, organization, roles etc.).

The attributes come from the user registry. The attributes to be included are defined in the administration console. These attributes can then be extracted and mapped to application object classes on the assertion consumer side.

1.2.2 Authentication

A trusted third party delivers SAML assertions. They are a proof that the user has been authenticated. But they are not an authentication mechanism by themselves.

1.2.3 Assertion Usage

An assertion can be used to get information about an authenticated user. For example on J2EE platform the user rights are determined from the assertion received from the authentication server.

An assertion can also be forwarded to a server. For example a trusted client can authenticate a user, get the corresponding assertion, and send it to a server. The server could be a Web Service provider; the client (the service consumer) could be a Web server. The Web server authenticates to the service provider; it authenticates the users and sends the assertions to the provider in the SOAP envelope.

As SAML is an OASIS standard many vendors support it. It is a good choice for interworking in a heterogeneous environment. As such it is a base for identity federation.

1.3 JAAS

The Java Authentication and Authorization Service (JAAS) is the Java standard for authentication and authorization. It is designed to provide a framework and standard programming interface for authenticating users and for assigning privileges. Introduced as an optional package to the JDK 1.3, JAAS has been integrated into the JDK 1.4 as a standard function.

It is composed of two interdependent parts: an authentication part and an authorization part. The authentication part is used to determine the identity of the user, while the authorization part checks the permissions of the authenticated user and controls resource access based on the user's privileges.

1.3.1 Pluggable and Stackable Authentication

The JAAS authentication framework is based on the Pluggable Authentication Module (PAM) model, and therefore allows system administrators to plug in the appropriate authentication services to meet their security requirements. The architecture also enables applications to remain independent from the underlying authentication services.

The authentication services are actually implemented in LoginModules. JAAS, like PAM, supports the notion of stacked LoginModules. The combination of LoginModules to be used by an application is defined by configuration, independently of the application. Below is a JAAS configuration file sample:

```
kerberos {
        com.sun.security.auth.module.Krb5LoginModule required
                    useTicketCache="true"
                    ticketCache="${user.home}${/}tickets";
} ;
loginChain {
        com.evidian.security.auth.login.SamlLoginModule requisite
                request=id-password;
        sample.SampleLoginModule optional
                debug=true;
} ;
```

In this sample, the “loginChain” entry has two LoginModules declared. The second LoginModule is invoked only if the first one succeeds as specified by the control flag “requisite”. Allowed keywords for the flag field are “Required”, “Sufficient”, “Requisite”, and “Optional”. The corresponding behaviors are:

Sufficient	If the LoginModule succeeds the authentication process terminates immediately and reports OK;
Requisite	If the LoginModule fails the authentication process terminates immediately and raises an exception;
Required (Default)	The LoginModule must succeed. However the next LoginModule is invoked in any case, even if the LoginModule fails. If it fails an exception will be raised at the end of the authentication process indeed;
Optional	The LoginModule module is not required to succeed. Whether authentication succeeds or fails, the process still continues down the LoginModule list. If the login sequence consists only of Optional modules, at least one module must successfully validate the user.

The control flag field may be followed by options passed to the LoginModule. The options string is a list of space-separated name=value pairs. A common option is debug=true.

The JAAS configuration file may contain several configuration blocs, as shown in the example above. The choice of the bloc to be used is determined at runtime, at the instantiation of the login context. Example:

```
LoginContext logctx = new LoginContext(“loginChain”, callbackHandler);
```

The default Configuration implementation can be changed by setting the value of the "login.configuration.provider" security property (in the java.security properties

file) to the fully qualified name of the desired Configuration implementation class. The java.security file is located in the directory $JAVA_HOME/lib/security, where $JAVA_HOME refers to the directory where the JDK was installed.

```
#
# Class to instantiate as the javax.security.auth.login.Configuration
# provider.
#
login.configuration.provider=com.sun.security.auth.login.ConfigFile

#
# Default login configuration file
#
#login.config.url.1=file:${user.home}/.java.login.config
```

J2EE platforms usually override the configuration provider, and store the JAAS configuration information in an XML file instead. It is recommended to use the vendor tools to manipulate this configuration.

1.3.2 Subject

JAAS uses the term Subject to refer to any entity that requests an action on resources. Therefore both users and computing services represent Subjects. In order to make access control, the Subjects must be identified and authenticated.

1.3.2.1 Principals

During the authentication process a Subject is created and populated with identities objects. A Subject may have many identities. For example, a person may have a name ("John Doe"), a Social Security Number, SSN, ("123-45-6789"), and an email address which distinguish it from other Subjects. JAAS handles identities as Principals. A Principal instance represents a name associated with a Subject. Since Subjects may have multiple names (potentially one for each service with which it interacts), a Subject comprises a set of Principals.

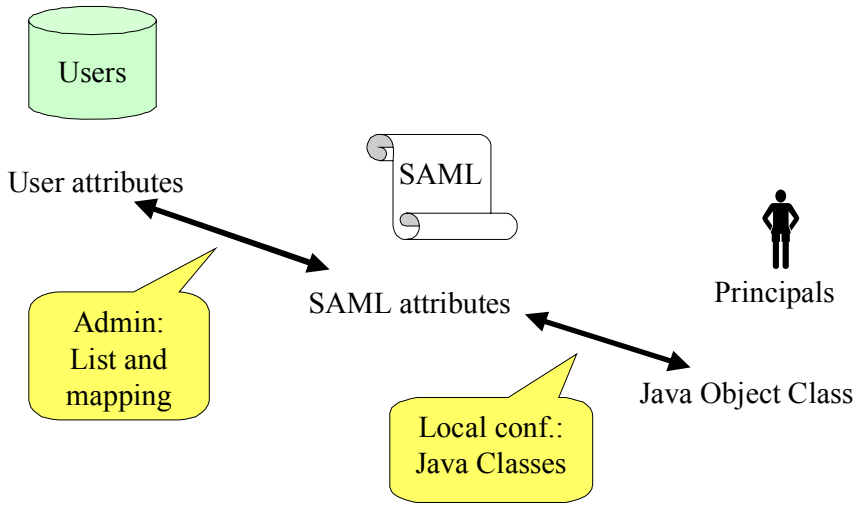
Actually Principal objects may contain attributes that do not uniquely identify a user. For instance a Principal may represent group membership, user’s job etc. The kind of information that is stored in the Principal can be determined from the Principal object class.

```
Set userIDs = subject.getPrincipals(LoginName.class)
```

SAM J2EE provides a facility to map the various SAML attributes to Principal classes. The principal.properties file defines this mapping:

```
# Evidian principalfactory with External Principal
eMail=sample.principal.SamlAttributeMail
Uid=sample.principal.SamlAttributeUId
cn=sample.principal.SamlAttributeCN
```

The attributes included in the SAML assertion are retrieved from the user registry. The following figure shows the link between a user attribute in a directory and its representation in the Subject.



Using both the SAM J2EE administration console and the principal.properties file, it is possible to control the Subject composition.

1.3.2.2 Credentials

Some applications may want to associate other security-related attributes and data with a Subject in addition to Principals. JAAS refers to such generic security-related attributes as Credentials. Credentials might contain data that enables the subject to perform certain activities. Cryptographic keys, for example, represent credentials that enable the subject to sign or encrypt data.

Two sets of Credentials can be associated to a Subject: public Credentials and private Credentials. For example an X509 certificate would be a public Credential and the corresponding private key would be in the private Credentials Set. Access to private Credentials requires permissions.

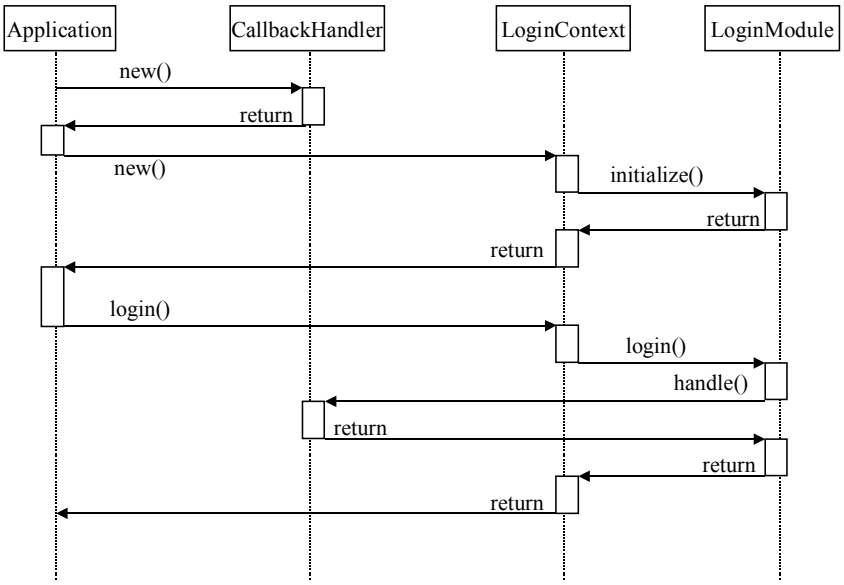
```
Set privKey = subject.getPrivateCredentials(); // security checked
```

JAAS Credentials may be any type of object. Within SAM J2EE, the SAML assertions are public Credentials

1.3.3 Authentication

Authentication represents the process by which a system verifies the identity of an entity. This consists in verifying information that only the entity can know (passwords), have (biometric data) or produce (electronic signatures, tokens).

The LoginModules need to get the data authenticating the users. For that the LoginModules call back the application through a CallbackHandler provided by the application. The CallbackHandler is an object that implements javax.security.auth.callback.CallbackHandler. This interface specifies a method handle() that is called by the LoginModule. This method accepts in input an array of Callbacks, which indicates the data expected by the LoginModule at this phase of the authentication. The JDK comes with a number of predefined callbacks such as NameCallback and PasswordCallback.



1.3.4 Authorization

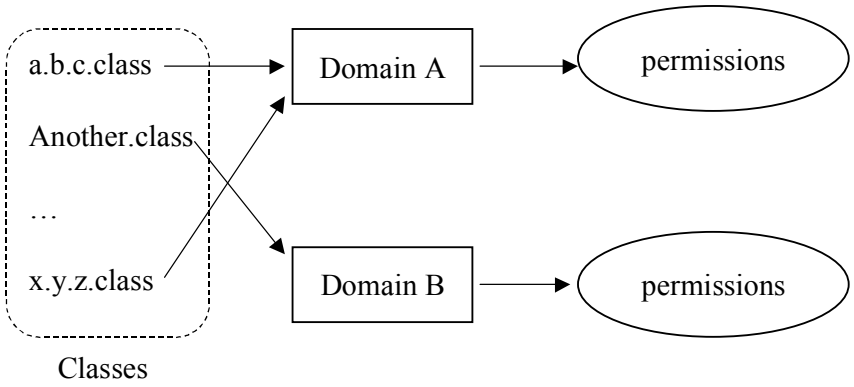
Once authentication has successfully completed, JAAS provides the ability to enforce access controls upon the principals associated with the authenticated subject.

1.3.4.1 ProtectionDomain

Originally, Java focused on environments in which runs potentially untrusted code downloaded from anywhere. The Java security model supported only “CodeSource-based” access control. Permissions were granted to byte code depending from where the code is coming from (‘codeBase’) and who signed it (‘signedBy’).

The permissions granted to code from various sources are defined in a configuration file which runtime representation is a Policy object. The Policy object is not consulted each time that an access decision is to be done. Instead, when the Class Loader loads a class, it gets the set of permissions granted to the class by invoking Policy.getPermission(SourceCode) and stores them in a

ProtectionDomain object. The Java application environment maintains a mapping from code (classes and instances) to their protection domains and then to their permissions, as illustrated by the figure below.



An AccessControlContext object represents the current security context of the caller. It contains an array of ProtectionDomain objects.

1.3.4.2 Access Controls

A thread of execution may occur completely within a single protection domain or may involve an application domain and also the system domain. In this case it is crucial that at any time the application domain does not gain additional permissions by calling the system domain. Otherwise, there can be serious security implications.

In the reverse situation where a system domain invokes a method from an application domain, such as when the system domain calls an applet callback, it is again crucial that at any time the effective access rights are the same as current rights enabled in the application domain.

In other words, a less “powerful” domain must not be able to gain additional permissions as a result of calling or being called by a more powerful domain. For this purpose, the basic rule to determine the permission set of an execution thread is making the intersection of the permissions of all protection domains traversed by the execution thread.

Access control checks are actually performed by the AccessController. Its checkPermission method traverses the call stack of the current threads of execution and checks whether all classes in the call stack have the requested permissions. It takes a snapshot of the execution stack to determine the list of ProtectionDomain objects to check against.

The previous rule would be too restrictive if there was no escape mechanism that allows a class to run some code with the privileges assigned to its protection domain. A piece of trusted code can be marked as a privileged block to temporarily enable access to more resources. See java.security.PrivilegedAction interface. When the AccessController walks the calling chain in the execution stack, it stops to follow the links when it encounters code marked as privileged.

1.3.4.3 JAAS Principal-based Access Controls

JAAS Authorization extends this model with “Subject-based” access control. Permissions can be granted to users executing the code. JAAS authorization component works in conjunction with the existing Java 2 “CodeSource-based” access control model. JAAS policy extends the Java 2 policy with the relevant Subject-based information. Therefore, permissions recognized and understood in Java 2 (java.io.FilePermission and java.net.SocketPermission, for example) are equally understood and recognized by JAAS. Although the JAAS security policy physically resides separately from the existing Java 2 security policy, the two policies should be treated as one logical policy.

JAAS authorization extends the Java 2 policy file syntax. Example:

```
grant
        codeBase "www.acme.com",
        signedBy "myAuthority",
        principal LoginNameClass "duke",
        principal GroupMembershipClass "administrator",
        …
{
        permission java.io.FilePermission "/etc", "read, write",
        permission java.io.FilePermission "/user/admin", "read, write",
        …
        ;
};
```

In this sample, the permissions are granted to code loaded from www.acme.com and signed by the alias myAuthority, and executed by duke or an administrator. Duke is identified by a Principal of class LoginNameClass whose getName method returns “duke”.

JAAS treats roles and groups simply as named principals. Therefore access control can be imposed upon roles and groups just as they are with any other type of principal.

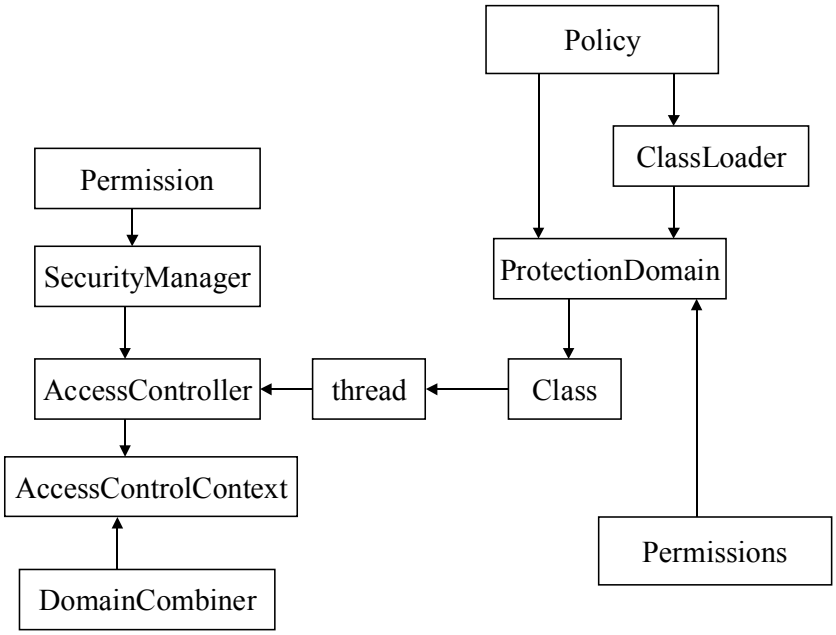
For flexibility, the JAAS policy also permits the Principal class specified in a grant entry to implement the PrincipalComparator interface. The permissions for such

entries are granted to any subject for which the PrincipalComparator implies method returns true.

```
public interface PrincipalComparator {
        boolean implies(Subject subject);
}
```

The PrincipalComparators can be used to support role hierarchies. For example an administrator inherits all permissions granted to user role. So the implies() method invoked on a Principal which represents a role, and is instanced for “users”, should returns true if the Subject has an administrator role (has a Principal object representing administrator role).

To associate a Subject object with a thread of execution, the AccessControlContext object accepts a DomainCombiner that is called during the authorization process. The purpose of the combiner is to modify the ProtectionDomain objects associated with the classes on the thread activation stack. The JAAS SubjectDomainCombiner logically extends each ProtectionDomain object found on the stack so that it will contain the Principal[] associated with the Subject.



1.3.4.4 Permissions

The permission classes represent access to system resources. The java.security.Permission class is an abstract class and is sub-classed, as appropriate, to represent specific accesses.

New permissions are sub-classed either from the Permission class or one of its subclasses, such as java.security.BasicPermission. Sub-classed permissions (other than BasicPermission) generally belong to their own packages. Thus, FilePermission is found in the java.io package.

