

# Oracle IDM: Authentication Primer

---

*More Than Just A Quick Peek!*

## Contents

License.....	3
Introduction .....	3
Chapter 1: Java Authentication and Authorization Service .....	3
What is JAAS?.....	3
Life without JAAS .....	3
Drawbacks of the implementation .....	4
JAAS: The Good Parts .....	4
Peek into JAAS driven Authentication .....	4
JAAS in action: Step by step .....	4
Sample JAAS based application.....	5
Required components.....	5
The summary of invocation .....	5
Conclusion.....	8
Chapter 2: Weblogic Security Providers .....	8
Basic terminology.....	8
Introduction: Weblogic Security Providers .....	12
Types .....	12
Authentication Providers .....	12
Identity Assertion Provider .....	13
Conclusion.....	14
Chapter 3: Oracle Identity Manager: Authentication Deep Dive.....	14
Authentication related scenarios/use cases .....	14
Authentication Process: Under the covers .....	16
Logging into OIM Self Service or Administrative console .....	16
Dissecting the OIM Authentication Provider .....	17
Using the OIM Design Console (standalone <i>thick client</i> ) or OIM APIs in a standalone manner...	18
The Signature Login process .....	21
Conclusion.....	21
Closing Thoughts.....	21

## License

This work is licensed under the **Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License**. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/4.0/>.

## Introduction

Due to realistic restrictions like bandwidth and scope it is often not possible to discuss the standards/technologies/techniques in details via blogs. The content within this document has more to do with **a particular set of technology/standards** on which the a specific product (OIM) functionality is based.

The **Java Authentication and Authorization API (JAAS)** and **Weblogic Security** services are based on Java standards which are by no means new. I decided to include them in order to provide a background for readers before we jump on to the core part which discusses/dissects **OIM Authentication** process.

Feel free to skip directly to **Chapter 3: Oracle Identity Manager: Authentication Deep Dive** in case your already familiar with JAAS, Weblogic and general Java EE Security concepts

## Chapter 1: Java Authentication and Authorization Service

### What is JAAS?

The Java Authentication and Authorization Service (JAAS) is a standard API used for building Authentication and Authorization features into Java based applications.

JAAS itself is based upon and is the Java implementation of the **Pluggable Authentication Module (PAM)** framework. *A detailed discussion of PAM is out of scope of this document.*

### Life without JAAS

Imagine a scenario where *you are asked to develop a standalone desktop application for end users.*

This application would be distributed/installed on the end users' work stations and they would have to sign into the application using their credentials.

As a part of the initial requirements, you have been told that the user authentication repository is a standard *SQL Database*. You would probably leverage Swing/AWT or JavaFX for the application development itself and are likely to embed the Authentication logic as standard *JDBC* code somewhere within the business logic itself - assuming you are unaware of JAAS ;-)

This approach is fine, until one day, the customer decides to switch to a shiny *new LDAP server* as their trusted user repository. What now?

Of course, the authentication portion of your desktop app needs to be rewritten. Now you would probably leverage *Java Naming Directory Interface (JNDI)* API to access the customer's LDAP repository and authenticate your users.

Fair enough.

So, lots of code changes, lots of testing . . but you finally made it!

*What if you had an easier way? What if you knew about the **JAAS** API?*

### Drawbacks of the implementation

- Authentication/security logic embedded or mixed with the business logic
- Lots of code changes and refactoring in order to make up for the change in authentication repository - hence more time to market
- What if you need to implement authorization as an application feature? Wow.... we are looking at some major work here!

### JAAS: The Good Parts

This is where JAAS comes into the picture and it's probably the right juncture to *talk about its positives*

- JAAS allows authentication and authorization to be done in a pluggable manner
- Application is completely agnostic of the underlying intricacies of authentication repository
- Application does not need to aware of the authentication technology
- Allows new authentication protocols/technologies to be smoothly integrated within an application
- Allows for multiple layers of authentication to be enforced upon an application without breaking a sweat!
- With JAAS, your authentication related code might span only a few lines (at least for basic applications)

### Peek into JAAS driven Authentication

Let's take a quick code driven tour of the JAAS API, it's major components and how to use it for authentication

The following classes/interfaces drive the basic JAAS functionality as far as authentication is concerned.

- `javax.security.auth.login.LoginContext` class
- `javax.security.auth.spi.LoginModule` interface
- `javax.security.auth.callback.Callback` interface
- `javax.security.auth.callback.CallbackHandler` interface

### JAAS in action: Step by step

How do the above components work in tandem?

