Oracle® Application Server Containers for J2EE

Security Guide

10*g* Release 2 (10.1.2)

B14013-02

July 2005

Oracle Application Server Containers for J2EE Security Guide, 10*g* Release 2 (10.1.2)

B14013-02

Primary Author: Elizabeth Hanes Perry

Contributing Author: Brian Wright

Contributor: Ganesh Kirti, Raymond Ng, Rachel Chan, Nithya Muralidharan, Moushmi Banerjee, Bill Bathurst, Tom Snyder, Jeff Trent, Cania Lee Chung

Contents

List of Examples

List of Figures

List of Tables

Preface

This manual discusses how to make effective use of the Oracle Application Server Containers for J2EE (OC4J) security features.

This preface contains these topics:

- Audience
- Documentation Accessibility
- Related Documentation
- Conventions

Audience

This manual is intended for experienced Java developers, deployers, and application managers who want to understand the security features of OC4J. It discusses the Oracle Application Server Java Authentication and Authorization Service (JAAS) Provider in detail, as well as discussing security implications of individual J2EE features, including EJBs, the J2EE Connector Architecture, SSL, and CSIv2.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at

`http://www.oracle.com/accessibility/`

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

Related Documentation

For more information, see these Oracle resources available from the Oracle Java Platform Group:

- *Oracle Application Server Containers for J2EE User's Guide*

  This book provides some overview and general information for OC4J; primer chapters for servlets, JSP pages, and EJBs; and general configuration and deployment instructions.

- *Oracle Application Server Containers for J2EE Stand Alone User's Guide*

  This version of the user's guide is specifically for the standalone version of OC4J, and is available when you download the standalone version from OTN. OC4J standalone is used in development environments, but not typically in production environments.

- *Oracle Application Server Containers for J2EE Servlet Developer's Guide*

  This book provides information for servlet developers regarding use of servlets and the servlet container in OC4J, including basic servlet development, use of JDBC and EJBs, building and deploying applications, and servlet and Web site configuration. Consideration is given to both OC4J in a standalone environment for development and OC4J in Oracle Application Server for production.

- *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide*

  This book provides information for JSP developers who want to run their pages in OC4J. It includes a general overview of JSP standards and programming considerations, as well as discussion of Oracle value-added features and steps for getting started in the OC4J environment.

- *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference*

  This book provides conceptual information and detailed syntax and usage information for tag libraries, JavaBeans, and other Java utilities provided with OC4J. There is also a summary of tag libraries from other Oracle product groups.

- *Oracle Application Server Containers for J2EE Services Guide*

  This book provides information about standards-based Java services supplied with OC4J, such as JTA, JNDI, JMS, JAAS, and the Oracle Application Server Java Object Cache.

Also available from the Oracle Java Platform group:

- *Oracle Database Java Developer's Guide*
- *Oracle Database JDBC Developer's Guide and Reference*

- *Oracle Database JPublisher User's Guide*

Available from the Oracle Application Server group:

- *Oracle Identity Management Concepts and Deployment Planning Guide*
- *Oracle Application Server Certificate Authority Administrator's Guide*
- *Oracle Application Server Single Sign-On Administrator's Guide*
- *Oracle Internet Directory Administrator's Guide*
- *Oracle Application Server Administrator's Guide*
- *Oracle Application Server Security Guide*
- *Oracle Application Server Performance Guide*
- *Oracle Enterprise Manager Concepts*
- *Oracle HTTP Server Administrator's Guide*
- *Oracle Application Server Globalization Guide*
- *Oracle Application Server Web Cache Administrator's Guide*
- *Oracle Application Server Web Services Developer's Guide*
- *Oracle Application Server Upgrade and Compatibility Guide*

Available from the Oracle JDeveloper group:

- Oracle JDeveloper online help
- Oracle JDeveloper documentation on the Oracle Technology Network:

  `http://www.oracle.com/technology/products/jdev/index.html`

Available from the Oracle Server Technologies group:

- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle XML API Reference*
- *Oracle Database Application Developer's Guide - Fundamentals*
- *PL/SQL Packages and Types Reference*
- *PL/SQL User's Guide and Reference*
- *Oracle Database SQL Reference*
- *Oracle Database Net Services Administrator's Guide*
- *Oracle Advanced Security Administrator's Guide*
- *Oracle Database Reference*

For additional information, see:

- The Sun Java and J2EE Web pages, especially the Java Authentication and Authorization Service (JAAS) Web site at :

  `http://java.sun.com/products/jaas/overview.html`

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1 Concepts

This chapter describes the following topics:

- Java 2 Security Model
- Principals
- Subjects
- Authentication and Authorization
- Secure Communications
- Developing Secure J2EE Applications

For a broader description of Oracle Application Server security in middle-tier environments that connect to the Internet, see the *Oracle Application Server Security Guide*. For information on Web services, see the *Oracle Application Server Web Services Developer's Guide*.

Java 2 Security Model

The Java 2 Security Model is fundamental to the Oracle Application Server Java Authentication and Authorization Service (JAAS) Provider.The Java 2 Security Model enables configuration of security at all levels of restriction. This provides developers and administrators with increased control over many aspects of enterprise applet, component, servlet, and application security. The Java 2 Security Model is capability-based and enables you to establish protection domains, and set security policies for these domains.

See Also:

- For a tutorial on Java 2 Security:

  `http://java.sun.com/docs/books/tutorial/security1.2/index.html`
- For full information on Java 2 Security:

  `http://java.sun.com/security`

Permissions

Permissions are the basis of the Java 2 Security Model. All Java classes (whether run locally or downloaded remotely) are subject to a configured security policy that defines the set of permissions available for those classes. Each permission represents a specific access to a particular resource. Table 1–1 identifies the elements that comprise a Java permission instance.

Table 1–1 Java Permission Instance Elements

Element	Description	Example
Class name	The permission class	`java.io.FilePermission`
Target	The target name (resource) to which this permission applies	Directory `/home/*`
Actions	The actions associated with this target	Read, write, and execute permissions on directory `/home/*`

Protection Domains

Each Java class, when loaded, is associated with a protection domain. Protection domains can be configured for all levels of restriction (from complete restriction on resources to full access to all resources). Each protection domain is assigned a group of permissions based on a configured security policy at Java virtual machine (JVM) startup.

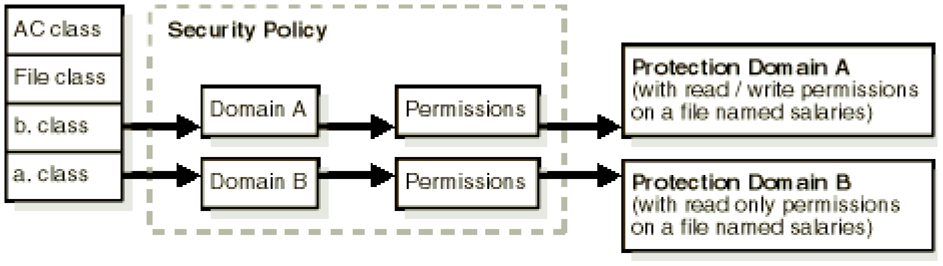
At runtime, the authorization check is done by stack introspection. This consists of reviewing the runtime stack and checking permissions based on the protection domains associated with the classes on the stack. This is typically triggered by a call to either:

- `SecurityManager.checkPermission()`
- `AccessController.checkPermission()`

The permission set in effect is defined as the intersection of all permission sets assigned to protection domains at the moment of the security check.

Figure 1–1 shows the basic model for authorization checking at runtime.

Figure 1–1 Java 2 Security Model



See Also:

- Sun Java documentation at:

  `http://java.sun.com/security/`

OracleAS JAAS Provider Permission Classes

Table 1–2 lists the permission classes furnished by the OracleAS JAAS Provider. These classes allow applications to control access to resources.

See Also:

- OracleAS JAAS Provider Javadoc for information about the classes discussed: *JAAS Provider API Reference*

Table 1–2 OracleAS JAAS Provider Permission Classes

Permission	Part of Package	Description
`AdminPermission`	`oracle.security.jazn.policy`	Represents the right to administer a permission (that is, grant or revoke another user's permission assignment).
`RoleAdminPermission`	`oracle.security.jazn.policy`	The grantee of this permission is granted the right to further grant/revoke the target role.
`JAZNPermission`	`oracle.security.jazn`	For authorization permissions. `JAZNPermission` contains a name (also called a target name), but no actions list; you either have or do not have the named permission.
`RealmPermission`	`oracle.security.jazn.realm`	Represents permission actions for a realm (such as `createRealm`, `dropRealm`, and so on). `RealmPermission` extends from `java.security.Permission`, and is used like any regular Java permission.