A permission typically has a name (often referred to as a "target name") and, in some cases, a comma-separated list of one or more actions. For example, the following code creates a FilePermission object representing read access to the file named **abc** in the **/tmp** directory:

```
perm = new java.io.FilePermission("/tmp/abc", "read");
```

Important

The above statement creates a permission object. Permission objects represent, but do not grant access to, a system resource. Permission objects are constructed and assigned ("granted") to code based on the policy in effect. When a permission object is assigned to some code, this code is granted the permission to access the system resource specified in the permission object, in the specified manner. A permission object may also be constructed by the current security manager when making access decisions. In this case, the (target) permission object is created based on the requested access, and checked against the permission objects granted to and held by the code making the request.

1.3.4.5 LoginModule Permissions

A javax.security.auth.AuthPermission is for authentication permissions. An AuthPermission contains a name (also referred to as a "target name") but no actions list; you either have the named permission or you don't.

Currently the AuthPermission object is used to guard access to the Subject, SubjectDomainCombiner, LoginContext and Configuration objects.

Possible AuthPermission are: doAS, modifyPrincipals, modifyPublicCredentials, modifyPrivateCredentials etc.

LoginModules must be granted AuthPermission, otherwise they will not be able to do their job.

1.3.5 SAM J2EE Login methods

1.3.5.1 Methods

Control Flag in JAAS configuration	Description of Login method
id-password	The LoginModule authenticates the user given a userID and a password. A SAML assertion is available as a result of authentication. The CallbackHandler is called to get the userID and password. The assertion attributes are translated into Principals.
id-password-with-change	This method extends the id-password method with the possibility to change the password. The Callbacks array contains an additional entry to specify the new password.
sdm-token	This method is intended to workstations configured with AccessMaster SAM SE. The credentials (token) are retrieved directly from the SDM context. The user has to be already authenticated with the SDM. There is no need of a CallbackHandler. A SAML assertion is available as a result of authentication.
auth-sdm	This method is intended to workstations configured with AccessMaster SAM SE. This method wraps the SDM authentication. The CallbackHandler is called to get the userID and password. No SAML assertion available.
auth-sdm-with-change	This method extends the auth-sdm method with the possibility to change the password. The Callbacks array contains an additional entry to specify the new password.
check-sdm	This method is intended to workstations configured with AccessMaster SAM SE. This method only checks that the user is already authenticated within the SDM.
token-checking	This method verifies a SAML assertion and populates the Subject. The assertion attributes are translated into Principals. The assertion is retrieved from the CallbackHandler.
auth-principal	This method must be invoked with a Subject that already contains a Principal. The Principal must contain an assertion. This method is intended to J2EE environment, JAAS is called by the application itself. There is no CallbackHandler.

In order to enforce the password policy, the authentication fails when the password is expired. In this case you have to authenticate with the id-password-with-change method and provide the new password. The authentication and the password change are done in the same operation.

SAML LoginModule supports password replacement. When password replacement is activated it adds a second `PasswordCallback` with prompt "`New Password:`". When callback handler fills this callback, SAML LoginModule sends a request to the authentication server to replace the password. The authentication server replaces the password after authenticate user with current one.

SAML assertions have a validity period. When a SAML assertion is expired, the token-checking and auth-principal methods can renew it. This facility is requested by adding the option refresh=true in the JAAS configuration. The assertion is reconstructed from the user registry. If the user registry has been updated, the renewed assertion will reflect the changes.

For details of how to configure SAML LoginModule to support the SAML assertion refresh feature, see Section 3, "Configuring SAML Login Module".

In the auth-principal method, the Principal passed to the LoginModule must implements the SecurityContext interface. This interface defines two methods:

```
setAssertion(byte[] assertion);
byte[] getAssertion();
```

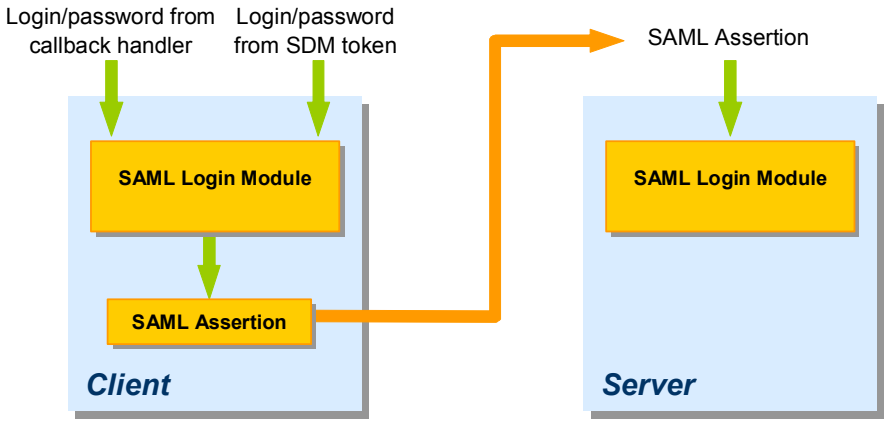
The LoginModule will invoke the getAssertion method to get the SAML assertion. Then process will be similar to token-checking method. The setAssertion method is intended to be used to create the corresponding Principal: if a Principal class implements the SecurityContext interface, the LoginModule will call setAssertion method passing it the SAML as a byte array. This facility is intended to J2EE platform (for more details, see Section 1.5.3 "Invoking JAAS within Applications"), but it can be used in other environments.

1.3.5.2 Typical Usage

This section illustrates one of the possibilities of LoginModule use. The configuration is a client/server use case. In this architecture two SAML LoginModules are installed:

The first one is on the client. It uses the login/password or the SDM token method. After authentication, the user application retrieves SAML assertion from public credentials and is responsible for sending it to server.

The second one is on the server. It uses the SAML assertion method and receives assertion from the client. The secure transmission of the assertion is out of the scope of SAML LoginModule. It gives you the ways to retrieve on the client and to handle it on server.



1.3.6 Adapting to Application Environments

As exposed in section 1.3.2.1 (“Principals”), the Subject may be populated with several kinds of information. The standard way to distinguish between the types of information is to use one Principal class for each type. Evidian LoginModule offers a flexible facility to specify the class to be used for each type of SAML attribute.

The principal.properties file defines the mappings between SAML attributes and the corresponding Principal classes.

```
# Evidian principalfactory with External Principal
eMail=sample.principal.SamlAttributeMail
Uid=sample.principal.SamlAttributeUId
cn=sample.principal.SamlAttributeCN
```

The credential.properties file defines the class to be used for storing the SAML assertion as a string.

```
credential=com.evidian.security.auth.saml.SamlAssertion
```

Actually the LoginModule uses a factory to create Principal and Credentials object. The principal.properties and credential.properties files are used by the default factory. The default factory should be appropriate for most of the cases. But it is possible to develop a dedicated factory if the default behavior is not adapted.

The factory is declared by a system property. Section 4, "Developing your own Principal and Credential Classes" describes how to reuse Evidian factory or implement your own one.

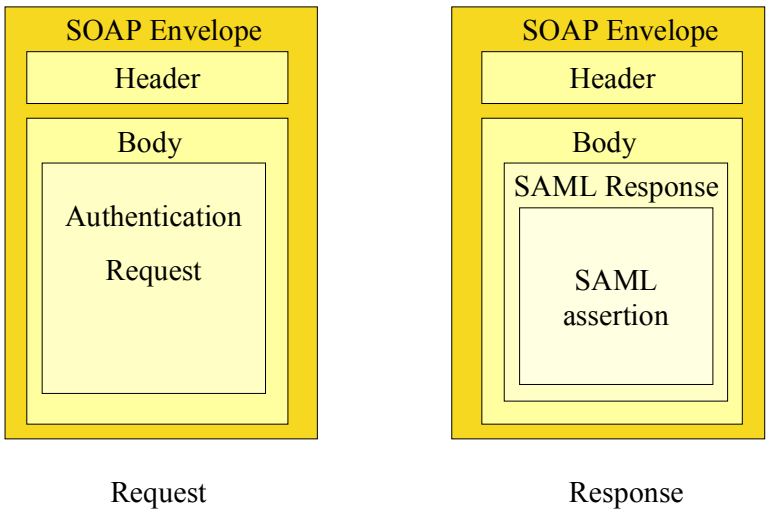
1.4 Web Service Interface

JAAS is an API to get and verify SAML assertions, and extract attributes from those assertions. In some environments it is not possible to require this API. The SAML provider Web Service interface is an alternative to get the user assertions.

1.4.1 SOAP Message Format

The authentication requests and responses are put in the Body part of a SOAP envelope.

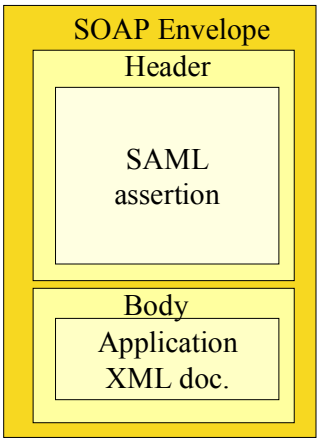
The assertion returned is encapsulated in a SAML Response.



Request Response

The WS-SAML provider interface uses the SOAP “document-style” flavour.

The assertion returned by the Web Service can be used in a subsequent request to an application Web Service as a token. See the SAML Token Profile defined in WS-Security. Note that the SAML assertion is included in the SOAP header (it was in the SOAP body in the WS-SAML response).



WS Request

1.4.2 SAML Authentication Web Service

This Web Service is aimed at getting the SAML assertion from a user after having successfully authenticated him.

1.4.2.1 Authentication Request

Here is an example of an authentication request.

```
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
<authn:Authentication
      xmlns:authn="http://evidian.com/security/auth/saml/1.0"
>
```

```
        <authn:AuthnData>
              <authn:Version>1.0</authn:Version>
              <authn:UserName>jeandel</authn:UserName>
              <authn:Password>*******</authn:Password>
              <authn:Domain>evidian</authn:Domain>
              <authn:Policy>strong</authn:Policy>
        </authn:AuthnData>
</authn:Authentication>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

1.4.2.2 Authentication Response

Here is an example of an authentication response.

```
<SOAP-ENV:Envelope
      xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
<samlp:Response xmlns:samlp="…" xmlns:saml="…" xmlns:ds="…">
      <Status>
            <StatusCodevalue="samlp:Success"/>
      </Status>
      <saml:Assertion>
            <saml:AuthenticationStatement>
                  …
            </saml:AuthenticationStatement>
            <ds:Signature> … </ds:Signature>
      </saml:Assertion>
</samlp:Response>
</SOAP-Env:Body>
</SOAP-ENV:Envelope>
```

1.4.3 Certificate to SAML Web Service

This Web Service is aimed at getting a user SAML assertion from its X509 certificate.

1.4.3.1 Assertion Request

Here is an example of an assertion request.

```
<SOAP-ENV:Envelope
xmlns:SOAP-ENV=”http://schemas.xmlsoap.org/soap/envelope/”>
<SOAP-ENV:Body>
<xsd1:CertificatePresentation>
```

```
      <CertData>
            <Version>1.0</Version>
            <X509Certificate>
            -----BEGIN CERTIFICATE-----  -----END
CERTIFICATE-----
            </X509Certificate>
            <Policy>Authentication with certificate and
SAML</Policy>
            <Check>CheckOcspResponse</Check>
            <OCSPResponse>
            KioqKioqKioqKiogT0NTUCBSRVNQT05TRSAqKioqKioqKioqK
g==
            </OCSPResponse>
      </CertData>
</xsd1:CertificatePresentation>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

1.4.3.2 Assertion Response

Here is an example of an assertion response.

```
<SOAP-ENV:Envelope
      xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
<samlp:Response xmlns:samlp="…" xmlns:saml="…" xmlns:ds="…">
      <samlp:Status>
            <samlp:StatusCode Value="Success"/>
      </samlp:Status>
      <saml:Assertion>
            <saml:AuthenticationStatement>
                  …
            </saml:AuthenticationStatement>
            <ds:Signature> … </ds:Signature>
      </saml:Assertion>
</samlp:Response>
</SOAP-Env:Body>
</SOAP-ENV:Envelope>
```

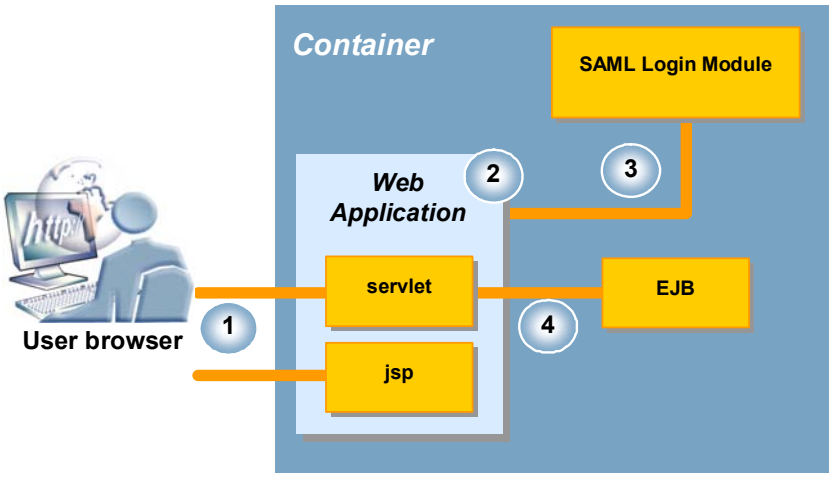
1.5 Using JAAS LoginModule on J2EE Platforms

1.5.1 J2EE Architecture

In the J2EE context, the container performs the authentication. The user application retrieves identity from `HTTPServletRequest.getUserPrincipal` (as a String)

in servlet context and from `SessionContext.getCallerPrincipal` in EJB context (as a principal whose class is platform dependent).

Authentication is triggered by resource protection declared in servlet entry of web application descriptor and handled by container using an HTTP 401 or a form based.



Process of authentication:

1	User browser connects to the web application
2	In the web application, the descriptor resource is protected by a container so JAAS module is invoked
3	After successful authentication the user identity is returned to the web application
4	A call is made to an EJB. The identity is propagated to the EJB by the container

1.5.2 Integration within J2EE Containers

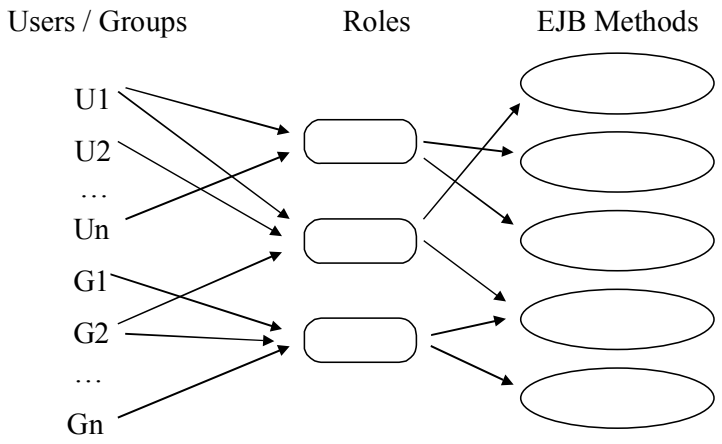
J2EE platforms offer a framework for application servers. They provide many technical services for applications. Among the available services are the security services.

J2EE containers control access to the resources they host. The control is fine grained:

Within Web containers, resources are servlets and JSP pages;

Within EJB containers, resources are methods of the class instances.

The access control is done by the framework. The security is declarative. For each application, roles are defined. The access to the resources is granted to the roles. The roles are assigned to individual users and to group of users. The following figure illustrates this.



The users must be first authenticated. The J2EE platforms are in charge of the authentication. The acquisition of the userId and password is done by the containers. Web containers can automatically send an HTML form to prompt the user for credentials. For the effective authentication, commercial and open source J2EE platforms support JAAS. JAAS is invoked by the container, and the application has nothing to do.

In order to map user and groups to roles, the containers have to know which Principal represents the user name and which ones represent a group membership. Each J2EE platform vendor has its own specific conventions. For instance, BEA WebLogic recognizes the userID Principal as the one which implements the WLSUSER interface.

Evidian LoginModule facilitates the customization of Principals classes to adapt to the conventions used by the platform vendors. See 1.3.6 “Adapting to Application Environments” and the chapters dedicated to each vendor.

1.5.3 Invoking JAAS within Applications

Applications may have to get the SAML assertion established in the login phase. For example a Web Service consumer might include the assertion in a SOAP request.

The easiest way to do that is through the Principal object. The methods `getUserPrincipal` in servlets, and `getCallerPrincipal` in EJB, return a Principal object that represents the entity running the current thread of execution. On nearly all platforms, the Principal returned originate from the JAAS LoginModule. If it implements the SecurityContext interface, it contains a SAML assertion. Then the application can get the assertion from the Principal invoking the getAssertion on it, or using the LoginModule with the auth-principal method. This last case offers an easy way to access to the attributes contained in the assertion.