1. It all begins when the application creates an instance of the **`javax.security.auth.login.LoginContext`** class and passes the name of the module (configured in a configuration file pointed by the) and the call back handler instance
2. The JAAS configuration file consists of the Login Module implementation(s)
3. The call back handler is an implementation of the **`javax.security.auth.callback.CallbackHandler`** interface. The call back handler provides a way to *collect the credentials* meant for authentication.
4. The Login Module implementation sends callbacks (instances of the implementation for the **`javax.security.auth.callback.Callback`** interface) to the call back handler. Think of the callback as *types of challenges* e.g. a username and password. The call back handler's job is to *propagate* this challenge to the user e.g. in the form of a prompt and propagate the response back to the caller i.e. the Login Module.
5. It's the job of the Login Module to use these credentials and *verify* them e.g. authenticate via a database and deem the overall authentication process as successful or failed.

**Note:** *These steps provide a high level breakup of the authentication process using the JAAS API. Please bear in mind that many of the finer details like handling of Subjects, abort, logout scenarios etc have not been covered in order to focus on the basic flow of authentication*

## Sample JAAS based application

Let's dive into some code and try to understand what happens under the covers during authentication process via a simple example. The components (introduced above) would be explained as and when they come into play within the context of the use case

*We'll use a command line application which prompts for the username and password and displays whether the authentication was successful*

The source code can be downloaded for reference from this [GitHub link](#)

## Required components

- Custom Login Module implementation -  
`com.wordpress.abhiroczkz.jaas.impl.CustomLoginModule.java`
- Custom Callback Handler implementation -  
`com.wordpress.abhiroczkz.jaas.impl.CustomCallbackHandlerImpl.java`
- `jaas.config` - The JAAS configuration file which contains the Login Module related details

## The summary of invocation

1. Authentication initiated by calling login method on the *Login Context* instance

```

public static void main(String[] args) {

    System.out.println("ENTER AppLauncher/main");

    System.setProperty("java.security.auth.login.config", "jaas.config");
    LoginContext loginContext = null;

    try {

        loginContext = new LoginContext("CustomLoginModule", new CustomCallbackHandlerImpl());

        System.out.println(Thread.currentThread().getName());
        System.out.println("LoginContext created");

    } catch (LoginException e) {
        System.err.println("Unable to create LoginContext!!!");
    }
}

```

2. The contents of the **jaas.config** file

```

CustomLoginModule{
    com.wordpress.abhirockzz.jaas.impl.CustomLoginModule
    required debug=true;
};

```

3. The **CustomLoginModule.java** initiates the authentication

```

package com.wordpress.abhirockzz.jaas.impl;

import java.util.Map;

public class CustomLoginModule implements LoginModule{

    private CallbackHandler callbackHandler = null;
    private boolean authenticated = false;
    private Subject subject;
    private Map<String, ?> sharedState;
    private Map<String, ?> options;
    private boolean committed = false;

    public void initialize(Subject subject, CallbackHandler callbackHandler,

    public boolean login() throws LoginException {

    @Override
    public boolean commit() throws LoginException {
        System.out.println("ENTER CustomLoginModule/commit");

        if(!authenticated){
            return false;
        }else{
            committed = true;
        }
        return true;
    }
}

```

4. Throws challenges (callbacks) in the form of *javax.security.auth.callback.NameCallback* and *javax.security.auth.callback.PasswordCallback*

```

@Override
public boolean login() throws LoginException {
    System.out.println("ENTER CustomLoginModule/login");

    boolean result = true;

    Callback nameCBack = new NameCallback("Username: ");
    Callback pwdCBack = new PasswordCallback("Password: ", false);

    Callback[] callbacks = new Callback[2];
    callbacks[0] = nameCBack;
    callbacks[1] = pwdCBack;

    try {
        callbackHandler.handle(callbacks);
    } catch (Exception e) {
        result = false;
        e.printStackTrace();
    }
}

```

- These challenges are handled by the *CustomCallbackHandlerImpl* which further prompts the caller for the username and password

```

public class CustomCallbackHandlerImpl implements CallbackHandler{

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

        System.out.println("ENTER CustomCallbackHandlerImpl/handle");

        try(Scanner scanner = new Scanner(System.in)){
            for (int i = 0; i < callbacks.length; i++) {

                if(callbacks[i] instanceof NameCallback){
                    NameCallback cBack = (NameCallback) callbacks[i];
                    System.out.println(cBack.getPrompt());

                    String username = scanner.nextLine();

                    cBack.setName(username);

                }else if(callbacks[i] instanceof PasswordCallback){
                    PasswordCallback cBack = (PasswordCallback) callbacks[i];
                    System.out.println(cBack.getPrompt());
                    //Scanner scanner = new Scanner(System.in);
                    String password = scanner.nextLine();

                    cBack.setPassword(password.toCharArray());

                }
            }
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}