Principals

A *principal* is a specific identity, such as a user named `frank` or a role named `hr`. A principal is associated with a subject upon successful authentication to a computing service. Principals are instances of classes that implement the `java.security.Principal` interface. A principal class must define a namespace that contains a unique name for each instance of the class.

Subjects

A *subject* represents a grouping of related information for a single user of a computing service, such as a person, computer, or process. This related information includes the identities and security-related attributes of the subject (such as passwords and cryptographic keys).

Subjects can have multiple identities; principals represent identities in a subject. A subject becomes associated with a principal (user `frank`) upon successful authentication to a computing service—that is, the subject provides evidence (such as a password) to prove its identity.

Principals bind names to a subject. For example, a person subject, user `frank`, may have two principals:

- One binds the principal `frank doe` (name on his driver license) to the subject
- Another binds the identification principal `999-99-9999` (number on his student identification card) to the subject

Both principals refer to the same subject.

Subjects can also own security-related attributes (known as *credentials*). Sensitive credentials requiring special protection, such as private cryptographic keys, are stored in a private credential set. Credentials intended to be shared, such as public key certificates or Kerberos server tickets, are stored in a public credential set. Different permissions are required to access and modify different credential sets.

Subjects are represented by the `javax.security.auth.Subject` class.

To perform work as a particular subject, an application invokes the method `Subject.doAs(Subject, PrivilegedAction)` (or one of its variations). This method associates the subject with the `AccessControlContext` of the current thread and then executes the specified request.

Authentication and Authorization

Software security depends on two fundamental concepts: authentication and authorization.

- *Authentication* deals with the question "Who is trying to access my services?" In any system and application it is paramount to ensure that the identity of the entity or caller trying to access your application is identified in a secure manner. In a multitier application, the entity or caller can be a human user, a business application, a host, or one entity acting on behalf of (or impersonating) another entity.

  Authentication information is stored in a *user repository*. When a subject attempts to access a J2EE application, a *user manager* looks up the subject in the user repository and verifies the identity. A user repository can be a file or a directory server, depending on your environment. The Oracle Internet Directory is an example of a user repository.

  Although each J2EE application determines which user can use the application, it is the user manager that authenticates the user's identity using the user repository.

  OC4J supports several different authentication options; for details, see "Authentication Environments" on page 3-3.

- *Authorization* deals with the question "Who can access what services offered by which components?" For large-scale enterprises, where the access to various business-critical services and resources by millions of users need to be managed, it is important that a scalable authorization infrastructure be in place to deal with user and application provisioning. Unfortunately, in part due to the complex nature of authorization, this is also an area where confusion reigns and incompatible technologies and standards are prevalent.

  Developers specify authorization for subjects in the application J2EE and OC4J-specific deployment descriptors. These deployment descriptors indicate what roles are needed to access the different parts of the application. *Roles* are the identities that each application uses to indicate access rights to its different objects. The OC4J-specific deployment descriptors provide a mapping between the logical roles and the users and groups known by OC4J.

Secure Communications

To communicate securely, applications must satisfy the following goals:

- Secure communications—the data transmitted over the network cannot be intercepted, read, or altered by a third party. OC4J supports secure communications using the HTTP protocol over the Secure Sockets Layer.
- Network authentication—clients and servers must be able to authenticate themselves to one another over the network. This is achieved using digital certificates, single sign-on, or username/password combinations.
- Identity propagation—allowing one client to act as the agent of another client, using the original client identity.

Secure Sockets Layer

The Secure Sockets Layer (SSL) is the industry-standard point-to- point protocol which provides confidentiality through encryption, authentication and data integrity. Although SSL is used by many protocols, it is most important for OC4J when used with the HTTP browser protocol and in the AJP link between the OHS and OC4J processes.

Certificates

Applications need to transmit authentication and authorization information over the network. A *digital certificate*, as specified by the X.509 v3 standard, contains data establishing authentication and authorization information for a principal. A certificate contains:

- A public key, which is used in public key infrastructure (PKI) operations
- Identity information (for example, name, company, country, and so on)
- Optional digital rights which grant privileges to the owner of the certificate

Each certificate is digitally signed by a *trustpoint*. The trustpoint signing the certificate can be a certificate authority such as VeriSign, a corporation, or an individual.

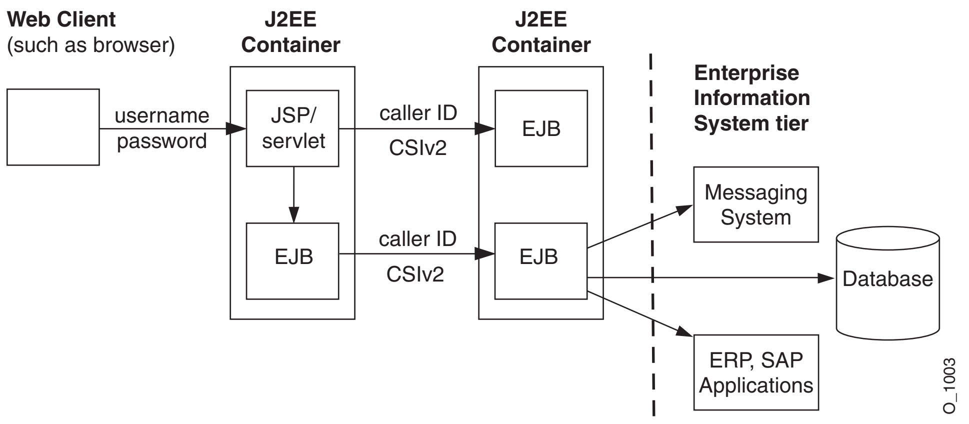
HTTPS

For convenience, this book uses "HTTPS" as shorthand when discussing HTTP running over SSL. Although there is an `https` URL prefix, there is no HTTPS protocol as such.

Identity Propagation

OC4J supports propagating the identity of principals from context to context. A Web client can establish its identity to a servlet; the servlet can then use that identity to communicate with other EJBs and servlets, as illustrated in Figure 1–2.

Figure 1–2 Identity Propagation Using CSIv2



Developing Secure J2EE Applications

J2EE software development is based on a develop-deploy-manage cycle. The OracleAS JAAS Provider plays an important role in the deploy-manage part of the cycle. The

OracleAS JAAS Provider is integrated with J2EE security. This means that developers can use a declarative security model instead of having to integrate security programmatically, unburdening the developer.

The following list summarizes the J2EE development cycle, with an emphasis on the tasks specific to developing secure applications.

1. The software developer creates Web components, enterprise beans, applets, servlets, and application clients.

   The OracleAS JAAS Provider offers programmatic interfaces, but the developer can create components without making use of those interfaces.

2. The application assembler takes these components and combines them into an Enterprise Archive (EAR) file.

   As part of this process, the application assembler specifies OracleAS JAAS Provider options appropriate to the environment.

3. The deployer installs the EAR into an instance of OC4J.

   As part of the deployment process, the deployer may map roles to users.

4. The system administrator maintains and manages the deployed application.

   This task includes creating and managing JAAS roles and users as required by the application customers.

2 Overview of JAAS in Oracle Application Server

This chapter further discusses the Oracle Application Server Java Authentication and Authorization Service (JAAS) Provider, first mentioned in the preceding chapter, in OC4J. The OracleAS JAAS Provider enables application developers to integrate authentication, authorization, and delegation services with their applications.

This chapter contains these topics:

The OracleAS JAAS Provider

The Oracle Application Server supports JAAS with the OracleAS JAAS Provider. The OracleAS JAAS Provider implements user authentication, authorization, and delegation services that developers can integrate into their application environments. Instead of devoting resources to developing these services, application developers can focus on the presentation and business logic of their applications.

Note: Some class and component names contain the word "JAZN", which is a shortened name for the OracleAS JAAS Provider.

The JAAS framework and the Java 2 Security model form the foundation of JAAS. The OracleAS JAAS Provider implements support for JAAS policies. Policies contain the rules (permissions) that authorize a user to use resources, such as reading a file. Using JAAS, services can authenticate and enforce access control upon resource users. The OracleAS JAAS Provider is easily integrated with J2SE and J2EE applications that use the Java 2 Security model.

Provider Types

The OC4J JAAS implementation supports two different provider types. Each provider type implements a repository for secure, centralized storage, retrieval, and administration of provider data. This data consists of realm (users and roles) and JAAS policy (permissions) information.