The method exposed in the previous paragraph does not work for Oracle OC4J as the methods `getUserPrincipal` and `getCallerPrincipal` return a Principal that does not originate from the JAAS LoginModule, but an object that is forged by the platform. So for OC4J you have to use another method: get the Subject from the AccessController. Here is a sample of code:

```
AccessControlContext ctx = AccessController.getContext();
Subject sub = Subject.getSubject(ctx);
Set principals = sub.getPrincipals(YourSAMLPrincipal.class);
```

2. Installing SAML Login Module

2.1 Configuring SAM J2EE Authentication Server

This section describes the configuration of the authentication server part of SAM J2EE.

Before starting SAM J2EE server must be installed and running (refer to *SAM Web and SAM J2EE Installation Guide*).

Procedure

1. Check that you have a valid SAM J2EE license in the following file **$LPF_ROOT_DIR/config/license.conf**
2. In a browser, type the following URL to start the SAM J2EE console:(root URL of your SAM J2EE Administration Server) **http://my.company.com:9119**.

   Note: To know the port used by SAM J2EE Administration Server, use the following command: `$LPF_ROOT_DIR/bin/lpfstatus`.

3. In the **Keys and Certificates** tab, create a server key for the Authentication Server and a signature key for SAML assertion signature.
   You may have to create Certification Authority objects first for the authorities issuing these certificates. Enable the boxes "trusted for server certificates" and "trusted for signature certificates". For more details, see Section 13, "Managing Keys and Cetificates" in *SAM Web and SAM J2EE Administrator's Guide*.
4. In the **Authentication** tab, specify the user directories and multi-user directory that will be used for authentication.
5. In the **Authentication** tab, edit the Authentication Policy **Authentication with form and SAML** and specify the Multi Users Directory defined above. You may also create a new Authentication Policy, but do not forget to edit the **uri** attributes in `security.cfg` file to reflect the path of this new Authentication Policy.

6. In the **Authentication** tab, edit the **Built-in Authentication Server**.
   In the **General** tab, set protocol, port and server key.
   In the **SAML** tab, specify the signature key created above and a service provider identifier.
   For more details, see Section 10, "Managing Authentication" in*SAM Web and SAM J2EE Administrator's Guide*.
7. In the **SSO and Injection** tab, create one Injection Data in **Injection Database for SAML** for each attribute that should be included in SAML assertions.

   Note: Also, you can create your own Injection Data Base Object but do not forget to declare it in the SAML tab of your Authentication Policy.

   For more details, refer to Chapter 9, “Managing Data Injection” in SAM Web and SAM J2EE Administrator's Guide.

2.2 SAML Login Module Files

This section describes the files of SAML Login Module and their location, and details how SAML Login Module gets its resources. It also describes the configurable options and how to use them to define SAML Login Module behavior.

Note: By resources, we mean the **.properties** configuration files as well as directories of data files.

2.2.1 samllogin.jar

This is the archive file of SAML Login Module. For more information about how Java classes are found, see **http://java.sun.com/j2se/1.4.2/docs/tooldocs/findingclasses.html** (URL valid in June 2005).

Select a class loader to load **samllogin.jar** and add an entry to its `classpath` definition.

EXAMPLE 1:

Defining the `CLASSPATH` environment variable for a file located into the **C:\java\lib** directory:

```
CLASSPATH=%CLASSPATH%;C:\java\lib\samllogin.jar
```

Note

Under Windows, the semi-colon is used as a separator.

EXAMPLE 2:

Defining the `CLASSPATH` environment variable for a file located into the **/usr/local/java/lib** directory:

```
CLASSPATH=$CLASSPATH:/usr/local/java/lib/samllogin.jar
```

Note

Under UNIX, the colon is used as a separator.

Important

If you make an error at this step as a typing error, you will get a **ClassNotFoundException** error.

2.2.2 Security Module Shared Library

The Security Module resides in a shared library which name is:

- **samllogin.dll** on Windows.
- **libsamllogin.sl** on HP-UX.
- **libsamllogin.so** on other Unix systems.

The search rules for shared libraries are OS dependant:

- Windows looks for shared libraries into the current directory (`.`) and the directories listed in the `PATH` environment variable.
- Any Solaris system searches into **/usr/ccs/lib** and **/usr/lib**, but an installed Solaris 9 searches also in **/usr/platform/SUNW,Ultra-250**. To add your own shared library you can use the `LD_LIBRARY_PATH` environment variable.
- HP-UX search libraries in the path specified by `SHLIB_PATH` environment variable.

If you don't want to modify your environment variables, specify the library path as a system property named **java.library.path**.

EXAMPLE 1: PATH DEFINITION ON WINDOWS

Example for a file located in the **C:\java\bin** directory:

```
PATH=%PATH%;C:\java\bin
```

EXAMPLE 2: SYSTEM PROPERTY DEFINITION ON WINDOWS

Example for a file located into the **C:\java\bin** directory:

```
-Djava.library.path=C:\java\bin
```

EXAMPLE 3: PATH DEFINITION ON WINDOWS

Example with `LD_LIBRARY_PATH` for a file located into the **/usr/local/java/bin** directory:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/java/bin
```

Important

If you make an error at this step, a **LoginException** error with the message **java.lang.UnsatisfiedLinkError: no samllogin in java.library.path** is raised.

For more information about the search mechanism, see **http://java.sun.com/docs/books/jni/html/start.html** (URL valid in June 2005).

2.2.3 security.cfg

2.2.3.1 Locating security.cfg

This is the Security Module configuration file.The Security Module explores a sorted list of directories and stops when it finds an existing non-empty file.

First, it looks up for the **SAMLCONFIG** environment variable. If it exists, **SAMLCONFIG** can point:

- To a directory where it looks up for **security.cfg**.
- To a file enabling you to use another name than **security.cfg**.

Then it looks through a list of directories until it finds an existing non empty **security.cfg** file.

The following files are searched:

- Under UNIX:
  - $SAMLCONFIG
  - $SAMLCONFIG/security.cfg
  - $HOME/saml/security.cfg
  - /etc/conf/saml/security.cfg
- Under Windows:
  - %SAMLCONFIG%
  - %SAMLCONFIG%\security.cfg
  - %USERPROFILE%\saml\security.cfg
  - %WINDIR%\%USERNAME%\saml\security.cfg
  - %WINDIR%\saml\security.cfg
  - %SystemDrive%\%USERNAME%\saml
  - %SystemDrive%\saml

2.2.3.2 Configuring "security.cfg"

security.cfg is an XML file where some parameters should be set for security module operation:

- Directories for working files.
- URLs of authentication server and CA distribution point.
- Authentication policy.

Procedure

Important

Pay attention to the protocol selected in your authentication server (`HTTP` or `HTTPS`).

1. Configure the **security.cfg** file as described in the following table.

   The template provided for **security.cfg** contains placeholder values in the form `__value__` that should be instanciated when configuring the security module; these values are summarized below:

Parameter	Description
`__saml_base__`	Root of the tree structure of the security module, used by other directory relative parameters.
`__CA_path__`	Directory where the security module looks for CA certificates (either absolute or relative to `__saml_base__`); generally set to `ca.d`.
`__CRL_path__`	Directory where the security module looks for Certificate Revocation Lists (either absolute or relative to `__saml_base__`); generally set to `crl.d`
`__auth_url__`	URL of the authentication server, e.g. **https://my.company.com:9131/pxpadmin/bin/authform.cgi**
`__renewal_url__`	URL of the assertion renewal server, e.g. **https://my.company.com:9131/pxpadmin/renewal/**
`__CA_url__`	URL used to download CA certificates and CRLs, e.g **http://my.company.com:9119/bin/ca.sh**. If SSL is activated, the download is automatic when certificates are missing. This URL is located on the SAM J2EE Administration Server.
`__user__`	Identifies the SAML Login Module for assertion renewal
`__password__`	Clear-text password for this user
`__cpassword__`	Crypted password for this user; use `samlpwdcrypt` utility to generate **cpassword**. **password** and **cpassword** are mutually exclusive. To encrypt the password, use the **samlpwdcrypt** utility as follows: `samlpwdcrypt -p __password___ -f security.cfg` **samlpwdcrypt** replaces any existing **password** or **cpassword** attributes in the **<authentication>** element of `security.cfg` by the newly computed **cpassword** value.

2. Check the syntax of **security.cfg** with the following command:

   ```
   samltestcfg "%USER_PROFILE%\saml\security.cfg"
   ```

   > In case of error, the output ends with:
   > `SAML configuration failed: <diagnose information>.`

2.2.3.3 Activating Security Module Traces

In **security.cfg**, in the `<general>` tag (which belongs to the `<samlConfig>` tag), add the following attributes:

`TraceFile`: add the name of the trace file

`TraceLevel`: add value `0xf97f`

EXAMPLE:

```
<samlConfig>
  <general
      type="appclient,verifier"
      base="C:\software\saml\apps"
      traceFile="saml.trc"
      traceLevel="0xf97f">
  </general>
. . .
```

Changes made to this file will take effect on the next JVM restart since the shared library is loaded once for all by a static bloc.

2.2.4 CA Certificates and CRL

The installation of the Certificate Authorities accepted to emit SAML assertion signature and SSL server certificates is automatic when the certificate distribution URL on server uses SSL.Nevertheless, you can download the certificates and CRL of your trusted Certification Authorities in the right directory using the following command:

```
samlloadca ~/saml/security.cfg
```

Before using the **samlloadca** command you must have configured **security.cfg**.

A list of directories is defined in **security.cfg**, which are either absolute or relative to the attribute base of the general tag in **security.cfg**. SAML login module creates these directories if they do not exist.

2.2.5 JAAS Configuration File

JAAS configuration file is indicated by the value of a system property **java.security.auth.login.config**, as described in Step 7 of the following URL: **http://java.sun.com/j2se/1.4.2/docs/guide/security/jaas/JAASLMDevGuide.html** (URL valid in June 2005).

EXAMPLE:

To load the JAAS configuration from the **jaas.config** file located in the **C:\java\conf** directory, add the following to the JVM options:

```
-Djava.security.auth.login.config=C:/Java/conf/jaas.config
```

There are SAML Login Module specific options indicated in the JAAS configuration file. All these options have a default value:

- `debug`: on `true`, activates traces to standard output. The default value is `false`.
- `request`: selects the authentication method. Authorized values are:
  - `auth-principal`: to authenticate user based on assertion retrieve from existing subject populated by the previous authentication.
  - `id-password`: to authenticate user based on login/password retrieved from callback handler invocation. This is the default value.
  - `id-password-with-change`: same as `id-password` except that callbacks array sent to handler contains a second `PasswordCallback` for password replace.
  - `sdm-token`: to authenticate user based on information retrieved from the Security Data Manager. As for `auth-principal` this request does not invoke callback handler.
  - `token-checking`: to authenticate user based on assertion retrieve from callback handler invocation.
  - `auth-sdm`: SAML Login Module requests Security Data Manager to authenticate a user based on login/password retrieved from callback handler invocation. No SAML assertion will be generated in this case.
  - `auth-sdm-with-change`: same as `auth-sdm` except that callbacks array sent to handler contains a second `PasswordCallback` for password replace.
  - `check-sdm`: SAML Login Module requires Security Data Manager to check that the user is already authenticated. As for `sdm-token` this request does not invoke callback handler.
- `refresh`: on `true`, renew expired SAML assertion. The default value is `false`.

EXAMPLE:

To configure SAML Login Module for an application to authenticate user based on login/password, the configuration file might look like this:

```
AppName {
    com.evidian.security.auth.login.SamlLoginModule
    REQUIRED debug=true refresh=false request=id-
    password;
};
```

where `AppName` should be whatever name the application uses to refer to this entry in the login configuration file. The application specifies this name as the first argument to the `LoginContext` constructor.

2.2.6 .Properties Files

The **principal.properties**, **credential.properties** and **cache.properties** files are loaded as resources through class loaders. This way SAML Login Module is disk position independent and prevents at deployment time any edition of directory values.

The installation of these files is similar to the installation of the SAML login module archive as described in Section 2.2.11, "samllogin.jar".

If SAML Login Module cannot find a .properties file in the specified location, it checks **SAMLCONFIG** if this environment variable is defined. If **SAMLCONFIG** points to a file (even a non-existing one), SAML Login Module looks for .properties files in the parent directory. If no .properties file is found, a **SamlLoginException** error is raised.

2.2.6.1 principal.properties

Principal Factory abstracts SAML Login Module from classes used to populate Subject. The Evidian implementation of Principal factory reads its configuration from **principal.properties**. It consists of a mapping between SAML attribute and principal classes:

- `__principal_attribute__`, is a SAML attribute in assertion whose value is used to instantiate a principal.
- `__classnames__`, is a space separated list of Principal classes that are instantiated with the value of the `__principal_attribute__` attribute.
- `__principal_attribute__` must match the "SAML Assertion tag" attribute declared in the correspondent **Data Injection** (see section 2.1, "Configuring SAM J2EE Authentication Server").

For more details on how to declare SAML Attribute with SAM-Web console, see to Chapter 9, "Managing Data Injection" in *SAM Web and SAM J2EE Administrator's Guide.*

Important

To be supported, your class must implement the **com.evidian.security.auth.interfaces.SecurityContext** interface.

2.2.6.2 credential.properties

Credential Factory abstracts SAML Login Module from classes used to populate the Subject. The Evidian implementation of Credential factory reads its configuration from **credential.properties**:

`__classname__` is the name of the Credential class to be instantiated with SAML assertion.

2.2.6.3 cache.properties

SAML Login Module uses a cache mechanism to reduce time response. When an assertion is generated, it is put in a cache file. SAML Login Module sets the "last modification date" of a cache file to its expiration date. All cache files with a "last modification date" earlier than now are expired assertions.

SAML Login Module Responses are retrieved from cache only when authentication method is based on assertion (auth-principal or token-checking method) and the cached assertion is still valid. If an expired assertion is reused, its corresponding cache is removed from the disk.

Cache is encrypted to prevent users from granting privileges by corrupting its content.

- Use `__seed__` to replace the secret key used to encrypt cache. Its value is an alphanumeric string of 16 characters at least. If less SAML Login Module runs with a circular padding reducing key strength. If missing, SAML Login Module runs with a hard wired seed value.

  Important

  As `seed` is a confidential information `__cache.properties__` access must be restricted on disk.

- A "Cache external purge" is needed to remove expired assertions. To find and delete those files under Windows, create a new schedule task that uses the `forfiles` command from the resource kit:

  ```
  C:\WINNT\system32\CMD.EXE /c forfiles -
  p"%userprofile%\saml\cache" -d-1  -c"CMD /C del @FILE"
  ```

  Note: There is no space between the option (-p) and the value.

- To find and delete those files under UNIX, add an entry to your crontab using `crontab -e` as follows:

  ```
  00 4 * * * find $HOME/saml/cache -type f -mtime +1 -exec rm
  {} \;
  ```

  Note:
  - You may need to set the EDITOR environment variable into vi.
  - Prefer environment variables like %userprofile% or $PATH to literal. To protect against unsupported space in file name, enclose variable **%userprofile%** with double quotes.

3. Implementing SAML Login Module With a Standalone Java Application

This chapter describes through an example all the requirements and configuration steps of a standalone java application implementing the Evidian SAML Login Module. Also it describes the configurable options and how to use them to define SAML Login Module behavior.

The sample application detailed in this chapter is an update of the j2se 1.4.2 authentication tutorial application code for using Evidian SAML Login Module.

3.1 Description

The purpose of this sample application is to authenticate a user by using Evidian SAML Login Module and recover user identity into a Subject Object.
The use of SAM J2EE allows to configure the instances of Principals classes and defines user's attributes to returned into these classes. Once authenticated, the authentication server delivers an SAML assertion with the declared user's attributes. The SAML Login Module recovers this assertion and generates a Subject composed with principals from user's attributes and a public credential from SAML assertion.

For this application the user is stored in Built-in User's directory.

3.2 Installing SAM J2EE

Before Starting

Download the last version of Evidian SAML Login Module.

For this application, all configuring instructions are given for Windows plateform.

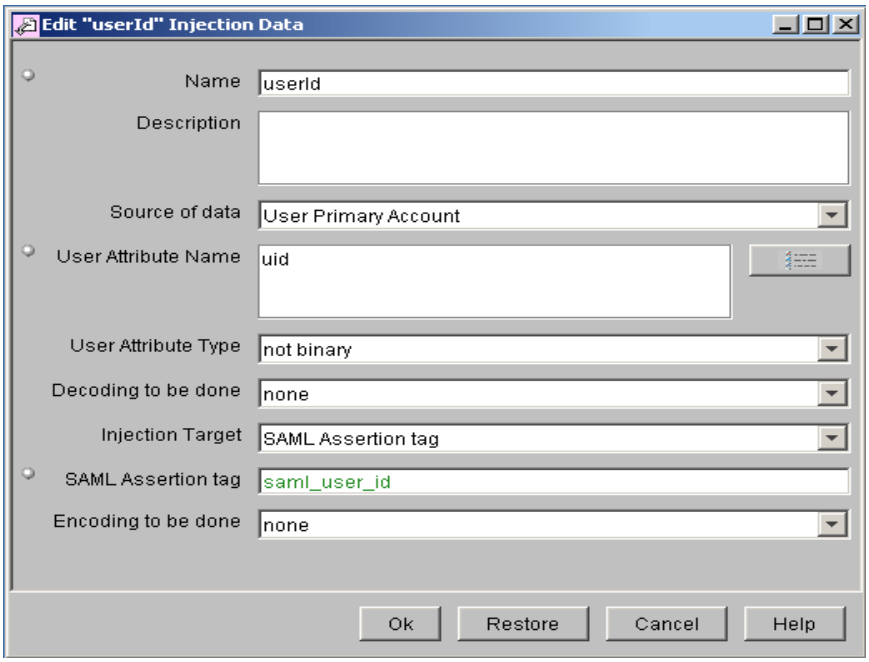
3.2.1 Configuring SAM J2EE Authentication Server

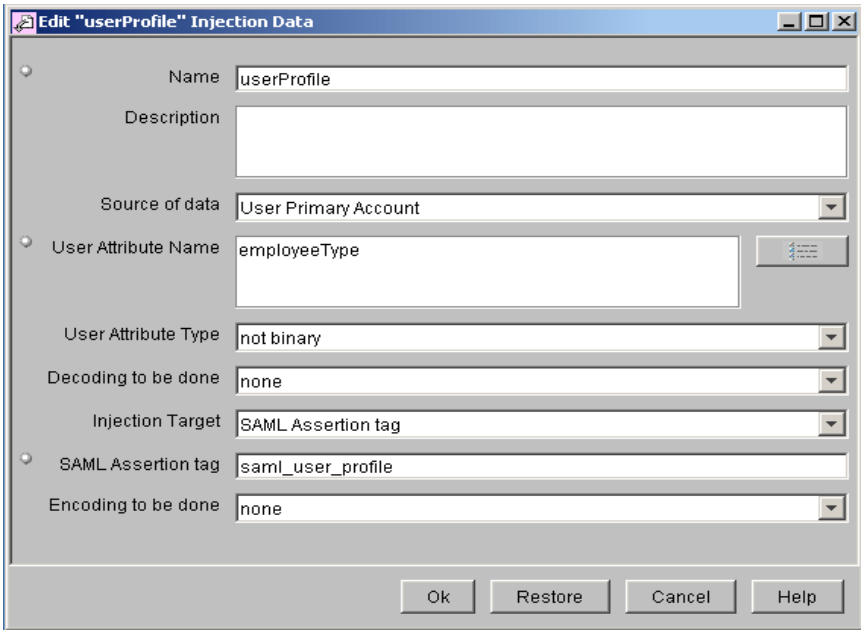
Before Starting

- Make sure that SAM J2EE is installed and the Authentication Server is configured to deliver assertions, as mentionned in Section 2.
- Make sure that the the Built-in User's directory is the SAM J2EE Users Directory.

Procedure

In the **SSO & Injection** tab, **Accounts Bases for Single Sign-On/Data Injection/Data Injection base for SAML,** create with SAM-Web console two Data Injection Object "userId" and "userProfile" with the following attributes:





3.2.2 Installing SAM J2EE AuthClient

Unpack Evidian SAML Login Module package in a selected directory. For this example, we named this directory APP_HOME.

This produces the following:

APP_HOME /bin/samlloadca.exe

APP_HOME /bin/samllogin.dll

APP_HOME /bin/samlpsw.exe

APP_HOME./bin/samlpwdcrypt.exe

APP_HOME /bin/samltestcfg.exe

APP_HOME /config/cache.properties.tmpl

APP_HOME /config/credential.properties.tmpl

APP_HOME /config/jaas.config.tmpl

APP_HOME /config/principal.properties.tmpl

APP_HOME /config/security.cfg.tmpl

APP_HOME /lib/samllogin.jar

3.2.3 Configuring SAM J2EE AuthClient

3.2.3.1 samllogin.jar

Set the environment variable CLASSPATH as follows:

```
CLASSPATH=%CLASSPATH%;%APP_HOME%\lib\samllogin.jar
```

3.2.3.2 Security Module shared library

Set the environment variable PATH as follows:

```
PATH=%PATH%;%APP_HOME%\bin
```

3.2.3.3 security.cfg

- This file is written from the **%APP_HOME %/config/security.cfg.tmpl** file. (see Section 2.2.3, "security.cfg")
- Set the SAMLCONFIG environment variable as follows:

  ```
  SAMLCONFIG =%;%APP_HOME%\config
  ```

3.2.3.4 jaas.config

This file is written from the **%APP_HOME% /config/jaas.properties.tmpl** file(see Section 2.2.5, "JAAS Configuration File").

```
SamlEntry {

 com.evidian.security.auth.login.SamlLoginModule required
 debug=true request=id-password;

};
```

3.2.3.5 principal.properties

- This file is written from the **%APP_HOME %/config/principal.properties.tmpl** file (see Section 2.2.6, "Properties Files").

  ```
  saml_user_id=sample.principal.SampleIdPrincipal
  saml_user_profile=sample.principal.SampleProfilePrincipal
  ```

- Set the CLASSPATH environment variable as follows:

  ```
  CLASSPATH=%CLASSPATH%;%APP_HOME%\config
  ```

3.2.3.6 credential.properties

This file is written from the **%APP_HOME% /config/credential.properties.tmpl** file (see Section 2.2.6, "Properties Files").

```
credential=com.evidian.security.auth.saml.SamlAssertion
```

3.2.3.7 cache.properties

This file is written from the **%APP_HOME% /config/cachel.properties.tmpl** file (see Section 2.2.6, "Properties Files").

```
seed= lkafm3riv8po4wec
```

3.3 Writing the Application Code

The code for this application is composed of three files:

- **SampleSaml.java** contains the sample application class (Sample) and another class used to handle user input (MyCallbackHandler).
- **SampleIdPrincipal.java** is a sample class implementing the **java.security.Principal** interface. It is used by SAML LoginModule to instantiate with the user "uid" attribute.
- **SampleProfilePrincipal.java** is a sample class implementing the **java.security.Principal** interface. It is used by SAML LoginModule to instantiate with the user "‘employeeType" attribute.

3.3.1 SampleSaml.java

The sample application code is contained in a single source file: **SampleSaml.java**. This file contains two classes:

- The **SampleSaml** Class
- The **MyCallBackHandler** Class

3.3.1.1 The SampleSaml Class

The `main` method of the SampleSaml class performs the authentication and then reports whether or not the authentication succeeded.

The code for authenticating the user is very simple, consisting of just two steps:

1. Instantiate a LoginContext.
2. Call the LoginContext's login method.

Instantiating a LoginContext

In order to authenticate a user, you first need a `javax.security.auth.login.LoginContext`.
Here is the basic way to instantiate a LoginContext:

```
import javax.security.auth.login.*;
. . .
LoginContext lc =
    new LoginContext(<config file entry name>,
           <CallbackHandler to be used for user
interaction>);
```

and here is the specific way our tutorial code does the instantiation:

```
import javax.security.auth.login.*;
. . .
LoginContext lc =
    new LoginContext("SamlEntry",
          new MyCallbackHandler());
```

The arguments are the following:

- **The name of an entry in the JAAS login configuration file**

  This is the name for the LoginContext to use to look up an entry for this application in the JAAS login configuration file.

  The entry in the login configuration file we use for this example (see jaas.config section 2.3.3.4) has the name "SamlEntry", so that is the name we specify as the first argument to the LoginContext constructor.

- **A CallbackHandler instance.**

  When a LoginModule needs to communicate with the user, for example to ask for a user name and password, it does not do so directly. That is because there are various ways of communicating with a user, and it is desirable for LoginModules to remain independent of the different types of user interaction. Rather, the LoginModule invokes a `javax.security.auth.callback.CallbackHandler` to perform the user interaction and obtain the requested information, such as the user name and password.

  MyCallbackHandler is an instance of CallbackHandler and used as the second argument to the LoginContext constructor. The LoginContext forwards that instance to the SamlLoginModule.

Calling the LoginContext's `login` Method

Once we have a LoginContext `lc`, we can call its `login` method to carry out the authentication process:

lc.login();

The LoginContext instantiates a new empty **javax.security.auth.Subject** object (which represents the user or service being authenticated). The LoginContext constructs the configured LoginModule (in our case SamlLoginModule) and initializes it with this new Subject and MyCallbackHandler.

The LoginContext's `login` method then calls methods in the SamlLoginModule to perform the login and authentication. The SamlLoginModule will utilize the MyCallbackHandler to obtain the user name and password. Then the

SamlLoginModule will request Evidian Authentication Server to check that the name and password are the ones it expects.

If authentication is successful, the SamlLoginModule populates the Subject with a Principal representing the user. The calling application can subsequently retrieve the authenticated Subject by calling the LoginContext's `getSubject` method.

The Complete `Sample` Class Code

Now that you have seen the basic code required to authenticate the user, we can put it all together into the full class in SampleSaml.java, which includes relevant import statements and error handling:

```
package sample;

import javax.security.auth.login.*;
// . . . other import statements needed by MyCallbackHandler . . .

/**
 * This Sample application attempts to authenticate a user
 * and reports whether or not the authentication was
 * successful.
 */
public class SampleSaml {
      private static final Class CREDENTIAL_CLASS =
      com.evidian.security.auth.saml.SamlAssertion.class;
    /**
    * Attempt to authenticate the user.
    *
    * @param args input arguments for this application.
    * These are ignored.
    */
    public static void main(String[] args) {

      // Obtain a LoginContext, needed for authentication.
      // Tell it to use the LoginModule implementation
      // specified by the entry named "Sample" in the
      // JAAS login configuration file and to also use the
      // specified CallbackHandler.
      LoginContext lc = null;
      try {
          lc = new LoginContext("SamlEntry",
                                new MyCallbackHandler());
      } catch (LoginException le) {
          System.err.println("Cannot create LoginContext. "
              + le.getMessage());
          System.exit(-1);
      } catch (SecurityException se) {
          System.err.println("Cannot create LoginContext. "
              + se.getMessage());
          System.exit(-1);
      }
```

```
// the user has 3 attempts to authenticate successfully
int i;
for (i = 0; i < 3; i++) {
    try {

        // attempt authentication
        lc.login();

        // if we return with no exception,
        // authentication succeeded
        break;

    } catch (LoginException le) {

        System.err.println("Authentication failed:");
        System.err.println("  " + le.getMessage());
        try {
            Thread.currentThread().sleep(3000);
        } catch (Exception e) {
            // ignore
        }

    }
}

// did they fail three times?
if (i == 3) {
    System.out.println("Sorry");
    System.exit(-1);
}

System.out.println("Authentication succeeded!");
 Subject mySubject = lc.getSubject();
 // let's see what Principals we have Iterator
 principalIterator = mySubject.getPrincipals().iterator();
 System.out.println("Authenticated user has the following
 Principals:");
 while (principalIterator.hasNext()) {
   Principal p = (Principal)principalIterator.next();
   System.out.println("\t" + p.toString());
}
 System.out.println("User has " +
 mySubject.getPublicCredentials().size() + " Public
 Credential(s)");
   // Now let's see the expiration date
 Object[] objs =
 mySubject.getPublicCredentials(CREDENTIAL_CLASS).toArray();
  SamlAssertion token = (SamlAssertion)objs[0];
         byte[] assertion = token.getCredential();
 System.out.println("Credentials expire date:" +
 token.getExpiryDate());
   // Finally display all the Saml assertion
   System.out.println("\n\nAssertion dump:\n" +
   token.toString());
   try {
```

```
                lc.logout();
            } catch (Exception e) {
                // Ignore
            }
            System.exit(0);
        }
    }
```

3.3.1.2 The MyCallbackHandler Class

An application can either use one of the sample implementations provided in the `com.sun.security.auth.callback` package or, more typically, write a CallbackHandler implementation. The sample code supplies its own CallbackHandler implementation, the MyCallbackHandler class in Sample.java.

CallbackHandler is an interface with one method to implement:

```
     void handle(Callback[] callbacks)
         throws java.io.IOException,
UnsupportedCallbackException;
```

The LoginModule passes the CallbackHandler handle method an array of appropriate javax.security.auth.callback.Callbacks, for example a NameCallback for the user name and a PasswordCallback for the password, and the CallbackHandler performs the requested user interaction and sets appropriate values in the Callbacks.

The MyCallbackHandler `handle` method is structured as follows:

```
public void handle(Callback[] callbacks)
  throws IOException, UnsupportedCallbackException {

  for (int i = 0; i < callbacks.length; i++) {
    if (callbacks[i] instanceof TextOutputCallback) {

      // display a message according to a specified type
      . . .

    } else if (callbacks[i] instanceof NameCallback) {

      // prompt the user for a username
      . . .

    } else if (callbacks[i] instanceof PasswordCallback) {

      // prompt the user for a password
      . . .

    } else {
        throw new UnsupportedCallbackException
         (callbacks[i], "Unrecognized Callback");
    }
  }
}
```

A CallbackHandler `handle` method is passed an array of Callback instances, each of a particular type (NameCallback, PasswordCallback, etc.). It must handle each Callback, performing user interaction in a way that is appropriate for the executing application.

MyCallbackHandler handles three types of Callbacks: **NameCallback** to prompt the user for a user name, **PasswordCallback** to prompt for a password, and **TextOutputCallback** to report any error, warning, or other messages the SamlLoginModule wishes to send to the user.

3.3.2 SampleIdPrincipal.java

SampleIdPrincipal.java is a sample class implementing the java.security.Principal and java.io.Serializable interfaces. If authentication is successful, the SamlLoginModule populates a Subject with a SampleIdPrincipal representing the 'uid' attribute user.

3.3.3 SampleProfilePrincipal.java

SampleIdPrincipal.java is a sample class implementing the java.security.Principal and java.io.Serializable interfaces. If authentication is successful, the SamlLoginModule populates a Subject with a SampleIdPrincipal representing the 'employeeType' attribute user.

3.4 Running the Code

1. Under APP_HOME, compile **SampleSaml.java**, **SampleIdPrincipal.java** and **SampleProfilePrincipal.java**:

   ```
   javac sample/SampleSaml.java
   sample/principal/SampleIdPrincipal.java
   sample/principal/SampleProfilePrincipal.java
   ```

2. Execute the `SampleSaml` application :

   ```
   java -Djava.security.auth.login.config=config/jaas.config
   -classpath "%CLASSPATH%" sample.SampleSaml
   ```

   > You will be prompted for your user name and password, and the SamlLoginModule specified in the login configuration file will check to ensure these are correct.

For our configuration, the SamlLoginModule expects a login and password for a user stored in Built-in User's directory.

You will see some messages output by SamlLoginModule as a result of the `debug` option being set to `true` in the login configuration file. Then, if your login is successful, you will see the following message output by SampleSaml like this:

```
…
Authentication succeeded!
Authenticated user has the following Principals:
        SampleProfilePrincipal:  Countable head
        SampleIdPrincipal:  hall
User has 1 Public Credential(s)
Credentials expire date:Fri Jan 06 19:02:59 CET 2006
…
```

If the login is not successful, you will see `Authentication failed`.

4. Developing your Own Principal and Credential Classes

This section describes how your own Principal and Credential classes can be handled by SAML Login Module.

4.1 Design

As explained in introduction, SAML Login Module is independent of the classes used to populate the Subject at compile time. To use your own classes first try to reuse Evidian factory. When it is not possible, implement your own version of these abstract factories to handle your principal and credential classes.

4.2 Using Evidian Principal Factory

Your implementation of a JAAS principal can be handled by Evidian principal factory at the condition that it has a public String argument constructor.

Evidian principal factory invokes this constructor to instantiate your principal with the value of a SAML Attribute in the **login** method.

Caution

If your class does not have this constructor, an error is detected at runtime and a SamlLoginException is thrown with the following error code: PRINCIPAL_METHOD_NOT_FOUND.

4.2.1 Implement the SecurityContext Interface for Principal Authentication

When your principal needs to be used in a principal authentication, you have to implement the **SecurityContext** interface.

If your class does not implement this interface, SAML Login Module cannot retrieve the assertion and raises an exception with the following error code:`SAML_REQUEST_MISSING_TOKEN`.

You have to implement two methods:

- `getAssertion`
- `setAssertion`

```
byte[] getAssertion()
```

Evidian principal factory uses this marker to retrieve a principal from the subject and then invokes this method to get the SAML assertion used for authentication.

```
void setAssertion(byte[] assertion)
```

Evidian principal factory invokes this method to set SAML assertion in your principal in `login` method.

4.2.2 Override "equals" and "hashCode" in Your Principal

SAML Login Module performs the following operations in your principal:

- Adds individually into subject in `login`
- Adds collectively into subject in `commit`
- Removes collectively from subject in `abort` and `logout`

All these operations on collection involve the **equals** and **hashCode** methods. By default these methods inherited from **Object** base equality on the object's identifier.

But with this implementation, two instances returning a same name are not equal, because these are two different instances. This would cause duplicated principals in subject. That is why you surely have to override **equals** and **hashCode** in your principal to base equality on the name's value.

4.2.3 Clone the Returned Object of the "getAssertion" in Your Principal

When you implement the **SecurityContext**, return a copy of the assertion and not a reference to the field, otherwise it authorizes modification of security related data to caller.

4.3 Implement your Principal Factory

You have to extend **AbstractPrincipalFactory**, which means to create a public no argument constructor and implement its two abstract methods:

- `constructor`
- `getAssertion`
- `getPrincipal`

The **getInstance** method of the **AbstractPrincipalFactory** is a singleton method that instantiates the implementation of Principal Factory. By this way, the time consuming operation of initialization (like reflecting classes from their names) can be done once for all and optimize speed in next two services methods. As **AbstractPrincipalFactory** instantiates using the **newInstance** method, this implies that your factory cannot have a restricted visibility or non empty argument constructor.