```

**Note:** Handling passwords/sensitive information in the form of plain String objects is not recommended. This has been used as an example and should not be adopted for production quality implementation

6. **CustomLoginModule** implementation checks the authenticity of the credentials collected as a response of the challenge/callback and deems the authentication process as success/failure. The commit method is called post successful authentication

**Note:** I have used dummy credentials in this example

```
    if((username.equals("Abhishek") && password.equals("Password"))){  
        System.out.println("Credentials verified!!");  
    }else{  
        result = false;  
    }  
    authenticated = result;  
    System.out.println("CustomLoginModule/login returning :"+ result);  
    System.out.println("EXIT CustomLoginModule/login");  
    return result;  
}
```

## Conclusion

This chapter was a gentle code/example driven introduction to the JAAS API. This should help you in understanding portions of **Chapter 3** which discusses about the application of JAAS within Oracle IDM

## Chapter 2: Weblogic Security Providers

### Basic terminology

Before diving into Security Providers, it's important to understand some basic Java EE related *Security terminology*

#### Realms

- A Realm is a security policy domain of a Java EE container (web/application). It serves as a security configuration layer on top of the applications deployed in a Java EE run time server.



**Summary of Security Realms**

A security realm is a container for the mechanisms—including users, groups, security roles, security policies, and security providers—that are used to protect WebLogic resources. You can have multiple security realms in a WebLogic Server domain, but only one can be set as the default (active) realm.

The Security Realms page lists each security realm that has been configured in the WebLogic Server domain. Click the name of the realm to explore and configure that realm.

[Customize this table](#)

**Realms (Filtered - More Columns Exist)**

New Delete Showing 1 to 1 of 1 Previous Next

Name	Default Realm
myrealm	true

New Delete Showing 1 to 1 of 1 Previous Next

↓  
default realm in Weblogic

- It encapsulates information about users (deemed as valid users of the deployed applications), groups, authentication repository/database.

**Settings for myrealm**

Configuration Users and Groups Roles and Policies Credential Mappings **Providers** Migration

**Authentication** Password Validation Authorization Adjudication Role Mapping Auditing Credential Mapping Certification Path Keystores

An Authentication provider allows WebLogic Server to establish trust by validating a user. You must have one Authentication provider in a security realm, and you can configure multiple Authentication providers in a security realm. Different types of Authentication providers are designed to access different data stores, such as LDAP servers or DBMS. You can also configure a Realm Adapter Authentication provider that allows you to work with users and groups from previous releases of WebLogic Server.

[Customize this table](#)

**Authentication Providers**

New Delete Reorder Showing 1 to 3 of 3 Previous Next

Name	Description	Version
DefaultAuthenticator	WebLogic Authentication Provider	1.0
OIMAuthenticationProvider	Provider that performs authentication thru the Oracle Identity Manager relational database	1.0
DefaultIdentityAsserter	WebLogic Identity Assertion provider	1.0

New Delete Reorder Showing 1 to 3 of 3 Previous Next

OOTB weblogic authenticator - uses embedded LDAP server as user repository for authentication

OOTB authenticator for OIM - uses the OIM DB as the default repository for authentication

Settings for OIMAuthenticationProvider

Configuration


Database information encapsulated within the OIM Authenticator

Common


Provider Specific

Save


This page allows you to configure additional attributes for this security provider.

 DBDriver:


oracle.jdbc.OracleDrive

 DBUrl:


jdbc:oracle:thin:@localh

 PKIKeystore Provider:


sun.security.rsa.SunRsa

 Symmetric Key Keystore Provider:


com.sun.crypto.provider


 DBPassword:

.....

 Please type again To confirm:

.....

☐  SSOMode

 DBUser:

dev\_oim

Save

- A realm is protected by a single unified security policy
- An web/application server can have multiple security realms each having their unique security configurations. Applications can be configured to leverage a particular realm for authentication and authorization purposes

### Users

Simply put, a User is a prospective client of the web application. As stated above, a Realm defines a the set of valid users which exist within a repository which the Realm interacts with e.g. RDBMS, LDAP servers etc. These end users are authenticated/authorized as per configuration in Security constraints (declarative or annotation based)

Settings for myrealm

Configuration **Users and Groups** Roles and Policies Credential Mappings Providers Migration

**Users** Groups

### Users of the 'myrealm' security realm - OOTB with OIM

This page displays information about each user that has been configured in this security realm.