- XML-based provider

  The XML-based provider is used for lightweight storage of information in XML files. The XML-based provider stores user, realm, and policy information in an XML file, normally `jazn-data.xml`.

  Note: XML files are used as property and configuration files by both LDAP-based and XML-based provider types. However, only the XML-based provider stores users, realms, and policies in an XML file, `jazn-data.xml`.

- LDAP-based provider

  The LDAP-based provider is based on the Lightweight Directory Access Protocol (LDAP) for centralized storage of information in a directory. The LDAP-based provider stores user, realm, and policy information in the LDAP-based Oracle Internet Directory.

  Note: We recommend that you use the LDAP-based provider in a production environment; the XML-based provider is suitable for prototyping only.

What Is JAAS?

JAAS is a Java package that enables applications to authenticate and enforce access controls upon users. The OracleAS JAAS Provider is an implementation of the JAAS interface.

JAAS is designed to complement the existing code-based Java 2 security. JAAS implements a Java version of the standard Pluggable Authentication Module (PAM) framework. This enables an application to remain independent from the authentication service.

JAAS extends the access control architecture of the Java 2 Security Model to support principal-based authorization.

This section describes JAAS support for the following authentication, authorization, and user community (realm) features. The OracleAS JAAS Provider enhances some of these features.

- Login Module Authentication
- Roles
- Realms
- Applications
- Policies and Permissions

See Also:

- "JAAS Framework Features" on page 2-4 for information on how the OracleAS JAAS Provider enhances the JAAS framework to explicitly define key authorization, authentication, and user community (realm) features
- JAAS documentation at the following Web site for more specific discussions of key JAAS features:

  `http://java.sun.com/products/jaas/`

Login Module Authentication

To associate a principal (such as `frank`) with a subject, a client attempts to log in to an application. In login module authentication, the `LoginContext` class provides the basic methods used to authenticate subjects such as users, roles, or computing services. The `LoginContext` class consults configuration settings to determine whether the authentication modules (known as login modules) are configured for use with the particular application that the subject is attempting to access. Different login modules can be configured with different applications; furthermore, a single application can use multiple login modules.

Because the `LoginContext` separates the application code from the authentication services, you can plug a different login module into an application without affecting the application code.

Actual authentication is performed by the `LoginContext.login()` method. If authentication succeeds, then the authenticated subject can be retrieved by invoking the `LoginContext.getSubject()` method. The real authentication process can involve multiple login modules. The JAAS framework defines a two-phase authentication process to coordinate the login modules configured for an application.

After retrieving the subject from the `LoginContext`, the application then performs work as the subject by invoking the `Subject.doAs()` or `Subject.doAsPrivileged()` method.

Roles

The JAAS framework does not explicitly define roles or groups. Instead, roles or groups are implemented as concrete classes that use the `java.security.Principal` interface.

The JAAS framework does not define how to support the role-based access control (RBAC) role hierarchy, in which you can grant a role to a role.

Realms

The JAAS framework does not explicitly define user communities. However, the J2EE reference implementation (RI) defines a similar concept of user communities called *realms*. A realm provides access to users and roles (groups) and optionally provides administrative functionality. A user community instance is essentially a realm that is maintained internally by the authorization system. The J2EE RI Realm API supports user-defined realms through subclassing.

Applications

The JAAS framework does not explicitly define an application or subsystem for partitioning authorization rules.

Policies and Permissions

A *policy* is a repository of JAAS authorization rules. The policy includes grants of *permissions* to principals, thus answering the question: "Given a grantee, what are the granted permissions of the grantee?"

Policy information is supplied by the OracleAS JAAS Provider. The JAAS framework does not define an administrative API for policy administration. The administrative API provided by the OracleAS JAAS Provider is an Oracle extension.

Table 2–1 describes the Sun Microsystems implementation of policy file parameters.

Table 2–1 Policy File Parameters

Parameter	Defined As	Example
Subject	One or more principal(s)	duke
Codesource	*`codebase`*, *`signer`*	`http://www.example.com, mysigner`

XML-Based Example

The JAAS XML-based provider can store policy data in the file `jazn-data.xml`. In the following example, a segment of the `jazn-data.xml` file grants the `admin` principal permission to log in.

```
<jazn-policy>
  <grant>
   <grantee>
     <principals>
      <principal>
        <class>oracle.security.jazn.samples.SampleUser</class>
        <name>admin</name>
      </principal>
    </principals>
   </grantee>
   <permissions>
     <permission>
       <class>oracle.j2ee.server.rmi.RMIPermission</class>
       <name>login</name>
     </permission>
   </permissions>
  </grant>
</jazn-policy>
```

See Also:

- "Sample: jazn-data.xml Configuration" on page A-1 to view a complete `jazn-data.xml` file

JAAS Framework Features

Table 2–2 contains the JAAS framework features implemented by the OracleAS JAAS Provider.

Table 2–2 OracleAS JAAS Provider Features

Feature	Description	See Also
Authentication	■ Integrates with Oracle Application Server Single Sign-On for login authentication in J2EE application environments. ■ Supplies an out-of-the-box `RealmLoginModule` class for non-OracleAS Single Sign-On environments, such as OracleAS Core or Java Edition. ■ Supports any JAAS-compliant custom `LoginModule`.	Chapter 3, "Understanding OC4J Security"
Declarative Model	■ Integrates J2EE deployment descriptors, such as. `web.xml`, with JAAS security. ■ Supports programmatic model as well.	Chapter 4, "Overall Security Configuration"
Authorization	■ Provides centralized role-based access control, including support for hierarchical roles.	"Role-Based Access Control (RBAC)" on page 2-8
Realms	■ Organizes users and roles (groups) around user communities. An Oracle API package (`oracle.security.jazn.realm`) is provided to support user and role management. This API includes a `RealmPrincipal` interface that extends from `java.security.Principal` and associates a realm with users and roles.	"Realms" on page 2-3
Management	■ Manages settings and data using command-line tool (Admintool) or Oracle Enterprise Manager 10*g*. ■ Supports a centrally managed provider type with Oracle Internet Directory.	
`JAZNUserManager`	■ Provides an implementation of the OC4J `UserManager` that integrates with both the XML-based and the LDAP-based providers.	"JAZNUserManager" on page 3-3

User Managers

OC4J security employs a user manager to authenticate and authorize users and groups that attempt to access a J2EE application. You base your choice of user manager on performance and security needs.

All `UserManager` classes implement the `com.evermind.security.UserManager` interface. `UserManager` classes manage users, groups, and passwords through methods such as `createUser()`, `getUser()`, and `getGroup()`.

See Also:

- OracleAS JAAS Provider Javadoc for further information: *JAAS Provider API Reference*

OC4J provides two predefined user managers, `JAZNUserManager` and `XMLUserManager`. `JAZNUserManager` supports both XML-based and LDAP-based providers. We recommend using `JAZNUserManager` because it is based on the JAAS specification and is integrated with Oracle Application Server Single Sign-On and Oracle Internet Directory. `JAZNUserManager` is the default security provider, because it offers powerful and flexible security control. Customers can also supply their own classes that implement the `UserManager` interface, although this will be deprecated at a future release.

See Also:

- For a discussion of creating a custom `UserManager`:

  `http://www.oracle.com/technology/sample_code/tech/xml/xmlnews/News_Security.html`.

Table 2–3 lists the user managers provided by OC4J.

Table 2–3 OC4J User Managers and Repositories

User Manager Class	User Repository
`oracle.security.jazn.oc4j.JAZNUserManager`	Several types: ■ Using the XML-based provider, `jazn-data.xml` ■ Using the LDAP-based provider, Oracle Internet Directory ■ Using a third-party LDAP provider
`com.evermind.server.XMLUserManager`	The `principals.xml` file
Custom user manager	Customized user repository

See "Specifying a User Manager in orion-application.xml" on page 4-6 for directions on how to define the default `UserManager` for all applications or a single `UserManager` for a specific application.

The following sections describe the JAZN and XML user managers:

- Using JAZNUserManager
- Using XMLUserManager

Using JAZNUserManager

The `JAZNUserManager` class is the default user manager. The primary purpose of the `JAZNUserManager` class is to leverage the OracleAS JAAS Provider as the security infrastructure for OC4J.

There are two JAAS providers supplied with OC4J security: XML-based and LDAP-based.

- The XML-based provider is a fast, lightweight implementation of the JAAS Provider API. This provider type uses XML to store user names and encrypted passwords. The user repository is stored in the `jazn-data.xml` file, in a location specified in the `jazn.xml` file. For details, see Chapter 8, "Configuring the XML-Based Provider".