```
public byte[] getAssertion(Subject subject);
```

The **getAssertion** method is called to authenticate a principal. It retrieves from its Subject argument the principal that contains SAML assertion and returnes it as a byte array.

```
public Set getPrincipals(SecurityModuleInterface
securityModule)
```

The **getPrincipals** method is called to populate Subject when login succeeds. It retrieves from **SecurityModuleInterface** values to pass to the Principal constructors via a mapping between SAML attributes and JAAS Principals.

4.4 Declare your Principal Factory

Declare your Principal Factory using the following system property: `com.evidian.factory.principal`.

EXAMPLE:

To declare the Principal Factory **com.acme.auth.PrincipalFactoryImpl**:

```
-Dcom.evidian.factory.principal=
com.acme.auth.PrincipalFactoryImpl
```

4.5 Using Evidian Credential Factory

To be handled by Evidian Credential Factory, your Credential class must have a public String and Date argument constructor.

Evidian Credential Factory invokes this constructor to instantiate your credential with the SAML assertion and its expiration date.

If your class does not have this constructor, an error is detected at runtime and a **SamlLoginException** is thrown with the following error code: `CREDENTIAL_METHOD_NOT_FOUND`.

As for the Principal implementation, your credential has:

- To override **equals** and **hashCode**, see Section 4.2.2, "Override "equals" and "hashCode" in Your Principal" for details
- To return a copy, not a reference, of SAML Assertion in its assessor. See Section 4.2.3, "Clone the Returned Object of the "getAssertion" in Your Principal" for details. As the expiration field is probably of type **Date**, which is immutable, it is not concerned.

4.5.1 Implement the Interface "Destroyable"

With the **logout** method, SAML Login Module removes the Principal and the Credential from the Subject. When the Subject is in read only state, a JAAS module is expected to invoke the **destroy** method of the credential.

Your credential must implement the two following methods:

- `destroy`
- `isDestroyed`

In our context where we authenticate using the SAML assertion authentication method (see Section 1.2, "Authentication Methods" in Chapter 1), destroy means "delete the assertion field of your credential".

4.6 Implement your Credential Factory

You have to extend **AbstractCredentialFactory**, which means to create a public no argument constructor and implement its abstract method:

- `constructor`
- `getCredential`

For more details on the discussion on public no argument constructor, see Section 4.3, "Implement your Principal Factory".

```
public Object getCredential(SecurityModuleInterface
securityModule);
```

The **getCredential** method is called to populate the Subject when login succeeds. It retrieves from the **SecurityModuleInterface** values to pass to the Credential constructor via a mapping between SAML assertion and JAAS credential.

4.7 Declare your Credential Factory

Declare your Credential Factory using the following system property: `com.evidian.factory.credential`.

EXAMPLE:

To declare the principal factory **com.acme.auth.CredentialFactoryImpl**:

```
-Dcom.evidian.factory.credential=com.acme.auth.CredentialFactoryImpl
```

5. SAML Web Services - Resources Location

SAML Authentication and **Certificate to SAML** Web Services are extensions of AccessMaster SAM J2EE that allow you to add a Web Service access to the Authentication server (SAML provider):

- The **SAML Authentication** Web Service is aimed at getting the SAML assertion from a user after having successfully authenticated him.
- The **Certificate to SAML** Web Service is aimed at getting a user SAML assertion from its X509 certificate.

This section describes the location of the configuration resources and the data resources related to the SAML Authentication and the Certificate to SAML Web Services, and gives you use cases.

Note

By resources, we mean the **.properties** configuration files as well as directories of data files.

Please take into account that by default, the Authentication Web Service and the Certificate to SAML Web Service are activated when the Authentication server starts and are available only with an Authentication server configured to generate SAML assertions.

5.1 Location of Configuration Resources

This section describes the location of the configuration resources related to **SAML Authentication** and **Certificate to SAML** Web Services.

5.1.1 Location of SAML Authentication Configuration Resources

All the configuration resources concern the Apache server and are located in the **$LPF_ROOT_DIR/authsrv/server/config/pxphttpd.conf** file, as shown in the following extract:

```
# Load the SOAP toolkit module
<IfModule mod_saml.c>
LoadModule gsoap_module /usr/evidian/lib/mod_gsoap.so
</IfModule>

[…]
# Web Service that delivers a SAML assertion
#
<IfModule mod_gsoap.c>
<IfModule mod_saml.c>
      <FilesMatch "(WS-auth)$">
            Order deny,allow
            deny from all
            allow from env=lpf_allow_cond_for_auth_remote

            SetHandler gsoap-handler
            SOAPLibrary /usr/evidian/authsrv/lib/libWS-
auth.so
      </FilesMatch>
</IfModule>
```

In this extract, the first part loads the **gsoap** module and the second part activates the Web Service on the expected URL.

Note

The path of the libraries – in bold hereabove – has to be adapted to your configuration environment.

5.1.2 Location of Certificate to SAML Configuration Resources

All the configuration resources concern the Apache server.

- One part is located in the **$LPF_ROOT_DIR/authsrv/server/config/pxphttpd.conf** file, as shown in the following extract:

```
# Load the SOAP toolkit module
<IfModule mod_saml.c>
LoadModule gsoap_module /usr/evidian/lib/mod_gsoap.so
</IfModule>
```

This first part loads the **gsoap** module.

Note

The path of the library – in bold hereabove – has to be adapted to your configuration environment.

- Another part of the resources is located in the **$LPF_ROOT_DIR/authsrv/server/config/pxpvirtualhost.conf** file, as shown in the following extract:

```
<IfModule mod_gsoap.c>
<IfModule mod_saml.c>
<Directory "/usr/evidian/authsrv/cgi-bin">
<Files cert2saml>
      <IfModule mod_lpf_authform.c>
            LpfAuthTimeout 600
            LpfAuthCookieType Session
            LpfAuthCheckIP On
      </IfModule>

      # Make sure to have both host-level access control
and user authentication
      Satisfy All
      Deny from All
      Allow from env=lpf_allow_cond_for_remote

      <IfModule mod_ssl.c>
            # Type of Client Certificate verification
            SSLVerifyClient none
      </IfModule>

      <IfModule mod_auth_lpf.c>
            lpfAuthPolicyName
      "lpfauthenticationpolicy_LoginModule" 0
      </IfModule>
      AuthType Basic
      AuthName "SAM Web infrastructure domain"
      require valid-user

      # PXP:
      # PXP: This file has been instanciated with the
template file:
```

```
        # PXP:
/usr/evidian/authsrv/tmpl/http_basic_auth.conf.tmpl
        # PXP:

        <IfModule mod_auth_lpf.c>
              LpfAuthAuthoritative on
        </IfModule>

        # End of template file

        SetHandler gsoap-handler
        SOAPLibrary /usr/evidian/authsrv/lib/libWS-auth.so
</Files>
</Directory>
</IfModule>
</IfModule>
```

This second part activates the Web Service on the expected URL.

Note: The path of the library – in bold hereabove – is automatically adapted to your configuration environment when the Authentication server starts.

5.2 Data Resources Location

This section describes the location of the data resources related to **SAML Authentication** and **Certificate to SAML** Web Services.

5.2.1 Web Service Resources

- SAML Authentication Web Service is accessible at the following URL:

  < authentication server URL >/pxpadmin/bin/WS-auth
- Certificate to SAML Web Service is accessible at the following URL:

  < authentication server URL >/pxpadmin/bin/cert2saml

Both Web Services use the following libraries:

- **$LPF_ROOT_DIR/lib/mod_gsoap.so**.
- **$LPF_ROOT_DIR/authsrv/lib/libWS-auth.so**.

5.2.2 The WSDL File

The WSDL file is the description file of the **SAML Authentication** and the **Certificate to SAML** Web Services.

- For SAML Authentication Web Service, it is located in: **$LPF_ROOT_DIR/authsrv/wsdl/WS-auth.wsdl**.
- For Certificate to SAML Web Service, it is located in : **$LPF_ROOT_DIR/authsrv/wsdl/WS-cert2saml.wsdl**

In order to set the WSDL file public, you just need to copy it in a web accessible folder. For example, you can copy it in **$LPF_ROOT_DIR/authsrv/html/**.

That way, the WSDL file is available at the following URL:

- For SAML Authentication Web Service: **< authentication server URL >/pxpadmin/authsrv/WS-auth.wsdl**.
- For Certificate to SAML Web Service: **< authentication server URL >/pxpadmin/authsrv/WS-cert2saml.wsdl**

5.2.3 Default Web Service Clients

Evidian delivers two Web Service clients for Authentication and for Certificate to SAML Web Services. They are all located in **$LPF_ROOT_DIR/authsrv/bin**.

- To test the Web Service through HTTP:
  - For SAML Authentication Web Service: **WS-auth-client**
  - For Certificate to SAML Web Service: **WS-cert2saml-client**

  Note: Even if the Authentication and Certificate to SAML Web Services is supposed to run only through HTTPS, testing using HTTP is usefull to qualify a possible "Service versus Authentication server" problem.

- To test the Web Service through HTTPS (expected running):
  - For SAML Authentication Web Service: **WS-auth-sslclient**: t
  - For Certificate to SAML Web Service: **WS-cert2saml-sslclient**

5.3 Use Cases

5.3.1 SAML Authentication Web Service Use Cases

5.3.1.1 Standard Scenario

A user wants to reach a Service available as a Web Service. The access to this Service is controlled, and a SAML assertion is required in the SOAP request.

First, the consumer will authenticate the user sending its username and password in a SOAP request to the authentication server.

If successful it will get back the SAML assertion in the SOAP response.

Now the service request can be built using the received assertion.

On the server side, the server may use the JAAS API to verify the assertion.

5.3.1.2 Using the WS-auth-client Binary

The WS-auth-client binary sends the XML request matching the type described in **$LPF_ROOT_DIR/authsrv/wsdl/WS-auth.wsdl**.

The client calls the Service to get back the SAML assertion.

Usage:

Type the following command:

```
./WS-auth-client url version username password domain policy
```

where:

- url (mandatory): Authentication Web service URL.

  (example: **< authentication server URL>/pxpadmin/bin/WS-auth**)
- version (mandatory): protocol version (one value: 1.0).
- username (mandatory): user login.
- password (mandatory): user password.
- Domain (optional): specifies the user repository.

  Important

  This element is mandatory if there are several user repositories configured in the SAM J2EE base.

- policy (mandatory): name of the policy object in the SAM J2EE base. This policy object specifies options on the authentication, and contents of the issued SAML assertions.

The client in return treats the XML flow then displays both assertion and signature.

5.3.1.3 Using the WS-auth-sslclient Binary

The WS-auth-sslclient works in the same way as WS-auth-client (as described in Section 5.3.1.2 "Using the WS-auth-client Binary"), except that it requires an additional parameter: the path to a file containing the authentication server certificate in PEM format.

5.3.2 Certificate to SAML Web Service Use Cases

5.3.2.1 Standard Scenario

A user wants to reach a Service available as a Web Service. The access to this Service is controlled, and a SAML assertion is required in the SOAP request.

First, the consumer has to send the user certificate in a SOAP request to the authentication server. And to do so, the consumer must authenticate against the authentication server.

If successful it gets back the SAML assertion in the SOAP response.

Now the service request can be built using the received assertion.

On the server side, the server may use the JAAS API to verify the assertion.

5.3.2.2 Using the WS-cert2saml-client Binary

The WS-cert2saml-client binary sends the XML request matching the type described in **$LPF_ROOT_DIR/authsrv/wsdl/WS-cert2saml.wsdl**.

The client calls the Service to get back the SAML assertion.

Usage:

Type the following command:

```
./WS-cert2saml-client url app_name app_pass version
user_certificate policy checkcert OCSP_response
```

where:

- url (mandatory): Authentication Web service URL.

(example: **< authentication server URL>/pxpadmin/bin/cert2saml**)

- app_name (mandatory): consumer login.
- app_pass (mandatory): consumer password.
- version (mandatory): protocol version (one value: 1.0).
- user_certificate (mandatory): user certificate in pem format
- policy (NOT mandatory): name of the policy object in the SAM J2EE base. This policy object specifies options on the authentication, and contents of the issued SAML assertions.
- checkcert (mandatory): tells the web service the type of certificate verification to perform; either "NoCheck", "CrlOcspVerification" or "CheckOcspResponse".
- OCSP_response (NOT mandatory): the **base64** encoded OCSP response, ie the HTTP body of the response to the OCSP request emitted by the application, encoded in base64 format.

The client in return treats the XML flow then displays both assertion and signature.

5.3.2.3 Using the WS-cert2saml-sslclient Binary

The WS-cert2saml-sslclient works in the same way as WS-cert2saml-client (as described in Section 5.3.2.2, "Using the WS-cert2saml-client Binary"), except that it requires an additional parameter: the path to a file containing the authentication server certificate in PEM format.

6. WebLogic Platform

This chapter describes platform specific considerations to take into account to use SAML Login Module under WebLogic.

- Section 6.1, "Populating the Subject" describes how does SAML Login Module populate subject for WebLogic.
- Section 6.2, "Using SAML Login Module Under WebLogic" describes how to use a custom login module under WebLogic.
- Section 6.3, "JAAS Configuration File in RMI Client" describes JAAS configuration in RMI client connected to WebLogic server.

WebLogic version used is 8.1 SP2. There is a known limitation with the first release of version 8.1 about role support, so prefer using the service pack release.

6.1 Populating the Subject

This section describes how SAML Login Module fills in the Subject for the expected Principal (i.e. the one designed to authenticate with the Principal method) to be returned in EJB under WebLogic.

6.1.1 WebLogic Principal Selection Mechanism

EJB invokes the platform implementation of **getCallerPrincipal** to retrieve one specific Principal. When there are many Principals in the Subject, the WebLogic implementation of the **getCallerPrincipal** method selects in first the Principal that implements the **WLUser** interface.

WebLogic gives an implementation named **WLSUserImpl** of this interface. You can use it in Principal Factory as for the implementation named **WLSGroupImpl**. But if you need to authenticate with the Authenticate Principal method in your EJB, you need to develop your own implementation.

Before describing an example of such a principal we need to study validator.

6.1.2 Principal Validator

In the WebLogic architecture, the Principal added to the Subject needs to be signed to prevent tampering. This is the job of the Principal Validator. WebLogic gives an implementation of a validator named **PrincipalValidatorImpl**. It signs and validates a Principal that extends **WLSAbstractPrincipal**.

6.1.3 Principal Conception

A sample principal class designed to handle principal authentication in WebLogic:

- Extends `WLSAbstractPrincipal` so that you don't need to develop your own validator.
- Implements `WLSUser` interface so that WebLogic returns an instance of this class in `getCallerPrincipal`.
- Implements your(s) interface(s) if you need to retrieve this instance from subject after explicit invocation to SAML Login Module.
- Implements Evidian interface when you rely on Evidian principal factory implementation to populate subject.

6.2 Using SAML Login Module Under WebLogic

This section describes what you have to do in WebLogic to use SAML Login Module.

SAML Login Module is declared into the WebLogic framework by the mean of an authentication provider composed of:

- An MBean definition file.
- An authenticator provider implementation.

6.2.1 MBean Definition File For SAML Login Module

The management of custom login modules under WebLogic relies on MBean. WebLogic MBean maker generates MBean based on definition of MBean Definition File. To write this file follow the instructions indicated in the WebLogic documentation (http://edocs.bea.com/wls/docs81/dvspisec/atn.html#1116337). First copy the authentication sample of WebLogic, then replace the sample values with values for custom login module. The following is the list of the values used in the Evidian sample.

In the `MBeanType` element:

- `Name = "SamlAuthenticator"`
- `DisplayName = "SamlAuthenticator"`
- `Package = "com.evidian.security.auth.wls"`
- Other attributes unchanged.

In the `MBeanAttribute` element:

- `Default = ""com.evidian.security.auth.wls.SamlAuthenticationProviderImpl""`
- Other attributes unchanged.

6.2.2 Authentication Provider Implementation for SAML Login Module

In WebLogic, custom login modules are wrapped with a specific authentication provider. An Evidian sample implementation is given based on the WebLogic sample with the following modifications:

- The method **initialize** references Evidian **MBean** in place of the WebLogic one. Authentication provider queries MBean for modules properties.
- The method **getLoginModuleConfiguration** adds to SAML Login module the **request** specific option, -usually defined in the JAAS configuration file, with the value **id-password**. For development phase add the **debug** option with the value **true**.
- The method **getConfiguration** references the **com.evidian.security.auth.login.SamlLoginModule** SAML Login Module class in place of the WebLogic one.

If you don't want to extend **WLSAbstractPrincipal**, it implies that you develop your own Principal Validator provider. In such a case you have to reference this implementation in place of **PrincipalValidatorImpl** in the **getPrincipalValidator** method.

6.2.3 Login Module Scope under WebLogic

The scope of Security Providers (including Authentication Providers) is the Security Realms. Multiple realms can be declared but only one can be set as active. Each application deployed under a server uses the same login modules. The View Authentication Providers select in the Security Realm as login modules stack in the JAAS configuration file.

6.2.4 Class Loader

This section describes how to find SAML Login Module classes and resources under WebLogic.

As WebLogic loads JAAS module by the mean of an authentication provider, the first idea, suggested in the WebLogic sample is to package the SAML Login Module JAR file into your MBean JAR file (see Section 6.2.2, "Authentication Provider"). But this solution raises a "class not found exception" if you explicitly authenticate in EJB (precisely when it retrieves principal) because class loader inspection reveals that MBean class loader *is not a parent* of application class loader (see "Explicit Athentication in an EJB" in "Introduction").

As the first common ancestor of MBean class loader and application class loader is the system class loader, we recommend you to:

- Package the SAML Login Module JAR file with your Principal and Credential classes in one JAR
- Add this JAR to the WebLogic classpath by modifying its definition in the WebLogic startup script

6.3 JAAS Configuration File in RMI Client

This section describes how to configure JAAS in RMI client of EJB deployed into WebLogic.

This configuration is done on the client side whereas the authentication occurs on the server side. The client does not indicate SAML Login Module but **weblogic.security.auth.login.UsernamePasswordLoginModule** (WebLogic module). Then the Weblogic module uses the declared SAML Login Module to authenticate the user.

Example from the WebLogic documentation
http://edocs.bea.com/wls/docs81/security/fat_client.html#1046438