Note: The authentication provider named OIMAuthenticationProvider does not support viewing or managing its users through the WebLogic console.

[Customize this table](#)

Users

New Delete Showing 1 to 3 of 3 Previous Next

<input type="checkbox"/>	Name	Description	Provider
<input type="checkbox"/>	bininternal		DefaultAuthenticator
<input type="checkbox"/>	OracleSystemUser	Oracle application software system user.	DefaultAuthenticator
<input type="checkbox"/>	weblogic	This user is the default administrator.	DefaultAuthenticator

New Delete Showing 1 to 3 of 3 Previous Next

## Groups

A Group is a set of users with common traits. These are container specific entities and are not related to the Java EE application itself. Groups provide a way to categorize users as per specific criteria and these are derived from the within the security realm configuration e.g. groups in an LDAP server.

Settings for myrealm

Configuration **Users and Groups** Roles and Policies Credential Mappings Providers Migration

**Users** Groups

### Groups in the 'myrealm' security realm - OOTB with OIM

This page displays information about each group that has been configured in this security realm.

Note: The authentication provider named OIMAuthenticationProvider does not support viewing or managing its groups through the WebLogic console.

[Customize this table](#)

Groups

New Delete Showing 1 to 9 of 9 Previous Next

<input type="checkbox"/>	Name	Description	Provider
<input type="checkbox"/>	AdminChannelUsers	AdminChannelUsers can access the admin channel.	DefaultAuthenticator
<input type="checkbox"/>	Administrators	Administrators can view and modify all resource attributes and start and stop servers.	DefaultAuthenticator
<input type="checkbox"/>	AppTesters	AppTesters group.	DefaultAuthenticator
<input type="checkbox"/>	CrossDomainConnectors	CrossDomainConnectors can make inter-domain calls from foreign domains.	DefaultAuthenticator
<input type="checkbox"/>	Deployers	Deployers can view all resource attributes and deploy applications.	DefaultAuthenticator
<input type="checkbox"/>	Monitors	Monitors can view and modify all resource attributes and perform operations not restricted by roles.	DefaultAuthenticator
<input type="checkbox"/>	oimusers	oimusers	DefaultAuthenticator
<input type="checkbox"/>	Operators	Operators can view and modify all resource attributes and perform server lifecycle operations.	DefaultAuthenticator
<input type="checkbox"/>	OracleSystemGroup	Oracle application software system group.	DefaultAuthenticator

## Roles

A Role is an Java EE application level term and should not be confused with a Group. Although they are interrelated, they are different entities. Roles are defined by the Java EE application developer itself and are mapped at runtime to the Groups in the application server to drive authentication and authorization (this mapping can be used as per application server vendor specific settings).

### **Principal**

A Principal is something which can be authenticated and is identifiable and inherits from *java.security.Principal*

### **Credentials**

Contains security attributes used to authenticate a Principal. It is set post authentication

### **Subject**

A Subject is an authenticated entity which contains the Principal and its associated Credentials

## **Introduction: Weblogic Security Providers**

Weblogic Security Providers are **pluggable components** which act as **agents** between the calling application and the Weblogic Security Framework. The providers are part of a Weblogic Security Realm (described above). It is possible to use multiple security providers in tandem within a Weblogic realm.

### **Types**

There are many types of Security Providers which Weblogic supports. The ones which are important from the context of this material have been mentioned below

- Authentication Provider
- Identity Assertion Provider

For the sake of information, here are some other categories of Weblogic Security Providers (this is not an exhaustive list)

- Principal Validation Providers
- Authorization Providers
- Adjudication Providers
- Role Mapping Providers

### **Authentication Providers**

As the name suggests, Weblogic Authentication providers make sure that the user/system has to prove its identity in order to access protected resources within Weblogic. It's a mandatory component of a Weblogic realm.

Following are the high level steps involved in the authentication process which involves invocation of the Authentication Provider

- The calling application is challenged for credentials when it tries to access a protected Weblogic resource - this process can vary depending upon the client itself e.g. a remote thick/desktop client might use client side JAAS implementation to challenge the user, and a web application might be challenged with a login form by the Java EE Web container as per the security settings within the web.xml
- The calling application fulfills the challenges and transmits the information back to Weblogic

- Weblogic Security Framework intercepts the info and calls upon the Authentication Providers
- Specifically speaking, the actual authentication done by the authentication provider is via the JAAS Login Module specific to the provider
- The provider Login Module authenticates the caller and in case of a success, the principal is signed and associated with a subject.