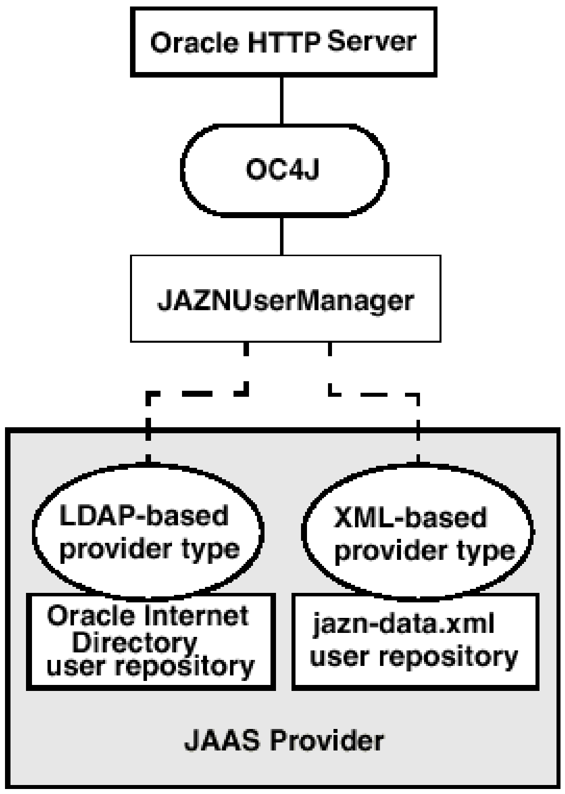
  Select JAZN-XML as the user manager in the Enterprise Manager. Configure its users, roles, and groups using the JAZN Admintool.

- The LDAP-based provider is scalable, secure, enterprise-ready, and integrated with OracleAS Single Sign-On. The LDAP-based provider is the only OracleAS JAAS Provider that supports OracleAS Single Sign-On.

  Select JAZN-LDAP as the user manager in the Enterprise Manager. Configure its users and groups using the Oracle Delegated Administration Services from Oracle Internet Directory. The user repository is an Oracle Internet Directory instance, which requires that the application server instance be associated with an infrastructure. If it the server is not associated with an Oracle Internet Directory instance, then the LDAP-based provider is not a security option.For information on configuring the LDAP-based provider, see Chapter 7, "Configuring the LDAP-Based Provider".

Figure 2–1 shows the two different JAAS providers supplied with OC4J.

Figure 2–1 OC4J Security Architecture under the JAZNUserManager Class



Using XMLUserManager

The `XMLUserManager` class is a simple user manager that manages users, groups, and roles in an XML-based system. It stores user passwords in the clear, and therefore is not as secure as the `JAZNUserManager`. All `XMLUserManager` configuration information is stored in the `principals.xml` file, which is the user repository for the `XMLUserManager` class.

Note: The `XMLUserManager` class is supported for backward compatibility only, and will be desupported in a forthcoming release. Oracle strongly recommends that you use one of the OracleAS JAAS Provider types.

Capability Model of Access Control

The *capability model* is a method for organizing authorization information. The OracleAS JAAS Provider is based on the Java 2 Security Model, which uses the capability model to control access to permissions. With the capability model, authorization is associated with the principal (a user named `frank` in the following example). Table 2–4 shows the permissions that user `frank` is authorized to use:

Table 2–4 User Permissions

User	File Permissions
`frank`	Read and write permissions on a file named `salaries.txt` in the `/home/user` directory

When user `frank` logs in and is successfully authenticated, the permissions described in Table 2–4 are retrieved from the OracleAS JAAS Provider (whether the LDAP- based Oracle Internet Directory or XML-based provider) and granted to user `frank`. User `frank` is then free to execute the actions permitted by these permissions.

Role-Based Access Control (RBAC)

RBAC enables you to assign permissions to roles. You grant users permissions by making them members of appropriate roles. Support for RBAC is a key JAAS feature. This section describes the following RBAC features:

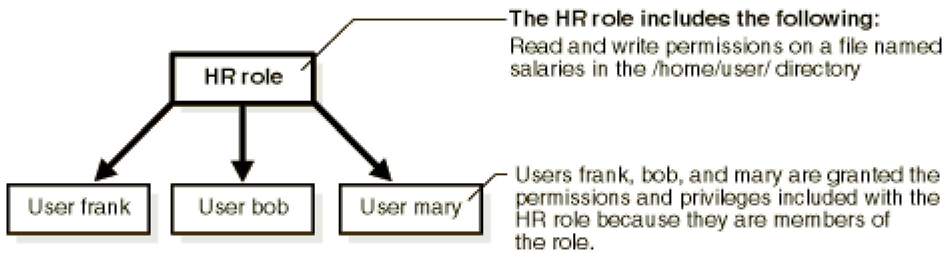
- Role Hierarchy
- Role Activation

Role Hierarchy

RBAC simplifies the management problems created by direct assignment of permissions to users. Assigning permissions directly to multiple users is potentially a major management task. If multiple users no longer require access to a specific permission, you must individually remove that permission from each user.

Instead of directly assigning permissions to users, permissions are assigned to a role, and users are granted their permissions by being made members of that role. Multiple roles can be granted to a user. A role can also be granted to another role, thus forming a *role hierarchy* that provides administrators with a tool to model enterprise security policies. Figure 2–2 provides an example of role-based access control.

Figure 2–2 Role-Based Access Control


The HR role includes the following:
Read and write permissions on a file named salaries in the /home/user/ directory
HR role
User frank
User bob
User mary
Users frank, bob, and mary are granted the permissions and privileges included with the HR role because they are members of the role.

When a user's responsibilities change (for example, through a promotion), the user's authorization information is easily updated by assigning a different role to the user instead of a massive update of access control lists containing entries for that individual user.

For example, if multiple users no longer require write permissions on a file named `salaries` in the `/home/user` directory, those privileges are removed from the `HR` role. All members of the `HR` role then have their permissions and privileges automatically updated.

Role Activation

A user is typically granted multiple roles. However, not all roles are enabled by default. An application can selectively enable the required roles to accomplish a specific task in a user session with the `run-as` security identity and `Subject.doAS()` method. Selectively enabling roles upholds the principle of least privilege—the application is not enabling permissions or privileges unnecessary for the task. This limits the damage that can potentially result from an accident or error.

Changes Since Release 9.0.4

- The correct setting for the environment variable controlling loading of dynamic libraries (for example, `LD_LIBRARY_PATH` in Solaris) is now *`ORACLE_HOME`*`/lib32` instead of *`ORACLE_HOME`*`/lib`). Table 2–5 shows the complete settings.

Table 2–5 Dynamic Library Path Settings

Operating System	Variable	Value
Solaris	`LD_LIBRARY_PATH`	*`ORACLE_HOME`*`/lib32`
	`LD_LIBRARY_PATH_64`	*`ORACLE_HOME`*`/lib`
HP/UX	`SHLIB_PATH`	*`ORACLE_HOME`*`/lib32`
	`LD_LIBRARY_PATH`	*`ORACLE_HOME`*`/lib`
AIX 32-bit applications	`LIBPATH`	*`ORACLE_HOME`*`/lib32`
	`LD_LIBRARY_PATH`	Null
AIX 64-bit applications	`LIBPATH`	*`ORACLE_HOME`*`/lib`
	`LD_LIBRARY_PATH`	Null
Windows	Not applicable	Not applicable

- The Java Development Kit 1.3 default installation does not include JAAS support. To use JAAS with JDK1.3, you must download JAAS 1.0_01 from the Sun Web site `http://java.sun.com/products/jaas/index-10.html` and follow the installation and deployment instructions.

> **Note:** When you develop applications using JDK 1.3, you should be aware of a JDK class loader issue. The class loader can locate custom login modules only if you deploy the JAR containing them as a standard extension in the *`ORACLE_HOME`*`/jre/lib/ext` directory. This problem will be fixed at the next release.

- At this release, Oracle supports third-party LDAP providers. See Chapter 9, "Configuring External LDAP Providers", for details.
- At this release, Oracle supplies a default file (`jazn.security.props`) in the directory *`ORACLE_HOME`*`/j2ee/home/startup` that specifies the OracleAS JAAS provider to be used for JAAS login configuration and policy. Note that these properties are set by default during OC4J startup, so in most circumstances you do not need to worry about setting these properties. For details, see "Specifying an Alternate Policy Provider (Optional)" on page 4-4.
- Custom `UserManager` classes are still supported at this release, but will be deprecated at a future release.