```
Sample {
   weblogic.security.auth.login.UsernamePasswordLoginModule
   required debug=false
};
```

If you indicate SAML Login Module, you authenticate on the client side. When you retrieve the Subject and try to execute an action as is, WebLogic will reject your Principal with a **SecurityException** because it does not validate them.

7. JBoss Platform

This section describes platform specific considerations you have to know to use SAML Login Module under JBoss.

- Section 7.1, "Populating the Subject" describes how does SAML Login Module populate subject for JBoss.
- Section 7.2, "Using SAML Login Module under JBoss" describes how to use a custom login module under JBoss.
- Section 7.3, "JAAS Configuration File in RMI Client" describes JAAS configuration in RMI client connected to JBoss server.

Note

JBoss 3.2.3 is used.

7.1 Populating the Subject

This section describes how SAML Login Module fills in the Subject for the expected Principal (that is the one designed to authenticate with the Principal method) to be returned in EJB under JBoss.

7.1.1 JBoss Principal Selection Mechanism

EJB invokes the platform implementation of **getCallerPrincipal** to retrieve one specific Principal. JBoss selects in the set of the Principals populated by the JAAS module the one belonging to the group (class `org.jboss.security.SimpleGroup`) named `CallerPrincipal` when it exists.

For SAML Login Module it implies a specific treatment of the Principal designed for the method "authenticate with principal" in a JBoss context (see Section 1.4.1, "Explicit Authentication in an EJB").

7.1.2 Specific Treatment for JBoss in SAML Login Module

The Evidian Principal Factory implementation populates the Subject with Principals. When it handles a Principal that implements the **SecurityContext** interface in a JBoss context it creates **org.jboss.security.SimpleGroup** (Jboss) named **CallerPrincipal** and puts the Principal into.

7.1.3 JBoss Context Detection in SAML Login Module

As there is a specific treatment for JBoss, SAML Login Module has to know when to perform it. To detect a JBoss context, Evidian Principal Factory loads at initialization the **org.jboss.security.SimpleGroup** class. A success means the JBoss context is detected whereas a "class not found exception" means "not in JBoss context".

With such a way to load classes needed for JBoss, SAML Login Module does not have dependency with JBoss JAR file in general case.

7.1.4 Conception of Principal

JBoss has no requirement on class for Principal, unlike WebLogic with its **WLSUser** interface. If you decide to use Evidian Principal Factory your Principal must implement the **SecurityContext** interface; if you have no additional requirement, use the Evidian implementation of **SecurityContext** otherwise develop your own implementation. Then declare the selected Principal class in **principal.properties**.

7.2 Using SAML Login Module under JBoss

This section describes what you have to do in JBoss to use SAML Login Module.

7.2.1 JAAS Configuration of JBoss

Under Jboss, all the login modules are declared in the XML configuration file of "XML JAAS Login Configuration Mbean". This file is named **login-config.xml** and is located into the **server/default/conf** directory. Using **XMLLoginConfig Mbean**, you can declare several stacks of login modules. The selection of login modules is made on the application side with JBoss specific descriptors. (see Section 7.2.2, "Select SAML Login Module for Your Application"). The following is an example of a configuration entry for SAML Login Module:

```
<application-policy name="auth-id-password">
   <authentication>
      <login-module code=
"com.evidian.security.auth.login.SamlLoginModule"
            flag="required">
         <module-option name="request">id-password</module-option>
         <module-option name="debug">true</module-option>
      </login-module>
   </authentication>
</application-policy>
```

SAML Login Module class is declared with the **code** attribute of the **login-module** element., Its specific options are declared using the **module-option** elements. You recognize the options **debug** and **request** described in Section 3.2, "JAAS Configuration File".

At this step SAML Login Module is available under a JBoss configuration.

7.2.2 Select SAML Login Module for Your Application

Each application deployed into JBoss selects its JAAS configuration entry. This selection is done into the JBoss specific deployments descriptors **jboss.xml** and **jboss-web.xml** used respectively for EJB and Servlet.

To activate authentication for an EJB, create a **security-domain** element into **jboss.xml** or **jboss-web.xml**.For its value concatenate constant `java:/jaas` with name attribute of `application-policy` element create before. In our case it gives:

```
<security-domain>java:/jaas/auth-id-password</security-domain>
```

At this step SAML Login Module is selected to authenticate user of deployed application.

7.2.3 Class Loader

This section describes how to find SAML Login Module classes and resources under JBoss.

To find SAML Login Module classes put its JAR file into the **lib** directory under **server/default**. When it starts JBoss loads every JAR from this directory.

Important

There is another **lib** directory at the root of JBoss install. Do not use this directory for SAML Login Module otherwise you will get a "class not found exception".

SAML Login Module loads its resources with the classloader, so it is needed to add the directory of **principal.properties** and **credential.properties** to JBoss CLASSPATH. To do so edit the JBoss startup script named **run.bat** (**run.sh** under UNIX) located into the **bin** directory. Add an entry to the `JBOSS_CLASSPATH` variable.

The follwing is an example under Windows for JBoss installed into **c:\java\jboss3.2.3**:

```
set JBOSS_CLASSPATH=%JBOSS_CLASSPATH%;%JBOSS_HOME%\my_classes
```

where **principal.properties** and **credential.properties** are located into **c:\java\jboss3.2.3\my_classes**.

This is required if you are using the Evidian Principal Factory implementation.

7.3 JAAS Configuration File in RMI Client

This section describes how to configure JAAS in the RMI client of EJB deployed into JBoss.

The client does not indicate SAML Login Module but **org.jboss.security.ClientLoginModule** (Jboss). This module does not authenticate user, but after retrieving login and password from callback handler links these two parameters to the EJB invocation. The Authentication occurs on server side.

```
Sample {
    org.jboss.security.ClientLoginModule  required;
};
```

Important

If you indicate SAML Login Module you authenticate on client side. When you retrieve the Subject and try to execute an action as is, JBoss will reject your Principal.

8. WebSphere Platform

This chapter describes platform specific considerations to take into account to use SAML Login Module under WebSphere.

- Section 8.1, "WebSphere Version" lists the versions of WebSphere taken into account by this chapter.
- Section 8.2, "Authentication in WebSphere using JAAS" describes WebSphere authentication mechanisms.
- Section 8.3, "Using SAML Login Module under WebSphere" describes how to use a custom login module under WebSphere.
- Section 8.4, "JAAS Configuration File in RMI Client" describes JAAS configuration in RMI client connected to WebSphere server.

8.1 WebSphere Version

The information in this chapter is based on WebSphere Application Server V5.1.1.5. The features described in this chapter first appeared in WebSphere Application Server V5.1.1 on the distributed platform, and are also available for WebSphere Application Server V5.1 on z/OS® as part of an update. In the preceding version (V5.0 or earlier), IBM WebSphere Application Server had a rigid authentication model and not customizable.

8.2 Authentication in WebSphere using JAAS

The authentication information can be one of the following:

- Basic authentication (user ID and password).
- Credential token (in case of Lightweight Third Party Authentication (LTPA)).
- Client certificate.

The Web authentication is performed by the Web Authentication module. The enterprise bean authentication is performed by the Enterprise JavaBean (EJB) authentication module.

The authentication module is implemented using the JAAS login module. The Web authenticator and the EJB authenticator pass the authentication data to the login module, which can use any of the following mechanisms to authenticate the data:

- Simple WebSphere Authentication Mechanism (SWAM)
- Lightweight Third Party Authentication (LTPA)

8.2.1 Simple WebSphere Authentication Mechanism

SWAM is intended for simple, non-distributed, single application server run-time environments. The single application server restriction is due to the fact that SWAM does not support forwardable credentials.

8.2.2 Lightweight Third Party Authentication

LTPA is intended for distributed, multiple application server and machine environments. It supports forwardable credentials and single signon (SSO). LTPA can support security in a distributed environment through cryptography. This supports permits LTPA to encrypt, digitally sign, and securely transmit authentication-related data, and later decrypt and verify the signature.

For more details on how configuring WebSphere authentication mechanisms, see WebSphere software information center.

8.2.3 JAAS Login Module in WebSphere

The WebSphere Application Server authentication process is fully pluggable. By providing plug points for custom code at most key steps, it is possible to heavily customize the WebSphere Application Server authentication process.
Code can be developed to add custom information to a subject, require additional authentication information as part of a login process, or even bypass the normal registry usage by asserting complete user credentials to WebSphere Application Server.

Most of the authentication process is built around JAAS login modules, and so it is possible to plug in custom login modules before, after, or between the login modules provided by IBM (however, the IBM modules must not be removed).

WebSphere Application Server provides a set of standard login configurations with login modules and callbacks that are used in various situations to achieve authentication. These modules and callback handlers are defined and available depending on the specific authentication situation.

8.2.3.1 Application Logins

Application logins are the ones that your enterprise applications can use.

8.2.3.2 System Logins

System login definitions are related to the application server itself, not the applications. These login modules are used by SWAM and LTPA mechanisms to authenticate data.

WebSphere Application Server Version 5.1.1.5 and later versions defines four system login configurations that are used in specific situations related to security:

- SWAM
- WEB_INBOUND
- RMI_INBOUND
- DEFAULT.

SWAM

Processes login requests in a single server environment when (SWAM) is used as the authentication method.

WEB_INBOUND

The WEB_INBOUND login configuration is used by LTPA mechanism. It handles logins for Web application requests, which includes servlets and JavaServer Pages (JSP) files. This configuration is intended to be used to authenticate Web-based (HTTP) traffic.

RMI_INBOUND

The RMI_INBOUND login configuration is used by LTPA mechanism. It handles logins for inbound RMI/IIOP requests. Typically, these logins are requests for authenticated access to Enterprise JavaBeans (EJB) files. Also, these logins might be Java Management Extensions (JMX) requests when using the RMI connector.

DEFAULT

The DEFAULT login configuration is used by both SWAM and LTPA mechanism; It handles the logins for inbound requests made by most of the other protocols and internal authentications.

Other configurations

There are other defined login configurations in WebSphere Application Server. The DEFAULT login configuration handles situations when none of the above configurations apply; these include SOAP requests from the admin client and JMX admin authentication requests. There might also be legacy login configurations, such as LTPA and LTPA_WEB. However, these should no longer be used as of WebSphere Application Server V5.1.1.

Additionally, there are two login configurations related to Web services, wssecurity.signature and wssecurity.IDAssertion

8.3 Using SAML Login Module under WebSphere

As mentioned in Section 8.2.3, "JAAS Login Module in WebSphere", the IBM modules must not be removed. For SAML Login Module to bypass the normal registry usage and assert user identity information to WebSphere Application Server, it must execute prior to IBM swamLoginModule and lptLoginModule. IBM Login Modules create user Principals and Credentials which are used in security attribute propagation from one WebSphere server to another.

8.3.1 Conception of Principal

SAML Login Module must identify and give the security name and the realm qualified groups to which the authenticated user belongs to IBM Login Modules. WebSphere Application Server uses these information for the getRemoteUser(), getUserPrincipal() and getCallerPrincipal() APIs.

The Principal class for the caller identity has to implement com.evidian.security.auth.interfaces.WASUserIDInterface interface.

If your class does not implement this interface, SAML Login Module cannot retrieve the caller identity and raises an exception with the following error code: WAS_PRINCIPAL_CONFIGURATION_ERROR.

Also if in your configuration users are member of groups, the Principal classes for user's groups have to implement com.evidian.security.auth.interfaces.WASGroupInterface interface.

For developping the other Principal and Credential Classes, refer to Chapter 4, "Developing your Own Principal and Credential Classes"

8.3.2 Conception of Subject

Once authenticated, a Subject is generated and contains principals of the J2EE caller and the J2EE caller credentials. It composed by:

- Principals and Credentials generated by Evidian or your Principal and Credential factories (For more details on Principal and Credential Factories see Chapter 4, "Developing your Own Principal and Credential Classes")
- Principals and Credentials generated by IBM Login Module(s).

The generated Subject can be returned with a static method **com.ibm.websphere.security.auth.WSSubject.getCallerSubject()**. This method is protected by Java 2 Security. If Java 2 Security is enabled, then access will be denied if the application code is not granted the permission javax.security.auth.AuthPermission("wssecurity.getCallerSubject").

8.3.3 Declare SAML Login Module Under WebSphere

JAAS login modules for WebSphere Application Server can be configured using the Administrative Console.

- In version 5.1.1, you can configure them under the following link: **Security -> JAAS Configuration**.
- In version 6, you can configure them under **Security -> Global Security ->Authentication -> JAAS Configuration**.

8.3.3.1 Login Configuration

When SAML Login Module is used by applications, you have to create a new Login Configuration with SAML Login Module as entry.

EXAMPLE

The following is the list of the values used to create a new Login Configuration **myLoginConfiguration** with the authentication method **auth-principal**:

- `alias = "myLoginConfiguration"`

In the General Properties element:

- `Module class name = "com.evidian.security.auth.login.SamlLoginModule"`
- `Proxy Classname = "com.ibm.ws.security.common.auth.module.proxy.WSLoginModule Proxy" (WebSphere V5)`

  `Under WebSphere 6, select the check box “login module proxy”.`
- `Authentication strategy = "REQUIRED"`

In the Additional Properties element:

- `request = "auth-principal"`

8.3.3.2 System Configuration

To assert the user information to WebSphere Application Server before it creates the subject on its own, you will insert SAML Login Module **before** the WebSphere Application Server modules on the IBM system login module stack with the authentication strategy **REQUIRED**.

You have to modify only system login modules which used by authentication mechanism.

EXAMPLE 1

- LTPA is the active authentication mechanism for WebSphere Application Server.
- An ejb application with protected methods is deployed in the server.
- EJB client provides a user name and password for login; connects to the server, looks up the ejb and then call it.

In this case, you will insert SAML Login Module before **com.ibm.ws.security.server.lm.ltpLoginModule** and **com.ibm.ws.security.server.lm..wsMapDefaultInboundLoginModule** modules on the RMI_INBOUD JAAS login module stack.

The following is the list of the values used:

In the General Properties element:

- `Module class name = "com.evidian.security.auth.login.SamlLoginModule"`
- `Proxy Classname = "com.ibm.ws.security.common.auth.module.proxy.WSLoginModule Proxy" (WebSphere V5)`

  `Under WebSphere 6, select the check box “login module proxy”.`
- `Authentication strategy = "REQUIRED"`

In the Additional Properties element:

- `request = "id-password"`
- `debug = "false"`

EXAMPLE 2

SWAM is the active authentication mechanism for WebSphere Application Server.

For all HTTP and RMI/IIOP requests, you just have to insert SAML Login Module before **com.ibm.ws.security.server.lm.swamLoginModule** module on the SWAM JAAS login module stack.

When you insert SAML Login Module on a system login module, the default is to add it at the end of the stack. You will need to use the Set Order button to fix this.

8.3.4 Class Loader

This section describes how to find SAML Login Module classes and resources under WebSphere.

JAAS is part of the JRE and by default can only see the JVM lib and ext classpath, IBM provides a proxy login module **com.ibm.ws.security.common.auth.module.proxy.WSLoginModuleProxy** that uses thread-based classloaders, enabling you to place the SAML Login Module files in the usual places, such as **WAS-INSTALL/lib/ext** or **WAS-INSTALL/classes** for **samllogin.jar** and **WAS-INSTALL/propteries** for **principal.properties** and **credential.properties.**

Important

When you configure SAML Login Module on application or system modules, the login module proxy discussed above has been selected.

8.3.5 WebSphere Security Configuration

Assuming that you have enabled Global Security for your server:

- Some of the JAAS APIs are protected by Java 2 Security permissions. When these APIs are used by application code and the **Enforce Java 2 security** option is enabled, make sure that your application doesn't requires more Java 2 security permissions than are granted in the default policy.
- For the pure Java client application or client container application under WebSphere version 5, you haven't to validate immediately after userid/password login, but wait for method request to send userid/password to server to validate them during the JAAS login. To do, you have to disable the **com.ibm.CORBA.validateBasicAuth** property in the **WAS-INSTALL/properties /sas.client.props** file.
- For debug, it is recommended to enable root cause login exception propagation to pure Java clients if you are in a trusted environment. To do click **Security > Global Security > Custom Properties** on the WebSphere Application Server administrative console and set the following property:

  com.ibm.websphere.security.registry.propagateExceptionsTo Client=true

8.4 JAAS Configuration File in RMI Client

This section describes how to configure JAAS in the RMI client of EJB deployed into WebSphere.

The client does not indicate SAML Login Module but **com.ibm.ws.security.common.auth.module.WSClientLoginModuleImpl** (WebSphere). This module does not authenticate user, but after retrieving login and password from callback handler links these two parameters to the EJB invocation. The Authentication occurs on server side.