### ***Weblogic has several Authentication Providers available for use OOTB.***

These are not configured OOTB, like the Default Authenticator (which uses an embedded LDAP server). These providers can be used together (in tandem with each other) or individually. In addition to ready-to-use providers, the Weblogic Security Framework also allows developers to write custom implementations of Authentication Providers

- LDAP - The LDAP providers interface with external LDAP stores e.g. Microsoft AD, OID, OUD, iPlanet etc
- RDBMS - These set of providers interact with external databases for user, group and credentials information which is used to drive authentication and authorization
- Windows NT - As the name suggests, this provider uses Windows NT as its repository (for users and groups)

### **Identity Assertion Provider**

As the name clearly suggests, Identity Assertion Providers in Weblogic assert on the identity of a user/entity. This is also known as Perimeter Authentication. Assertion based authentication works on the concept of an already authenticated external entity which needs to be mapped to a user/group within the Weblogic realm. Identity Assertion also forms the basis of SSO in Weblogic

High Level steps involved in Identity Assertion process

- The caller propagates a token which contains the information about the entity which is attempting to assert itself to the Weblogic server
- If there is an Assertion Provider which is capable of validating the token sent by the caller, it processes it and determines the presence of an equivalent identity within its realm based on the mapping rules between the identity represented by the tokens and realm users.
- After the assertion process, the Identity Assertion implementation passes the mapped user information in the form of a JAAS CallbackHandler. This callback handler is consumed by the JAAS Login Module class which further executes the authentication process depending upon the username and populates the subject with the principal

### *OOTB Identity Assertion Providers in Weblogic support various tokens*

- LDAP X509 Identity Assertion Provider - supports X509 certificates as tokens embedded within LDAP attributes
- SAML Identity Assertion provider - Accepts SAML assertion tokens
- Negotiate Identity Assertion Provider - Extracts Kerberos tokens from SPNEGO tokens

## Conclusion

This chapter explored the a few Java EE Security tenets and the two major Weblogic Security Providers. These would of prime relevance when you venture into **Chapter 3 to** understand the OIM Authentication Process.

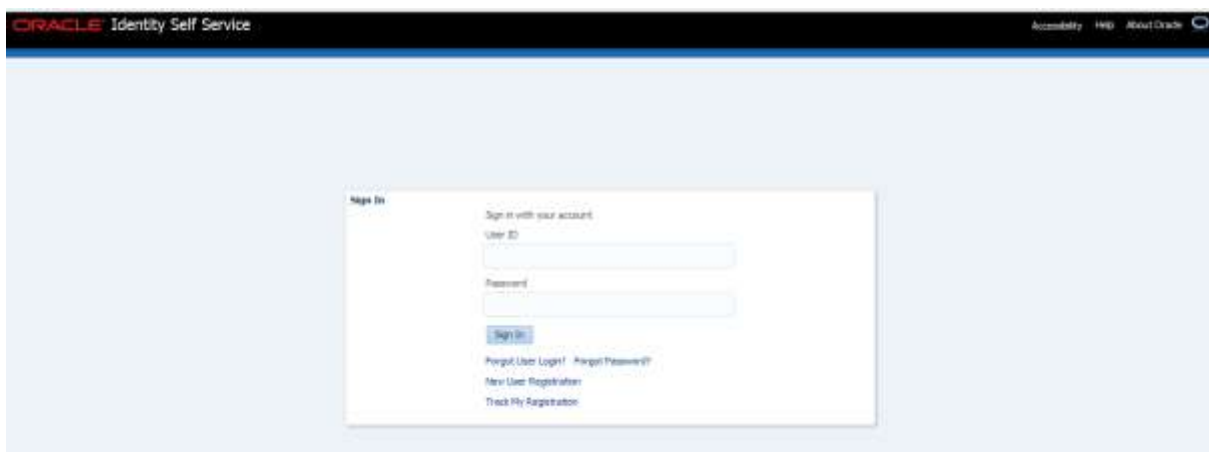
## Chapter 3: Oracle Identity Manager: Authentication Deep Dive

Let's explore the Authentication process within OIM in details and see what goes on behind the scenes.