- The file `principals.xml` will no longer be supported at a future release. We strongly encourage you to migrate your existing applications from using `principals.xml` to using `JAZNUserManager`. For instructions, see "Migrating Principals from the principals.xml File" on page 8-5.
- The interface for retrieving the SSL client certificate has changed. You now use the following:

  ```
  servletRequest.getAttribute("javax.servlet.request.X509Certificate");
  ```

  Instead of:

  ```
  servletRequest.getAttribute("javax.security.cert.X509certificate");
  ```

3

Understanding OC4J Security

This chapter describes security issues affecting J2EE applications in OC4J.

This chapter contains these topics:

Introduction

The following are components of the OC4J security architecture:

- The OracleAS JAAS Provider, which provides support for storage, retrieval, and administration of realm information (users and roles) and policy information (permissions). The OracleAS JAAS Provider supports two possible repositories or *provider types*:
  - XML-based provider type
  - LDAP-based Oracle Internet Directory (available only with Oracle Application Server Infrastructure installation)
- JAAS login modules, such as the `RealmLoginModule`, third-party login modules, and custom login modules

See Also:

- "Provider Types" on page 2-2 for further information about provider types
- *Oracle Application Server Installation Guide* for information on installing Oracle Internet Directory

Security Considerations During Development and Deployment

The OracleAS JAAS Provider is designed to work with the J2EE declarative security model. This declarative model requires little or no programming to use JAAS security in your application. Instead, most security decisions are made as part of the deployment process, making it easy to make changes without requiring re-coding. If the declarative model is not sufficient, the OracleAS JAAS Provider also supports programmatic security in the same manner that JAAS is used in any J2SE environment.

Development

If your application relies on the declarative security model (where J2EE security roles are defined in deployment descriptors, such as `web.xml`), the developer must determine if the application uses application-specific roles. If so, the developer must define these roles so that they can be mapped to the J2EE logical roles during the deployment phase.

Deployment

Using the declarative security model, the deployer must make the following security-related decisions:

- Determine the J2EE logical roles that are assumed in the application, then define these roles in the deployment descriptors. For example, an HR application may assume that the J2EE logical role `hr_manager` is running the application; the deployer must define that role.
- Determine the authorization constraints that apply to these roles and define them in the deployment descriptors. For web modules, these constraints typically apply to URL patterns as defined in the J2EE specification. EJB modules typically have constraints at the EJB-method level.
- Decide whether to use an XML flat file or Oracle Internet Directory (LDAP) as the repository for the OracleAS JAAS Provider. This also determines which provider, XML-based or LDAP-based, and user manager the application uses.
- Map the security roles (including the application-specific roles, if they exist) to users and groups defined by the OC4J user manager (for instance, `JAZNUserManager`). For example, the J2EE logical role called `hr_manager` may be mapped to a given set of users defined by the OC4J user manager.

See Also:

- Chapter 6, "Security Considerations During Application Deployment" for information on making and implementing these decisions
- *Oracle Application Server Containers for J2EE User's Guide* for a full discussion of deployment

OC4J and the OracleAS JAAS Provider

OC4J is a J2EE container that accepts HTTP and RMI client connections. These connections permit access to servlets, JavaServer Pages, and Enterprise JavaBeans (EJBs).

J2EE containers separate business logic from resource and lifecycle management. This enables developers to focus on writing business logic, rather than writing enterprise infrastructure. For example, Java servlets simplify Web development by providing an infrastructure for component, communication, and session management in a Web container integrated with a Web server.

OC4J Integration

The OracleAS JAAS Provider is integrated with OC4J and with OracleAS Single Sign-On to enhance application security. This integration provides the following benefits:

- `run-as` identity support, delegation support (from servlet to Enterprise JavaBeans)
- Full support for OracleAS Single Sign-On
- Support for custom login modules

JAZNUserManager

The OracleAS JAAS Provider is supported through `JAZNUserManager`. `JAZNUserManager`, an implementation of the OC4J `UserManager` interface, supports the following features:

- Secure storage of obfuscated passwords
- Full role-based access control (RBAC), including hierarchical roles
- Full support for the Java 2 permission model and JAAS
- Secure implementation based on the Java 2 permission model, allowing non-trusted (or partially trusted) code to run in the same JVM as the OracleAS JAAS Provider
- OracleAS Single Sign-On integration with OC4J
- `RealmLoginModule` integration in non-OracleAS Single Sign-On environments
- Support for custom JAAS login modules
- Identity propagation

Authentication Environments

The OracleAS JAAS Provider integrates with several different login authentication environments in a J2EE application.

- **OracleAS Single Sign-On**

  Uses OracleAS Single Sign-On to authenticate logins.
- **SSL**
  - Uses Secure Socket Layers for client certificate-based authentication.
  - Uses a login module (for example, `RealmLoginModule`) to authenticate logins.
- **Basic Authentication**
  - Prompts user directly for user name and password, without going through OracleAS Single Sign-On.
  - Uses a login module (for example, `RealmLoginModule`) to authenticate logins.
- **Form-based Authentication**

  When the user attempts to access a protected resource, OC4J checks whether the user has already been authenticated. If not, OC4J displays an application-specific login prompt for user name and password.

- **Digest Authentication**

  With the digest mechanism (a standard but optional feature of J2EE), the credentials that a client presents to authenticate itself consist of an MD5 digest. This is transmitted in the request message.

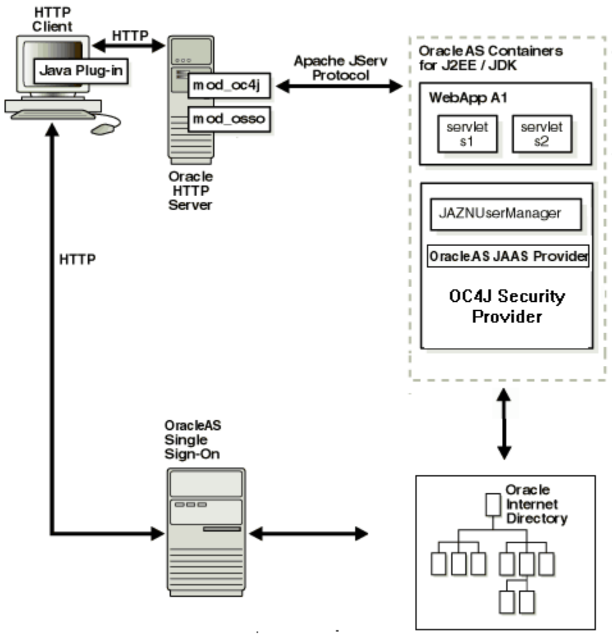
The following sections discuss how the OracleAS JAAS Provider integrates with each of these authentication types.

Enabling OracleAS Single Sign-On in J2EE Applications

OracleAS Single Sign-On lets a user access multiple applications with a single set of login credentials. Figure 3–1 shows JAAS integration in an application running in an OracleAS Single Sign-On-enabled J2EE environment.

Note: Alternatively, there is a single sign-on implementation specifically for the OC4J environment. See "Lightweight J2EE Single Sign-On" on page 3-9.

Figure 3–1 OracleAS Single Sign-On and J2EE Environments



OracleAS Single Sign-On-Enabled J2EE Environments: Typical Scenario

This section describes the responsibilities of Oracle components when an HTTP client request is initiated in an OracleAS Single Sign-On-enabled J2EE environment.

1. An HTTP client attempts to access a Web application, WebApp A1, hosted by OC4J (the Web container for executing servlets). Oracle HTTP Server (using an Apache listener) handles the request.
2. `mod_osso`/Oracle HTTP Server receives the request then:
   a. Determines that WebApp A1 application requires Web-based OracleAS Single Sign-On for authenticating HTTP clients.

   b. Redirects the HTTP client request to the Web-based OracleAS Single Sign-On (because it has not yet been authenticated).

3. The HTTP client is authenticated by OracleAS Single Sign-On through a user name and password or through a user certificate. OracleAS Single Sign-On then:

   a. Validates the user's stored login credentials.

   b. Sets the OracleAS Single Sign-On cookie (including the user's distinguished name and realm).

   c. Redirects back to the WebApp A1 application (in OC4J).