```
Sample {
   com.ibm.ws.security.common.auth.module.proxy.WSLoginModuleProxy required
   delegate=com.ibm.ws.security.common.auth.module.WSClientLoginModuleImpl;
};
```

J2EE application clients can be launched using the **launchClient** tool. In this case you can add the client Login Module entry on **USER_INSTALL_ROOT/wsjaas_client.conf** file.

Important

If you indicate SAML Login Module you authenticate on client side. When you retrieve the Subject and try to execute an action as is, WebSphere will reject your Principal.

9. Oracle Application Server Platform

This chapter describes platform specific considerations to take into account the Evidian SAML Login Module under Oracle Application Server for J2EE Container (OC4J).

This chapter describes through an example, all the configuration steps of an application, so that the Evidian SAML Login Module performs the authentication, and so that the SAML assertion is propagated to the various parts of the application.

The application used as an example is made of three parts:

- a light html client,
- a servlet which runs in the OC4J web container,
- an ejb application which runs in the OC4J Application Server.

This application is the Oracle "helloworld" demonstration which can be downloaded from the Oracle web site. This application must be built from the source files, with the ant tool, to constitute a deployment entity : the "helloworld.ear" file.

In this chapter is shown the building of this "helloworld" application, the deployment in the OC4J environment of this application, the configuration of this application so that it performs an authentication through the Evidian SAML Login Module: the SAML assertion is generated by the Evidian SAML Login Module, returned in the servlet context, and propagated from the Web Container environment to the EJB environment.

9.1 Prerequisites, Versions

9.1.1 Prerequisites

SAM Web installed, and configured as mentionned in Chapter 2.

SAM J2EE installed, and configured as mentionned in Chapter 2.

9.1.2 Versions of Software Used

Please check the last versions of the software in the Release Notes.

Note

Please note that versions of these software used when writing this documentation were the following ones:

- OS : RedHat AS 3
- Java : JDK 1.4.2_09
- Ant : 1.6.5
- OC4J : 10.1.3 Developer Preview 4
- SAM Web : 7.0.1.27
- SAM J2EE : 1.2.4.5

9.2 Installing the Software

9.2.1 Installing Ant

1. Download ant from http://ant.apache.org/bindownload.cgi (URL valid in June 2006):
2. Create $HOME/ANT_HOME directory

   ```
   PROMPT> mkdir $HOME/ANT_HOME
   ```

3. Unzip **apache-ant-1.6.5-bin.zip** in **$HOME/ANT_HOME**

   ```
   PROMPT> cd $HOME/ANT_HOME
   PROMPT> unzip <file name.zip>
   ```

4. Add the ant binary to the PATH

   ```
   PROMPT> export PATH=$PATH:$ANT_HOME/filename/bin
   ```

9.2.2 Installing OC4J

1. Download OC4J from http://www.oracle.com/technology/tech/java/oc4j/1013/index.html (URL valid in June 2006)
2. Create **$HOME/OAS** directory

   ```
   PROMPT> mkdir $HOME/OAS
   ```

3. Unzip the downloaded OC4J in **$HOME/OAS**

   ```
   PROMPT> cd $HOME/OAS
   PROMPT> unzip <file name.zip>
   ```

4. Set the environment variables

   ```
   PROMPT> export OAS_HOME=$HOME/OAS
   PROMPT> export ORACLE_HOME=$OAS_HOME
   PROMPT> export J2EE_HOME=$OAS_HOME/j2ee/home
   PROMPT> export PATH=$OAS_HOME/bin:$PATH
   ```

9.2.3 Installing SAM Web

Please refer to Section 2.

9.2.4 Installing SAM J2EE

Please refer to Section 2.

1. Download SAM J2EE from http://support.evidian.com/

   Note: We suggest you to copy SAM J2EE files into a OC4J subdirectory to facilitate future steps.

2. Create the Evidian SAM Web directory

   ```
   PROMPT> mkdir $J2EE_HOME/evidian
   ```

3. Unzip and untar the **<file name>.tar.gz** in **$J2EE_HOME/evidian**

   ```
   PROMPT> cd $J2EE_HOME/evidian
   PROMPT> gunzip <filename>.tar.gz
   ```

```
PROMPT> tar xvf <filename>.tar
```

> This produces the following:

bin/samlpsw
bin/samlloadca
bin/samltestcfg
bin/samlpwdcrypt
bin/libz.so
bin/libsamllogin.so
config/cache.properties.tmpl
config/credential.properties.tmpl
config/jaas.config.tmpl
config/principal.properties.tmpl
config/security.cfg.tmpl
lib/samllogin.jar

4. Set the environment variables:
   - CLASSPATH is set to **$J2EE_HOME/evidian/config** directory so that the property files, which are going to be written in this directory, are accessible.
     PROMPT> export **CLASSPATH=$J2EE_HOME/evidian/config**
   - LD_LIBRARY_PATH is set to **$J2EE_HOME/evidian/bin** so that the lpfsamllogin.so shared library, which is stored in it, is accessible.
     PROMPT> export LD_LIBRARY_PATH=**$J2EE_HOME/evidian/bin**
   - SAMLCONFIG indicates where the configuration file is stored.
   - PROMPT> export SAMLCONFIG=**$J2EE_HOME/evidian/config/security.cfg**

9.2.4.1 security.cfg

Here is an example of **security.cfg** file to be written from the **$J2EE_HOME/evidian/config/security.cfg.tmpl** file.

```
<?xml version="1.0"?>
<samlConfig>
  <general type="verifier,appclient"
        base="/bronco/desgranp/LM/"
        traceFile="trace/%p/%t/saml.trc"
        traceLevel="0xf97f">
  </general>
  <Connection>
    <Verification CaPath="ca.d" CrlPath="crl.d" Depth="5"/>
    <Ssl Method="all,-tlsv1"
Cipher="ALL:!ADH:!EXPORT56:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP:+eNULL"/>
  </Connection>
  <Signature Identity="client Saml1.1">
    <Verification CaPath="ca.d" Depth="5"/>
  </Signature>

  <samlserver
        type="authentication"
        domain="local">
     <query
        nonce="nonce"
        user="user"
        password="password"
        newpassword="newPassword">
        uri=/form_and_SAML/%26samlVersion=1.1%26version=1.0
     </query>
     <url>http://yale.frec.bull.fr:9139/pxpadmin/bin/authform.cgi</url>
  </samlserver>

  <samlserver
        type="renew"
        domain="local">
      <query
        nonce="nonce"
        user="user"
        policy="policy">
    uri=/form_and_SAML/%26samlVersion=1.1%26version=1
      </query>
      <authentication
        type="basic"
        user="pade" password="pade"/>
      <url>http://yale.frec.bull.fr:9139/pxpadmin/renewal/</url>
   </samlserver>
  <samlserver type="ca">
     <url>http://yale.frec.bull.fr:9128/bin/ca.sh</url>
  </samlserver>
</samlConfig>
```

9.2.4.2 principal.properties

Here is an example of **principal.properties** file, to be written from the **$J2EE_HOME/evidian/config/principal.properties.tmpl** file.

```
PxPUid=com.evidian.security.auth.samples.OASPrincipal
```

Java class indicated above must be developed, compiled and added to the jar file containing the Evidian SAML Login Module, as indicated in Section 9.3, "Building the samlloginoas.jar".

There must be in the SAMWeb configuration an Injection Data in the "SAML" Injection Data with the saml tag set to PXPUid and with user attribute "uid".

This enables that a Principal is set in the SAML Assertion with the uid of the authenticated user.

9.2.4.3 credential.properties

Here is an example of credential.properties file, to be written from the **$J2EE_HOME/evidian/config/credential.properties.tmpl** file.

```
credential=com.evidian.security.auth.samples.OASCredential
```

Java class indicated above must be developed, compiled and added to the jar file containing the Evidian SAML Login Module, as indicated in 10.3 paragraph.

9.2.4.4 cache.properties

Here is an example of cache.properties file, to be written from the **$J2EE_HOME/evidian/config/cache.properties.tmpl** file.

```
seed=9IJUHYTGRFderftg
```

9.2.5 Installing Oracle EJB Demos

1. Download EJB Demos from
   http://www.oracle.com/technology/tech/java/oc4j/demos/904/index.html
   (URL valid in June 2006)

   ```
   PROMPT> cd $J2EE_HOME/applications
   PROMPT> gunzip <filename>.zip
   ```

   Note: Example of interest can be found in the **$J2EE_HOME/applications/ejb/helloworld directory**

2. Modify and build this example as shown in Section 9.3, "Building the samlloginoas.jar".

9.3 Building the samlloginoas.jar

Add to the **samllogin.jar** the **OASPrincipal** and **OASCredential** classes which can be developped as indicated in the following sections.

The result (**samllogin.jar** and these two classes) constitutes a new jar file: **samlloginoas.jar**.

9.3.1 Develop your own Principal Class

```
package com.evidian.security.auth.samples;
import java.io.Serializable;
import java.util.Arrays;
import java.util.ArrayList;
import java.util.List;
import java.security.Principal;
import com.evidian.security.auth.common.SamlLoginException;
import com.evidian.security.auth.interfaces.SecurityContext;
import com.evidian.security.auth.saml.provider.MessageBuffer;

/**
*/
public class OASPrincipal
        implements Principal,
                Serializable,
                SecurityContext {


    private String name;
                byte[] assertion = null;
     private int    hashCode;
```

```
/**
  */
        public OASPrincipal(String name) {
         this.setName(name);
        }

/**
  */
public String getName() {
      return this.name;
}

        /**
         * @see
com.evidian.security.auth.interfaces.SecurityContext#getAssertion()
         */
        public byte[] getAssertion() {
            return (byte[]) (this.assertion).clone();
        }

/**
  */
public void setName(String name) {
       this.name = name;
}

        /**
         * @see
com.evidian.security.auth.interfaces.SecurityContext#setAssertion(byte[])
         */
        public void setAssertion(byte[] assertion) {
               this.assertion = assertion;
       this.hashCode  = new String(this.assertion).hashCode();
        }

    /**
     */
        public boolean equals(Object obj) {
                if (obj == null)
                                        return false;
                if (obj == this)
                    return true;
                if (!(obj instanceof OASPrincipal))
                    return false;

                OASPrincipal another = (OASPrincipal) obj;
                if (!Arrays.equals(this.assertion,

another.getAssertion())) {
                    return false;
                } else {
```

```
                                if (this.getName() == null) {
                        if (another.getName() == null) {
                        return true;
                        } else {
                        return false;
                       }
                } else {
                    return (this.getName().equals(another.getName()));
                }
                  }
        }

/**
  */
        public int hashCode() {
      return this.hashCode;
        }

/**
   * Returns a String representation of this object, which exposes only
information that should be public.
   */
  public String toString() {
        String returned  = "OASPrincipal";
        //
        returned = returned +
            "\n" + "name      = " + this.getName()  +
            "\n" + "password  = **************"     +
            "\n" + "assertion = " + new String(this.getAssertion()) +
            "\n";
        return returned;
    }
}
```

9.3.2 Develop your own Credential Class

```
package com.evidian.security.auth.samples;
import java.util.Date;
import com.evidian.security.auth.common.SamlLoginException;
import com.evidian.security.auth.saml.provider.MessageBuffer;
import com.evidian.security.auth.saml.SamlAssertion;

/**
 * This class is a sample of what should be developped by SAML LM customer.
 * <p>
 * As a result of the authentication, a SAM assertion has been generated and
is used to populate the Subject.
 *
 * When returning to the customer application, the application has indicated
the way they wanted the credential to be
 *  instanciated
 * This class extends the SamlAssertion to be registrable by the factory in
LoginModule at execution time.
 */
public class OASCredential
    extends SamlAssertion {

 /**
         * @param newCredential
         * @param newExpirationDate
         */
        public OASCredential(byte[] newCredential,
                                    Date   newExpirationDate) {
                   super(newCredential,
                       newExpirationDate);
         if (newCredential!=null) {
             String sNewCredential = new String(newCredential);
         }
        }

        /**
         * implements interface of OAS contract (non-Javadoc)
         */
        public String getXmlAssertion() {
            return new String(this.getCredential());
        }

 /**
   * Returns a String representation of this object, which exposes only
information that should be public.
   */
public String toString() {
         String returned = "OASCredential";
         returned = returned +
             "\n" + "credential = " + this.getXmlAssertion() +
```

```
            "\n";
        return returned;
    }
}
```

9.3.3 Build the jar

Unjar the **samllogin.jar**, and jar the result with the two classes into the **samlloginoas.jar**. This jar is put in the **$J2EE_HOME/evidian/jar** directory. This jar is going to be added in he OC4J environment as a shared library entity, as indicated in Section 9.7.3, "Shared Library Step".

9.4 Building the helloworld.ear

We are testing with the ejb "helloworld" demonstration found on the Oracle web site, which shows a light client, performing a servlet operation and an ejb operation.

This demonstration has been unzipped in the **$J2EE_HOME/applications/ejb/helloworld** directory.

But before building the helloworld demonstration, you can modify it a little to visualize Principal propagation during the test.

9.4.1 Modify the Servlet Code

To visualize the right propagation of the Principal to the servlet environment, the ‘doGet’ method of the **$J2EE_HOME/applications/ejb/helloworld/src/web/helloworld-web/HelloServlet.java** file must be modified the following way:

```
/**
  */
 public void doGet(HttpServletRequest request,
                   HttpServletResponse response)
     throws ServletException,
            IOException    {
     response.setContentType("text/html");
     ServletOutputStream out = response.getOutputStream();
     try {
         out.println("<html>");
         out.println("<body>");
         out.println(hello.sayHello("James Earl"));
```

```
            // DISPLAY SECURITY CONTEXT
            try {
                out.println("<li>DISPLAY SECURITY CONTEXT  1 </li>");
                out.println("request= "+request.toString() + "\n");
                String authtype = request.getAuthType();
                out.println("authtype = " + authtype + "\n");
                String remoteUser = request.getRemoteUser();
                out.println("remoteUser = " + remoteUser + "\n");
                Principal principal =  request.getUserPrincipal();
                if (principal!=null)
                    out.println("principal = "+principal+"
"+principal.toString() + "\n");
                else
                    out.println("principal = "+principal + "\n");
            } catch (Exception e) {
                out.println("<li>Cannot DISPLAY SECURITY CONTEXT 1 : " + e +
"</li>");
                e.printStackTrace();
            }


            //
            try {
                out.println("<li>DISPLAY SECURITY CONTEXT  2 </li>");
                AccessControlContext ctx               =
AccessController.getContext();
                Subject              sub               =
Subject.getSubject(ctx);
                Set                  principals        =
sub.getPrincipals();
                Iterator             principalsIterator =
principals.iterator();
                if (principalsIterator!=null) {
                    int ii = 0;
                    while (principalsIterator.hasNext()) {
                        ii++;
                        Object principal2 = (Object)
principalsIterator.next();
                        if (principal2 != null) {
                            String s = principal2.getClass().getName()+ " "
+principal2;
                            out.println(" s "+ii+" from Subject = "+s +
"\n");
                        }
                    }
                }
            } catch (Exception e) {
                out.println("<li>Cannot DISPLAY SECURITY CONTEXT 2 : " + e +
"</li>");
                e.printStackTrace();
```