### Authentication related scenarios/use cases

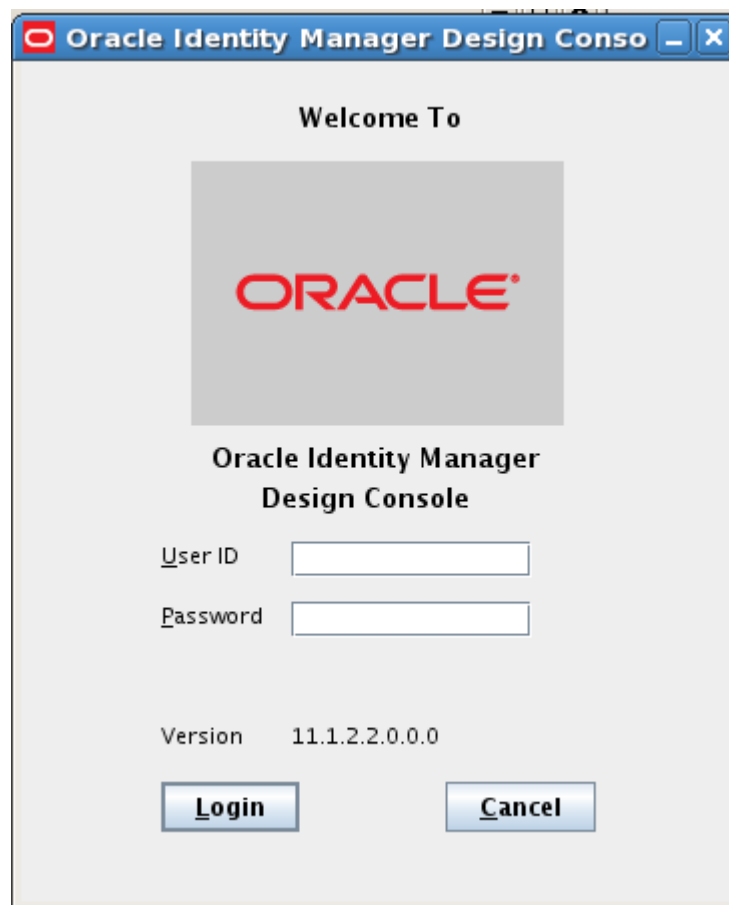
Generally speaking, there are three authentication scenarios which one typically encounters while using Oracle IDM

- **Logging into OIM Self Service or Administrative console** - All the authentication related business logic is executed on the server side



**OIM user challenge form**

- **Using the OIM Design Console** (standalone *thick client*) - There is an additional component which executes on the client side in order to initiate the authentication process. The authentication business logic is executed on the server side (just like in the above scenario)



OIM Design Console login prompt

- **Using OIM APIs** from a standalone environment (or from an environment external to the OIM application server/JVM) - the working logic is same as in *Design Console* based authentication

```
oimclient = new OIMClient(env);  
oimclient.login("xelsysadm", "<enter password>".toCharArray());
```

Invoking the login process using OIM APIs

**Q: What is common to all the authentication mechanisms listed in the above scenarios?**

**A:** The major components involved in the authentication process server as the common factor.

The components are as follows


- **Weblogic Authentication Providers** which drive Java EE Container based authentication ( Oracle IDM is certified on IBM Websphere as well, but *Weblogic* has being used as the primary example )
- **Java Authentication and Authorization Services (JAAS)** - It is used on the server side (*Authentication Providers*) as well as on the client end in a standalone mode (in case of design console or API based authentication).

**Note:** Both JAAS and Weblogic Authentication Providers have been covered in the previous chapters

## Authentication Process: Under the covers

### Logging into OIM Self Service or Administrative console

The **OIM Authentication Provider** is the authentication process driver. It is a part of the Weblogic security realm and is comprised of JAAS Login Modules which do all the heavy lifting ( actual authentication )

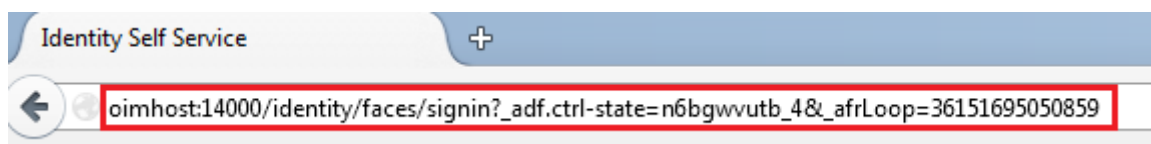


Name	Description	Version
DefaultAuthenticator	Weblogic Authentication Provider	1.0
OIMAuthenticationProvider	Provider that performs authentication thru the Oracle Identity Manager relational database	1.0
DefaultIdentityAsserter	Weblogic Identity Assertion provider	1.0

**OIM Authentication Provider**

### High Level process flow

- User accesses the OIM URL via a browser i.e. ***http://<hostname>:<port>/identity***