4. The OracleAS JAAS Provider retrieves the OracleAS Single Sign-On user.

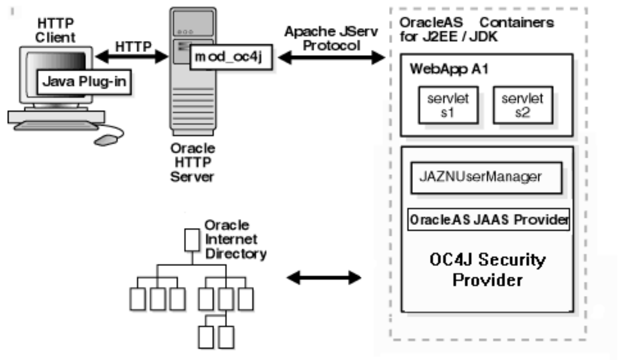
See Also:

- *Oracle Application Server Single Sign-On Administrator's Guide*, for full details on OracleAS Single Sign-On

Integrating the OracleAS JAAS Provider with SSL-Enabled Applications

SSL is an industry standard protocol for managing the security of message transmission on the Internet. Figure 3–2 shows JAAS integration in an application running in an SSL-enabled J2EE environment.

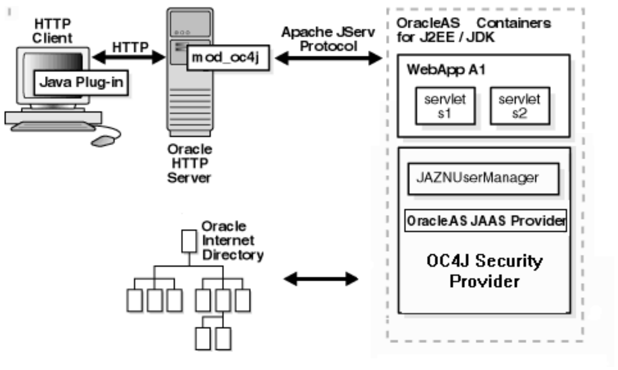
Figure 3–2 Oracle Component Integration in SSL-Enabled J2EE Environments



Integrating the OracleAS JAAS Provider with Basic Authentication

Basic authentication bypasses OracleAS Single Sign-On. Figure 3–3 shows specific JAAS integration in an application configured for Basic authentication in a J2EE environment.

Figure 3–3 Oracle Component Integration in J2EE Environment



Basic Authentication J2EE Environments: Typical Scenario

This section describes the responsibilities of Oracle components when an HTTP client request is initiated in a J2EE environment configured for Basic authentication. In this environment, OracleAS Single Sign-On is not used. A login module (for example, `RealmLoginModule`) is used.

1. An HTTP client attempts to access a Web application (named WebApp A1) hosted by OC4J (the Web container for executing servlets).
2. OC4J invokes the `RealmLoginModule` whenever user credentials are required. For example, when a request hits a protected page, OC4J will ask the OracleAS JAAS Provider to authenticate the user, then the `RealmLoginModule` will be invoked to authenticate the user, using the credentials sent by the user via the browser over HTTP.
3. The OracleAS JAAS Provider retrieves the user.

See Also:

- Sun Java documentation for more information on J2EE:

  `http://java.sun.com/j2ee/`

Authentication in the J2EE Environment

Authentication is the process of verifying the identity of a user in a computing system, often as a prerequisite to granting access to resources in a system. In Oracle Application Server, authentication in the J2EE environment is performed by the following:

- OracleAS Single Sign-On (for OracleAS Single Sign-On environments) or the OracleAS JAAS Provider `RealmLoginModule` or other login module (for non-OracleAS Single Sign-On environments)

  Before HTTP requests can be dispatched to the target servlet, the `JAZNUserManager` gets the authenticated user information (set by `mod_osso`) from the HTTP request object and sets the JAAS subject in OC4J.

Note: There is a separate single sign-on implementation for standalone OC4J. See "Lightweight J2EE Single Sign-On" on page 3-9.

- One of the following:

  - `JAZNUserManager`
  - `XMLUserManager`
  - A developer-supplied `UserManager`

> **Note:** Developer-supplied `UserManager` is deprecated and will be desupported in a future release.

Running with an Authenticated Identity

You can choose to configure the `JAZNUserManager` so that a filter enables the target servlet to run with the permissions and roles associated with an authenticated identity or run-as identity. To do this, configure the `<jazn-web-app>` element.

> **See Also:**
>
> - "JAZNUserManager" on page 3-3 for further information on options and configuration of the `JAZNUserManager` filter, including the `<jazn-web-app>` element

Retrieving Authentication Information

The following `javax.servlet.HttpServletRequest` APIs retrieve authentication information within the servlet:

- `getRemoteUser()` for the authenticated user name
- `getAuthType()` for the authentication scheme
- `getUserPrincipal()` for the authenticated principal object

> **Note:** The returned principal is an instance of the interface `oracle.j2ee.security.User`, which extends the `java.security.Principal` interface.

- `getAttribute("javax.servlet.request.X509certificate")` for the SSL client certificate

Authorization in the J2EE Environment

Authorization is the process of granting permissions and privileges to an authenticated user. This section discusses authorization within servlets.

If the servlet is configured to permit `doAs()`, the `JAZNUserManager` invokes an authenticated target servlet within a `Subject.doAs()` block to enable JAAS-based authorization in the target servlets.

Authorization is achieved through the following:

- `JAZNUserManager`
- Methods based on JAAS authorization:
  - `Servlet.service()` in the servlet
  - `Subject.doAs()` and `Subject.doAsPrivileged()` in the client
  - `SecurityManager.checkPermission()` in the server

See Also:

- Configuring J2EE Authorization on page 4-11

Security Role Mapping

Two distinct role types are available to application developers creating secure applications in J2EE environments: J2EE roles and JAAS roles. When these role types are mapped together using OC4J group mappings, users can access an application with a defined set of role permissions for as long as the user is mapped to this role.

This section describes these role types and how they are mapped together.

- J2EE Security Roles
- Deployment Roles and Users
- OC4J Group Mapping to J2EE Security Roles

J2EE Security Roles

The J2EE development environment includes a portable security roles feature defined in the `web.xml` file for servlets and JavaServer Pages (JSP). Security roles define a set of resource access permissions for an application. Associating a principal (in this case, a JAAS user) with a security role assigns the defined access permissions to that principal for as long as they are mapped to the role. For example, an application defines a security role called `sr_developer`:

```
<security-role>
     <role-name>sr_developer</role-name>
</security-role>
```

You also define the access permissions for the `sr_developer` role.

```
 <security-constraint>
    <web-resource-collection>
      <web-resource-name>access to the entire application</web-resource-name>
      <url-pattern>/*</url-pattern>
    </web-resource-collection>
        <!-- authorization -->
    <auth-constraint>
      <role-name>sr_developer</role-name>
    </auth-constraint>
  </security-constraint>
```

Deployment Roles and Users

JAAS roles and users are defined depending on the provider type, LDAP-based or XML-based.

For example, with the XML-based provider type, `developer` is listed as a role in the `jazn-data.xml` file:

```
<role>
  <name>developer</name>
  <members>
    <member>
      <type>user<type>
      <name>john<name>
    </member>
  </members>
</role>
```

OC4J Group Mapping to J2EE Security Roles

OC4J enables you to map portable J2EE security roles defined in the J2EE `web.xml` file to groups in an `orion-application.xml` file.

The roles and users defined in your provider environment are mapped to the OC4J `developer` group role in the `orion-application.xml` file.

For example, the `sr_developer` security role is mapped to the group named `developer`.

```
<security-role-mapping name="sr_developer">
   <group name="developer" />
</security-role-mapping>
```

Notice that a `<group>` entry in a `<security-role-mapping>` element corresponds to a role in the OracleAS JAAS Provider. Therefore, this association permits the `developer` group to access the resources allowed for the `sr_developer` security role.

In this paradigm, the user `john` is listed as a member of the `developer` role. Because the `developer` group is mapped to the J2EE security role `sr_developer` in the `orion-application.xml` file, `john` has access to the application resources defined by the `sr_developer` role.

Lightweight J2EE Single Sign-On

This section describes the use of a lightweight single sign-on (SSO) alternative for OC4J. Oracle Application Server already offers an SSO component, known as OracleAS Single Sign-On and discussed earlier in this chapter, but this requires the installation of several components. For OC4J, a simpler alternative was required.

This chapter contains the following topics:

- Introduction to Lightweight J2EE Single Sign-On
- Configuring Lightweight J2EE Single Sign-On
- Enabling Lightweight J2EE Single Sign-On

Introduction to Lightweight J2EE Single Sign-On

After a client has been authenticated in a given Web application, SSO allows the current security credential to be applied to additional Web applications within the same J2EE application (EAR file). OracleAS Single Sign-On is available for this functionality in a full Oracle Application Server environment (as described in "Enabling OracleAS Single Sign-On in J2EE Applications" on page 3-4), and now there is also a lighter-weight SSO alternative for use with OC4J.