```
            }


            //
            try {
                out.println("<li>DISPLAY SECURITY CONTEXT  3 </li>");
                SecurityManager      securityManager      =
System.getSecurityManager();
                AccessControlContext accessControlContext = null;
                Subject              sub                  = null;
                if (securityManager!=null) {
                    accessControlContext =
(AccessControlContext)securityManager.getSecurityContext();
                    if (accessControlContext!=null) {
                        sub = Subject.getSubject(accessControlContext);
                        if (sub != null) {
                            Set      principals         =
sub.getPrincipals();
                            Iterator principalsIterator =
principals.iterator();
                            //
                            if (principalsIterator!=null) {
                                int ii = 0;
                                while (principalsIterator.hasNext()) {
                                    ii++;
                                    Object principal2 = (Object)
principalsIterator.next();
                                    if (principal2 != null) {
                                        String s =
principal2.getClass().getName()+ " " +principal2;
                                        out.println(" s "+ii+" from Subject
= " +s + "\n");
                                    }
                                }
                            }
                        } else {
                            out.println(" sub = " + sub +"\n");
                        }
                    } else {
                        out.println(" accessControlContext = " +
accessControlContext + "\n");
                    }
                } else {
                    out.println(" securityManager = " + securityManager +
"\n");
                }
            } catch (Exception e) {
                out.println("<li>Cannot DISPLAY SECURITY CONTEXT 3 : " + e +
"</li>");
                e.printStackTrace();
            }
```

```
            out.println("</body>");
            out.println("</html>");
        } catch(EJBException e) {
            out.println("EJBException error: " + e.getMessage());
        } catch(IOException e) {
            out.println("IOException error: " + e.getMessage());
        } finally {
            out.close();
        }
    }
```

Add the following import as well

```
import java.security.Principal;
import javax.security.auth.Subject;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.*;
import java.security.AccessControlContext;
import java.security.AccessController;
import java.util.Iterator;
import java.util.Set;
```

9.4.2 Modify the ejb Code

To visualize the right propagation of the Principal to the ejb environment, the 'sayHello' method of the **$J2EE_HOME/applications/ejb/helloworld/src/ejb/helloworld-ejb/HelloBean.java** file must be modified the following way:

```
    public String sayHello(String myName)
        throws EJBException {
        Principal thePrincipal =  this.ctx.getCallerPrincipal();
        System.out.println("HelloBean.sayHello("+myName+") thePrincipal =
"+thePrincipal);

        Object p = this.ctx.getCallerPrincipal();
        System.out.println("HelloBean.sayHello("+myName+") p =
"+p.getClass().getName());
        System.out.println("HelloBean.sayHello("+myName+") p = "+p);

        AccessControlContext ctx               =
AccessController.getContext();
        Subject              sub               = Subject.getSubject(ctx);
        Set                  principals        = sub.getPrincipals();
        Iterator             principalsIterator = principals.iterator();
        if (principalsIterator!=null) {
```

```
            int ii = 0;
            while (principalsIterator.hasNext()) {
                 ii++;
                 Object principal = (Object) principalsIterator.next();
                if (principal != null) {
                    String s = principal.getClass().getName()+ "
"+principal;
                    System.out.println("HelloBean.sayHello ("+myName+") s
"+ii+" = "+s);
                }
            }
        }
```

Add the following import as well

```
import java.security.Principal;
import javax.security.auth.Subject;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.*;
import java.security.AccessControlContext;
import java.security.AccessController;
import java.util.Iterator;
import java.util.Set;
```

9.4.3 Build the jar

This demo must be built from the following command

```
PROMPT> cd $J2EE_HOME/applications/ejb/helloworld
PROMPT> ant
```

The ear file produced can be found here:
$J2EE_HOME/applications/ejb/helloworld/dist/helloworld.ear

9.5 Complete the SAM Web Configuration

As the ‘smith’ user is used during the test, create the "smith" user with the "smithj" password in your Users Directory, if it does not exist yet.

9.6 Verify the SAM J2EE Configuration

9.6.1 configuration.cfg

This file must exist, be accessible through the SAMLCONFIG environment variable, and be configured as indicated in Section 2, "Installing SAML Login Module".

9.6.2 Properties Files

Principal.properties, credential.properties, cache.properties must exist, be accessible through the CLASSPATH environment variable, and be configured as indicated in Section 2, "Installing SAML Login Module".

9.7 Configuring OC4J

9.7.1 Starting OC4J at First Time

It can be started with the following command line

```
PROMPT> java -Djavax.net.debug=all -
Djazn.debug.log.enable=true -jar $J2EE_HOME/oc4j.jar -
verbosity 10
```

> This produces the following output.

Set for the oc4jadmin login, the oc4jadmin password.

```
yale.frec.bull.fr.desgranp$java -Djavax.net.debug=all -
Djazn.debug.log.enable=true -jar $J2EE_HOME/oc4j.jar -verbosity 10
05/12/16 15:21:30 Auto-unpacking
/bronco/desgranp/OAS/j2ee/home/applications/admin_web.war... done.
05/12/16 15:21:30 Auto-unpacking
/bronco/desgranp/OAS/j2ee/home/applications/dms0.war... done.
05/12/16 15:21:30 Auto-unpacking
/bronco/desgranp/OAS/j2ee/home/applications/JMXSoapAdapter-web.war... done.
05/12/16 15:21:30 Auto-unpacking
/bronco/desgranp/OAS/j2ee/home/applications/jmsrouter.war... done.
05/12/16 15:21:30 Auto-unpacking
/bronco/desgranp/OAS/j2ee/home/connectors/datasources/datasources.rar...
done.
```

```
05/12/16 15:21:30 Auto-unpacking
/bronco/desgranp/OAS/j2ee/home/connectors/OracleASjms/OracleASjms.rar...
done.
05/12/16 15:21:30 Auto-unpacking
/bronco/desgranp/OAS/j2ee/home/applications/ascontrol.ear... done.
05/12/16 15:21:31 Auto-unpacking
/bronco/desgranp/OAS/j2ee/home/applications/ascontrol/ascontrol.war... done.
05/12/16 15:21:32 Copying default deployment descriptor from archive at
/bronco/desgranp/OAS/j2ee/home/applications/ascontrol/META-INF/orion-
application.xml to deployment directory
/bronco/desgranp/OAS/j2ee/home/application-deployments/ascontrol...
05/12/16 15:21:32 Set OC4J administrator's password (password text will not
be displayed as it is entered)
Enter password:
Confirm password:
The password for OC4J administrator "oc4jadmin" has been set.
05/12/16 15:21:49 The OC4J administrator "oc4jadmin" account is activated.
05/12/16 15:22:12 Copying default deployment descriptor from archive at
/bronco/desgranp/OAS/j2ee/home/applications/admin_ejb.jar/META-INF/orion-
ejb-jar.xml to deployment directory
/bronco/desgranp/OAS/j2ee/home/application-deployments/admin_ejb...
05/12/16 15:22:15 Copying default deployment descriptor from archive at
/bronco/desgranp/OAS/j2ee/home/applications/admin_web/WEB-INF/orion-web.xml
to deployment directory /bronco/desgranp/OAS/j2ee/home/application-
deployments/admin_web...
05/12/16 15:22:15 Copying default deployment descriptor from archive at
/bronco/desgranp/OAS/j2ee/home/applications/JMXSoapAdapter-web/WEB-
INF/oracle-webservices.xml to deployment directory
/bronco/desgranp/OAS/j2ee/home/application-deployments/JMXSoapAdapter-web...
05/12/16 15:22:16 Application system (system) initialized...
05/12/16 15:22:16 Copying default deployment descriptor from archive at
/bronco/desgranp/OAS/j2ee/home/connectors/datasources/datasources/META-
INF/oc4j-ra.xml to deployment directory
/bronco/desgranp/OAS/j2ee/home/application-
deployments/default/datasources...
05/12/16 15:22:16 Copying default deployment descriptor from archive at
/bronco/desgranp/OAS/j2ee/home/connectors/OracleASjms/OracleASjms/META-
INF/oc4j-ra.xml to deployment directory
/bronco/desgranp/OAS/j2ee/home/application-
deployments/default/OracleASjms...
05/12/16 15:22:17 Copying default deployment descriptor from archive at
/bronco/desgranp/OAS/j2ee/home/applications/jmsrouter-ejb.jar/META-
INF/orion-ejb-jar.xml to deployment directory
/bronco/desgranp/OAS/j2ee/home/application-
deployments/default/jmsrouter_ejb...
05/12/16 15:22:20 Application default (default) initialized...
05/12/16 15:22:21 Copying default deployment descriptor from archive at
/bronco/desgranp/OAS/j2ee/home/connectors/datasources/datasources/META-
INF/oc4j-ra.xml to deployment directory
/bronco/desgranp/OAS/j2ee/home/application-
deployments/ascontrol/datasources...
```

```
05/12/16 15:22:21 Copying default deployment descriptor from archive at
/bronco/desgranp/OAS/j2ee/home/applications/ascontrol/ascontrol/WEB-
INF/orion-web.xml to deployment directory
/bronco/desgranp/OAS/j2ee/home/application-
deployments/ascontrol/ascontrol...
05/12/16 15:22:21 Application ascontrol (Oracle Enterprise Manager - OC4J
Studio) initialized...
05/12/16 15:22:23 Oracle Containers for J2EE 10g (10.1.3.0.0) - Developer
Preview 4 initialized
```

9.7.2 Launching the Console

Configuration must be done by using the Oracle Application Server Control Console:

1. Launch the following command:

   ```
   PROMPT> firefox http://localhost:8888/em &
   ```

2. Authenticate as administrator (login: oc4jadmin, password: oc4jadmin).

9.7.3 Shared Library Step

Shared library Configuration Step: create a shared library for the Evidian SAML Login Module.

1. In the **Administration** tab, expand the **Properties** folder, go to task **Shared libraries**and perform the three steps to create a Shared Library.

   Attributes:

   – Shared Library Name: Evidian
   – Shared Library Version: 1.0
   – Add Archives: Add
     File is present on local host: $J2EE_HOME/evidian/jar/samlloginoas.jar

2. Upload the file, enable the **import** check box, and click **Finish**.

   > You can see now the Evidian shared library in the list of shared libraries.

9.7.4 Helloworld Deploiement Step

1. Go to the OC4J: Home page
2. In the **Application** tab, click **deploy** to deploy the helloworld application.
3. Follow the steps, as described below:

Deploy: Select Archive step

- In the **Archive** part, enable the following check box:
  Archive is present on local host. Upload archive to the server where Application Server Control is running.
- Add the path to your ear file:
  $J2EE_HOME/applications/ejb/helloworld/dist/helloworld.ear
- In the **Deployment plan** part, enable the following check box:
  Automatically create a new deployment plan
  All coming configuration steps will be stored in a file called **deployment_plan.dat**, which can be used as input file for future redeployment.

Deploy: Application Attribute step.

- Application Name: helloworld
- Parent Application: default
- Bind Web Module to Site: default-web-site

Deploy: Deployment Settings step.

- In the '**Deployment Tasks**' part, there are several tasks to perform.
  - Go to task: **Select Security Provider**
    - Security Provider must be choosen **Custom**
    - Click **Add Login Module**
    - **JAAS Login Module class** must be set to **com.evidian.security.auth.login.SamlLoginModule**
    - **Login Module Control Flag** must be set to '**Required'**
    - Add then two properties, by clicking on '**Add another row**'.

request	id-password
debug	true

  - Go to task: Map Security Roles
    Add "smith" user for both modules 'helloworld' and 'helloworld web application'.

  - Go to task: Configure Class Loading
    Enable the 'import' check box for the 'Evidian' shared library.
- In the **Advanced Deployment Plan editing** part, select the **helloworld j2ee application** in the browsing tree on the left, ans click on the **edit jazn** link.
  - Provider must be set to 'XML'.
  - authMethod must be set to 'BASIC'.
  - 'doAsPrivilegedMode' and 'runAsMode' must be set to 'true'.

  Two properties must be set:
  - custom.loginmodule.provider true
  - role.mapping.dynamic true

4. Once all deployment steps are performed, save your deployment plan, and click **deploy**.

9.7.5 Use a Deployment Plan

For future redeployment, you can base your deployment on the **helloworld_plan.dat** deployment plan, which should set everythings (or most of the thing as there seems to be some information missing) all right:

```
moduleType=ear
applicationID=helloworld
bindWebApp=default-web-site
parent=default
moduleID=.
doctype=orion-application
<?xml version="1.0"?>
<orion-application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://xmlns.oracle.com/oracleas/schema/orion
-application-10_0.xsd">
        <client-module path="helloworld-client.jar">
        </client-module>
        <jazn provider="XML">
                <jazn-web-app auth-method="BASIC" runas-mode="true">
                </jazn-web-app>
                <property name="role.mapping.dynamic" value="true">
                </property>
                <property name="custom.loginmodule.provider" value="true">
                </property>
        </jazn>
        <imported-shared-libraries>
                <import-shared-library name="evidian">
                </import-shared-library>
        </imported-shared-libraries>
```

```
        <jazn-loginconfig>
                <application Key="name">
                        <name>helloworld
                        </name>
                        <login-modules>
                                <login-module Key="class control-flag">

        <class>com.evidian.security.auth.login.SamlLoginModule
                                        </class>
                                        <control-flag>required
                                        </control-flag>
                                        <options>
                                                <option Key="name value">
                                                        <name>debug
                                                        </name>
                                                        <value>true
                                                        </value>
                                                </option>
                                                <option Key="name value">
                                                        <name>request
                                                        </name>
                                                        <value>id-password
                                                        </value>
                                                </option>
                                        </options>
                                </login-module>
                        </login-modules>
                </application>
        </jazn-loginconfig>
</orion-application>
moduleID=helloworld-web.war
doctype=orion-web-app
<?xml version="1.0"?>
<orion-web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://xmlns.oracle.com/oracleas/schema/orion
-web-10_0.xsd">
        <security-role-mapping impliesAll="false" name="users">
                <group name="users">
                </group>
                <user name="oc4jadmin">
                </user>
                <user name="smith">
                </user>
        </security-role-mapping>
        <web-app-class-loader search-local-classes-first="true">
        </web-app-class-loader>
</orion-web-app>
moduleID=helloworld-client.jar
doctype=orion-application-client
<?xml version="1.0"?>
```

```
<orion-application-client xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xsi:noNamespaceSchemaLocation="http://xmlns.oracle.com/oracleas/schema/orion
-application-client-10_0.xsd">
</orion-application-client>
moduleID=helloworld-ejb.jar
doctype=orion-ejb-jar
<?xml version="1.0"?>
<orion-ejb-jar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://xmlns.oracle.com/oracleas/schema/orion
-ejb-jar-10_0.xsd">
        <enterprise-beans>
                <session-deployment name="HelloBean">
                </session-deployment>
        </enterprise-beans>
        <assembly-descriptor>
                <security-role-mapping impliesAll="false" name="users">
                        <group name="users">
                        </group>
                        <user name="oc4jadmin">
                        </user>
                <user name="smith">
                        </user>
                </security-role-mapping>
        </assembly-descriptor>
</orion-ejb-jar>
```

9.8 Launch the Test

```
PROMPT> firefox http://localhost:8888/helloworld &
```

Authenticate with login ‘smith’ and password ‘smithj’

What does it show?

Authentication is performed against the Evidian LM configured in OC4J, a SAML assertion is generated and added as credential in the subject, and as principal in the subject. The subject is propagated to the servlet context, and to the ejb context.The user for who this assertion is generated has the right to perform servelt and ejb operations.

Integration of Evidian SAML Login Module and OC4J 10.1.3 is successful for this scenario.

10. Login Modules on SDM (Security Data Manager)

10.1 Introduction

This login module consists in 2 functions based on the SDM as authentication interface :

- check-sdm: checks if the user has been authenticated already against Secure Access Manager, the login module checks if a SDM security context is established on the workstation.
- auth-sdm: authenticates the subject against SDM (with primary password renewal possibility). This function doesn't modify the workstation SDM security context.

10.2 Prerequisites

An AccessMaster Security Data Manager must be active on the workstation.

You must have configured the Security Data Manager (authentication with the cache or with the authentication server .....).

10.3 Details

10.3.1 check-sdm

This function checks the current SDM security context (established during the last authentication).

10.3.2 auth-sdm

You can use this function to authenticate a user if his name contains a maximum of 30 characters. You can provide the user's domain name if the domain name contains a maximum of 30 characters.

This function allows you to change your primary password :

- The old password must be provided
- The new password must be different from the previous passwords (the number of passwords memorized is chosen by the administrator).
- The format and the length of the password are matched against the policy.

This function returns :

- OK. The warnings are displayed in the saml.trc file.
- nonOK with an error message (with a SDM error code defined in SDM\iss_errs.h).

10.4 Installing SAML Login Module

10.4.1 samlloginjar

For install, refer to section 2.2.1 in chapter 2 "Installing SAML Login Module"

10.4.2 Security Module shared library

For install, refer to section 2.2.2 in Section 2 "Installing SAML Login Module"

10.4.3 jaas.config

Three authentication methods are used : `auth-sdm, auth-sdm-with-change and check-sdm.`

For more details about these authentication methods, refer to section 2.2.5 in Section 2 "Installing SAML Login Module".

10.4.4 security.cfg

In this case, security.cfg file is needed only for Activating Security Module Traces.

`refer to section 2.2.3` in chapter 2 “Installing SAML Login Module”.

Important

For Login Modules on SDM, SAM J2EEauthentication server is not interroged and no assertion is generated.