- The user is challenged via a login form



**Sign In**

Sign in with your account

User ID

Password

Sign In

[Forgot User Login?](#) [Forgot Password?](#)

[New User Registration](#)

[Track My Registration](#)

**OIM user challenge form**

- **Where does this form pop up from?** A basic understanding of the *Servlet Security model* is required in order to fully understand this.  
*For the sake of simplicity, it is enough to remember that the authentication mode is configured within the web.xml present in the WEB-INF folder of the WAR file. In this case it either the oracle.iam.console.identity.sysadmin.war or the oracle.iam.console.identity.self-service.war for the Administrative and Self Service console respectively*
- User enters his credentials on the OIM login page
- The Weblogic Security Framework **intercepts** this and **invokes** the OIM Authentication Provider

## Dissecting the OIM Authentication Provider

OIM Authentication Provider is nothing but a default OOTB *implementation* of the **Weblogic Security Service Provider Interface** (SSPI).

The class **OIMAuthenticationProvider**, implements the following SSPI interface - **weblogic.security.spi.AuthenticationProviderV2** and provides implementations for the following below mentioned methods (the method *signatures* have been provided below)

- `javax.security.auth.login.AppConfigurationEntry getLoginModuleConfiguration()`
- `javax.security.auth.login.AppConfigurationEntry getAssertionModuleConfiguration()`
- `weblogic.security.spi.PrincipalValidator getPrincipalValidator()`
- `weblogic.security.spi.IdentityAsserterV2 getIdentityAsserter()`

It also implements these methods which it inherits from the master interface **weblogic.security.spi.SecurityProvider**

- `void initialize(weblogic.management.security.ProviderMBean, weblogic.security.spi.SecurityServices)`
- `String getDescription()`
- `void shutdown()`

**Note:** The highlighted methods are the ones which are applicable in case of OIM Authentication provider

The **JAAS Login Module** within the OIM Authentication Provider processes end user credentials and attempts to authenticate the client as per custom implementation logic (this is against the OIM Database as per OOTB product implementation)

The above mentioned methods are leveraged in this process. This is how it happens at run time

- The **`void initialize(weblogic.management.security.ProviderMBean, weblogic.security.spi.SecurityServices)`** method is invoked by the container. This is used to pick up or *bootstrap* the required information by extracting it from the **`weblogic.management.security.ProviderMBean`** reference object
- The **`javax.security.auth.login.AppConfigurationEntry getLoginModuleConfiguration()`** method is invoked next and it returns a handle/reference to the JAAS Login Module implementation instance which will be utilized to perform the actual authentication.
- In case of a username/password credential combo, the **`OIMAuthLoginModule`** class is the JAAS login module implementation. It delegates the authentication to a specialized class which then interacts with *OIM Database* in a secure fashion to carry out the authentication process.  
*A thing to notice here is that a server side JAAS callback handler is created by the Weblogic Security Framework and passed into this login module. This is how the credentials (username/password) are actually propagated from remote client to the server.*
- The **`OIMAuthLoginModule`** is a custom implementation of the standard **`javax.security.auth.spi.LoginModule`** interface. The actual authentication is initiated from within the overridden **`boolean login()`** method.

**Note:** For more on JAAS refer to **Chapter 1**

- If the authentication is successful, the Login Module populates the Subject with the appropriate Principals and an authentication context is established

### Using the OIM Design Console (standalone *thick client*) or OIM APIs in a standalone manner

What happens when you try to log into the OIM Design console or authenticate using the **`login(String username, char[] password)`** method from the **`oracle.iam.platform.OIMClient`** class?

Well in this case, the client implementation leverages the JAAS module on its end and then passes on the authentication to the OIM Application Server (explained above)

So let's look at how the client module (design console or the API) initiates the authentication by breaking it into a sequence of steps

- The Java client/code invokes the **`login(String username, char[] password)`** method from the **`oracle.iam.platform.OIMClient`** class to initiate authentication (the same thing happens from the design console as well - it uses a *sub class* of the `oracle.iam.platform.OIMClient` class)
- In both the cases, the username and password is collected from the caller - *the user enters it in case of the Design Console and the Java client has to provide the credentials within the implementation code*

**Note:** See snap shots at the beginning of the post

- The above mentioned `login` method initiates the JAAS authentication process by leveraging the standard **`javax.security.auth.login.LoginContext`** class. It passes the credentials to the login module implementation in the form of **`javax.security.auth.callback.CallbackHandler`** which is the standard way for all custom JAAS Login Module implementations. To be specific, the **`XellerateCallbackHandler`** provides the call back handler implementation. This in turn invokes the JAAS Login Module.