Security credentials for a Web application are generally stored within HTTP client sessions. However, because the servlet specification dictates that an HTTP session can extend only across the scope of a single Web application, OC4J must use a component outside the session to manage J2EE SSO across Web applications. The chosen component for this implementation of lightweight J2EE Single Sign-On is a stateful session EJB.

When J2EE Single Sign-On is enabled, OC4J creates the stateful session bean automatically, including the home interface, bean interface, and implementation class. Once a user is successfully authorized for a Web application participating in J2EE Single Sign-On, OC4J registers the user with the bean.

OC4J also creates a J2EE SSO cookie. The cookie indicates to Web applications that there is an authenticated user for J2EE Single Sign-On.

Also be aware of the following:

- The lightweight J2EE SSO implementation works across distributed environments, assuming EJB replication is enabled for the J2EE SSO bean as shown in "Contents of Specialized orion-ejb-jar.xml File" on page 3-11.
- The J2EE SSO implementation works with standalone OC4J. It will also work in a full Oracle Application Server (OPMN-managed) environment if all of the following are true:
  - Session replication is enabled for each Web application involved, through the `<distributable/>` subelement of `<web-app>` in the `web.xml` file.
  - EJB replication is enabled for the J2EE SSO bean.
  - You use only a single OC4J cluster island.
- The J2EE SSO implementation has a small performance impact.
- If a user who has been looked up has sufficient privilege to access the restricted resource, then OC4J displays the resource. Otherwise, a form authentication page is displayed, allowing a user with sufficient privilege to be specified for the login.
- If the HTTP session is invalidated, the user name is no longer set. The next attempted access of the resource will force another login through the form authentication page.

Configuring Lightweight J2EE Single Sign-On

The 10.1.2 implementation of lightweight J2EE Single Sign-On requires a specialized JAR file, named `OC4JINTERNAL_httpSession.jar`, to be included in your EAR file. This JAR file must contain the following XML files, each with specific contents as described immediately below:

- `/META-INF/ejb-jar.xml`
- `/META-INF/orion-ejb-jar.xml`

Contents of Specialized ejb-jar.xml File

The `ejb-jar.xml` file must contain the following:

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
<ejb-jar>
   <display-name>stateful, ejb-jar:</display-name>
   <enterprise-beans>
      <session>
         <display-name>WebAppsClusterBean</display-name>
         <ejb-name>WebAppsClusterBean</ejb-name>
         <home>
            oracle.oc4j.server.ejb.httpSession.HttpSessionSharingSFSBHome
         </home>
         <remote>
            oracle.oc4j.server.ejb.httpSession.HttpSessionSharingSFSB
         </remote>
         <ejb-class>
            oracle.oc4j.server.ejb.httpSession.HttpSessionSharingSFSBBean
         </ejb-class>
         <session-type>Stateful</session-type>
         <transaction-type>Container</transaction-type>
      </session>
   </enterprise-beans>
   <assembly-descriptor>
      <container-transaction>
         <method>
            <ejb-name>WebAppsClusterBean</ejb-name>
            <method-name>*</method-name>
         </method>
         <trans-attribute>Supports</trans-attribute>
      </container-transaction>
      <security-role>
         <role-name>users</role-name>
      </security-role>
   </assembly-descriptor>
</ejb-jar>
```

Note: The `<display-name>` elements are not required but are by convention. All other elements are required with settings as shown.

Contents of Specialized orion-ejb-jar.xml File

For a non-distributed environment, the `orion-ejb-jar.xml` file must contain the following:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE orion-ejb-jar PUBLIC "-//Evermind//DTD Enterprise JavaBeans 1.1
 runtime//EN" "http://xmlns.oracle.com/ias/dtds/orion-ejb-jar.dtd">
<orion-ejb-jar>
   <enterprise-beans>
      <session-deployment name="WebAppsClusterBean"
                          location="webAppsCluster/WebAppsClusterBean"
                          idletime="600" copy-by-value="false"  />
   </enterprise-beans>
   <assembly-descriptor>
      <security-role-mapping name="users" />
      <default-method-access>
         <security-role-mapping name="default-ejb-caller-role"
                                impliesAll="true" />
      </default-method-access>
   </assembly-descriptor>
</orion-ejb-jar>
```

For a distributed (clustered) environment, you must add the attribute setting `replication="EndOfCall"` to the `<session-deployment>` element, as follows:

```
      <session-deployment name="WebAppsClusterBean"
                          location="webAppsCluster/WebAppsClusterBean"
                          idletime="600" copy-by-value="false"
                          replication="EndOfCall" />
```

There are no other configuration differences for a distributed versus non-distributed environment.

Notes:

- In the `<security-role-mapping>` element, substitute the appropriate role for *`default-ejb-caller-role`*.
- The `idletime` attribute specifies a timeout value, in seconds, for the stateful session bean. You can adjust this as desired; however, it must be no less than your HTTP session timeout.
- Do not change any other settings.

Enabling Lightweight J2EE Single Sign-On

Enable the lightweight J2EE Single Sign-On implementation by specifying a `true` setting for the system property `oracle.application.sso.`*`appname`* when you start OC4J, where *`appname`* is the name of your J2EE application. For example, for the application `petstore`:

```
java ... -Doracle.application.sso.petstore=true ... -jar oc4j.jar
```

4 Overall Security Configuration

This chapter discusses tasks related to configuring the complete security system. It contains the following parts:

Choosing the XML-Based or LDAP-Based Provider

As part of installing OC4J, you determine whether to use the LDAP-based or XML-based provider. This section gives guidelines on how to make that choice.

- **XML-based provider**: Use the XML-based provider in development environments and in deployed applications with a small user population.
- **LDAP-based provider**: Use the LDAP-based provider in production environments.

  Compared to the XML-based provider, the LDAP-based provider offers better security and performance. The centralized Oracle Internet Directory server scales gracefully as the number of applications and users grows. We recommend you take advantage of a centralized Oracle Internet Directory server in your production deployments, both for better performance and to take advantage of such features as centralized account creation and management, single passwords, and credential management.

The LDAP-based provider enables you to configure cache parameters to improve overall performance of authentication and authorization. If secure communications are needed between the Oracle Internet Directory server and OC4J, configure the system to use the level of security required.

When you install infrastructure, then associate the OC4J instance with a mid-tier Oracle Internet Directory or Oracle Application Server Single Sign-On instance, the installer automatically selects the LDAP-based provider. For details, see the *Oracle Application Server Installation Guide* . If you need to configure OC4J to use the LDAP-based provider, see the instructions in the *Oracle Application Server Administrator's Guide*.

Locating jazn.xml, jazn-data.xml, and the <jazn> Element

To configure the OracleAS JAAS Provider, you must sometimes edit various configuration files using text editors. This section discusses how to locate the configuration files and the `<jazn>` element.

Locating jazn.xml

The OracleAS JAAS Provider must locate a valid `jazn.xml` file before it can begin running. The `jazn.xml` file is used to configure the OracleAS JAAS Provider.

The `jazn.xml` file for the OC4J `home` instance, sometimes referred to as the *bootstrap jazn.xml file*, is in the *`ORACLE_HOME`*`/j2ee/home/config` directory by default. The OracleAS JAAS Provider reads the information in this file before OC4J is started up, and certain settings can be made only in the bootstrap file. If changes are read after OC4J starts up, they have no effect on the OracleAS JAAS Provider.

There is also a `jazn.xml` file for each additional OC4J instance, in the *`ORACLE_HOME`*`/j2ee/`*`instance_name`*`/config` directory.

See Also:

- "Alternative jazn.xml Locations" on page 16-1

Locating jazn-data.xml

The file `jazn-data.xml` is the datastore for the XML-based JAAS provider. By default, for each OC4J instance, OC4J expects the file `jazn-data.xml` to be in the *`ORACLE_HOME`*`/j2ee/`*`instance_name`*`/config` directory. There is also a `jazn-data.xml` file for the OC4J `home` instance, sometimes referred to as the *bootstrap jazn-data.xml file*, in the *`ORACLE_HOME`*`/j2ee/home/config` directory.