A good question to be asked at this point of time is, ***which implementation of the Login Module is called and why?***

In the case of *design console*, the **`authwl.conf`** file is present in the **`config`** directory of the design console installation folder ( e.g. `/u01/Oracle/Middleware/Oracle_IDM1/designconsole/config`) and it contains the JAAS configuration details (i.e. the name of the Login Module and related info)

```
xellerate{
    weblogic.security.auth.login.UsernamePasswordLoginModule
    required debug=true;
};
```

#### Contents of the authwl.conf file

In case of ***OIM API***, it is mandatory for us to set the **`java.security.auth.login.config`** System property to point to the exact path of the JAAS configuration file - this is how it knows which Login Module implementation to load.

**Note:** Ideally speaking, the above mentioned system property is not hard coded. It's configured as static configuration e.g. server startup script by passing it as a property for the JVM - `Djava.security.auth.login.config=/u01/Oracle/Middleware/Oracle_IDM1/desginconsole/config/authwl.conf`

```

System.setProperty("XL.HomeDir", "<p>");

System.setProperty("java.security.policy",
    "/u01/Oracle/Middleware/Oracle_IDM1/designconsole/xl.policy");

System.setProperty("java.security.auth.login.config",
    "/u01/Oracle/Middleware/Oracle_IDM1/designconsole/authwl.conf");

System.setProperty("APPSERVER_TYPE", "wls");

System.setProperty("weblogic.Name", "oim_server1");

Hashtable<String, String> env = new Hashtable<String, String>();

env.put("java.naming.factory.initial",
    "weblogic.jndi.WLInitialContextFactory");

```

### Configuring the JAAS config file within the code

The implementation class in *authwl.conf* file ( see above snapshot ) i.e.

***weblogic.security.auth.login.UsernamePasswordLoginModule*** is provided by the Weblogic Server implementation OOTB. It is in charge of passing the credentials from the client to the remote Weblogic Server against which we are trying to authenticate. This is precisely why we provide the **T3 URL** (for Weblogic Application Server). Once the server side security framework is invoked, the process is the same as described above for the server authentication scenario.

**Note:** In case of the design console, the weblogic server URL is configured within a file called *xlconfig.xml* which is located in the same folder as the *authwl.conf* (see above)

In case of authentication via OIM APIs, the server URL can be configured in the code itself. This is only for testing purposes though. During deployment, the **XL.HomeDir** property needs to be set at the system level. This property needs to point to the design console installation. OIM automatically searches for the T3 URL info in *xlconfig.xml* file by peeking into the **<DESIGN\_CONSOLE\_HOME>/config** folder

```

java.naming.factory.initial
WebLogic: weblogic.jndi.WLInitialContextFactory
JBoss: org.jnp.interfaces.NamingContextFactory
WebSphere: com.ibm.websphere.naming.WsnInitialContextFactory

```

```

<Discovery>
  <CoreServer>
    <java.naming.provider.url>t3://localhost:14000/oim</java.naming.provider.url>
    <java.naming.factory.initial>weblogic.jndi.WLInitialContextFactory</java.naming.factory.initial>
  </CoreServer>
</Discovery>

```

T3 URL

```

<!--
Value of MultiCastAddress needs to same as OIM server
-->

```

### Excerpt from the xlconfig.xml file

- Post successful authentication, the remote weblogic server returns the authentication status to the client and it's up to the client to extract the authenticated **Subject** and the associated **Principals** from the response. This is done by the client side JAAS module which actually initiated the login process.

## The Signature Login process

Quite often, the **signature based login** feature in OIM is used in case the client needs to authenticate itself without the password. In 11g R2 onwards, the signature based login seems to be working from within the Weblogic container environment ONLY.

**Note:** *The signature login API might be discontinued in future*

Here is what happens during authentication via the Signature Login process

- In the absence of a password, the username is used as an authentication token (sort of)
- The username is first signed
- Then it is Base64 encoded
- Then the signed plus encoded string combo is used as the password and is passed to the OIM container (App server).
- From there onwards, the whole process is the same as described in the above modules.

**Note:** *A dedicated set of server logic processes the signed username*

## Conclusion

This chapter was an attempt to give a deep insight into different Authentication scenarios within Oracle IDM and the different standards which orchestrate behind the scenes.

## Closing Thoughts

With this, we have reached the end of this (mini) book which might prove to be the beginning of something new for you :-)

Thanks for Reading!