You can also specify an alternate path for `jazn-data.xml` in the `<jazn>` element in `jazn.xml`, as follows:

```
<jazn provider="xml" location="pathname">
   ...
</jazn>
```

Locating the <jazn> Element

The `<jazn>` element is used to configure the OracleAS JAAS Provider. Most often, the `<jazn>` element appears in one of two places:

1. The global `application.xml`, for global configuration

2. The application-specific `orion-application.xml`

The `<jazn>` element may also appear in the `jazn.xml` file when it is used to configure virtual machine properties. For details, see "Instance-Level jazn.xml File" on page 5-1.

Admintool Overview

This section discusses basic information needed to understand and use the JAZN Admintool. The JAZN Admintool is a lightweight tool used to manage user accounts stored in `jazn-data.xml`. To manage accounts in the LDAP-based provider, use the tools provided in the Oracle Delegated Administration Services.

Note: Changes made by the Admintool are not effective until the next OC4J restart.

Admintool Prerequisites

When you use the JAZN Admintool, by default it edits the file `jazn-data.xml` under the `config` directory of the OC4J home instance. For details on locating `jazn-data.xml`, see "Locating jazn-data.xml" on page 4-2. The password for the `admin` user is set during installation time to the same value as the Oracle Application Server administrator (`ias_admin`) password.

Before using the Admintool with the LDAP-based provider, be sure to set the correct environment settings as described in "Preparing to Use LDAP" on page 7-1.

Authenticating Yourself

If you are using the XML-based provider, you must authenticate yourself to the JAZN Admintool before making administrative changes. You authenticate by supplying a user name and password when prompted by the Admintool, as in:

```
java -jar jazn.jar -listrealms
>RealmLoginModule username: martha
>RealmLoginModule password: mypass
```

Cautions:

- The Admintool does not require authentication when used with the LDAP-based provider; anyone who runs the tool can perform Admintool operations against the Oracle Internet Directory server. This means that it is vital to secure access to the production machine(s) on which OC4J uses the LDAP-based provider. If you specify the `-user` and `-password` options when using the LDAP-based provider, they are ignored.
- Although it is possible to supply the user name and password on the command line with `-user` and `-password`, we recommend that you avoid this; specifying passwords on command lines creates security vulnerabilities.

Adding Clustering Support

```
-clustersupport oracle_home
```

Specifying this option instructs the Admintool to propagate all JAAS configuration changes throughout a cluster. The *`oracle_home`* argument specifies the absolute path name of *`ORACLE_HOME`*, the Oracle home directory. You can combine `-clustersupport` with the `-shell` option.

Notes:

- In using the `-clustersupport` option, you must specify it before all other options on the command line.
- The `-clustersupport` option is meaningful only when using the XML-based provider.

For example:

```
java -jar jazn.jar -clustersupport /oracle_home -shell
```

Specifying an Admintool Login Module in jazn-data.xml

To specify which `LoginModule` the JAZN Admintool uses to authenticate its users, you must add a `<login-modules>` element to the `<application>` element in `jazn-data.xml`. For example:

```
<application>
  <name>oracle.security.jazn.tools.Admintool</name>
  <login-modules>
    <login-module>
     <class>oracle.security.jazn.realm.RealmLoginModule</class>
      <control-flag>required</control-flag>
      <options>
         <option>
            <name>debug</name>
            <value>false</value>
         </option>
         <option>
            <name>addAllRoles</name>
            <value>true</value>
         </option>
      </options>
    </login-module>
  </login-modules>
</application>
```

If you try to run the JAZN Admintool without specifying a `LoginModule`, the `RealmLoginModule` with the default options is used. The default options are shown in Table 4–2, " RealmLoginModule Options".

Specifying an Alternate Policy Provider (Optional)

If you use the Java Virtual Machine shipped with Oracle Application Server, the OracleAS JAAS Provider is automatically specified as the JAAS policy provider. If you use another JVM, you must explicitly specify `oracle.security.jazn.spi.PolicyProvider` as the policy provider, because by default, the JVM uses the Sun JAAS provider.

Note: When you use OC4J, the JAAS configuration properties are set by default during OC4J startup, so in most circumstances you do not need to worry about setting these properties. You set them only when you are running a J2SE application outside OC4J.

You can specifying Oracle-specific JAAS properties in a separate file and passing them to the JVM on the command line. Oracle supplies a default file (*ORACLE_HOME*/j2ee/home/config/jazn.security.props) that specifies the OracleAS JAAS provider.

- To replace all security properties with the Oracle properties:

  ```
  java -Djava.security.properties==propfile
  ```

- To append the Oracle-specific properties to the other properties:

  ```
  java -Djava.security.properties=propfile
  ```

Specifying OracleAS JAAS Provider Settings

The home instance jazn.xml file is in the *ORACLE_HOME*/j2ee/home/config directory. The OracleAS JAAS Provider reads the information in this file before OC4J is started up, and certain settings can be made only in this file. If changes are read after OC4J starts up, they have no effect on the OracleAS JAAS Provider. These properties are discussed in detail in Chapter 5, "Configuring the OC4J Instance".

Enabling Debug Logging

To enable OracleAS JAAS Provider debug logging, set the system property jazn.debug.log.enable to true during Java Virtual Machine (JVM) startup.

You do this by modifying the JVM startup settings for your OC4J instance. In Oracle Application Server, you normally manage JVM settings with Oracle Enterprise Manager, using the Java Options textbox on the Server Properties page.

In standalone mode, you set this property using JVM command-line options. For example, you might start OC4J standalone with a command line such as:

```
java -Djazn.debug.log.enable=true -jar oc4j.jar
```

Or you can start the Admintool shell in debug mode with the command:

```
java -Djazn.debug.log.enable=true -jar jazn.jar -shell
```

In Oracle Application Server, the debug output is captured by OPMN and written to log files associated with each OC4J instance in the *ORACLE_HOME*/opmn/logs directory.

Specifying User Managers

The user manager, employing the user name and password, verifies the user's identity using information in the user repository. The user manager contains your definitions for users, groups, or roles. The default user manager is the JAZNUserManager.

You can define a user manager for all applications or for specific applications.

- Global user manager: The global (default) user manager is inherited by all applications within an instance that has not defined a specific user manager. The instance `UserManager` is defined in the `application.xml` file.
- Specific user manager: This user manager is defined in `orion_application.xml` solely for a single application. It is not used by any other application.

> **Note:** Within a single OC4J instance you can specify different values for the application-specific `UserManager` instance and the global `UserManager` instance. When you do this, we recommend that you not mix custom `UserManager` instances and Oracle-supplied `UserManager` instances. You can use different custom `UserManager` instances for the application and the global instance, and you can use different Oracle-supplied `UserManager` instances for the application and the global instance, but you should avoid using a custom `UserManager` for the one instance and an Oracle-supplied `UserManager` for the other.

- In some cases, if an application inherits from another application instead of inheriting from the global application, then the parent user manager of the application will be the global `UserManager` instance instead of the `UserManager` instance specified in the parent application.

Specifying a User Manager

To specify a `UserManager` for an entire OC4J instance or for a specific application within that instance, use Enterprise Manager. For details, see the Enterprise Manager help topic "Modifying the User Manager for All Applications".

Specifying a User Manager in orion-application.xml

Every application, including the top-level default application, has an associated `UserManager`. The primary function of the `UserManager` is to authenticate users who attempt to access Web pages and EJBs.

> **Note:** We strongly encourage you to use `JAZNUserManager`. User-defined user managers will stop being supported in a future release.

The `UserManager` is used to authenticate users when connections are made to the application. These are specified using subelements within an `<orion-application>` element that define the configuration. There are three elements that can be used to specify a `UserManager`, as shown in Table 4–1.

Table 4–1 UserManager Elements

Tag	Meaning
`<user-manager>`	A user manager implemented by a user-defined class
`<jazn>`	`JAZNUserManager`

Table 4–1 (Cont.) UserManager Elements

Tag	Meaning
`<principals>`	A user manager defined in a `principals.xml` file See "Using the <principals> Element and principals.xml" on page 4-14.

Advanced Configuration

There may be more than one of the user-manager configuration within a single `<orion-application>` element. Which element determines the `UserManager` is determined by the order the elements appear in the table: `<user-manager>` takes precedence over `<jazn>`, which takes precedence over `<principals>`. For example, if both a `<jazn>` element and a `<principals>` element are present, the `UserManager` is based on the `<jazn>` element. If multiple elements with the highest-priority element are present, then the `UserManager` instances are chained together as parents. That is, the `UserManager` specified in the first element becomes the parent of the `UserManager` specified in the second, and so on. The last `UserManager` specified then becomes the `UserManager` of the application. The parent of the first `UserManager` is the `UserManager` associated with the parent application (if any) of the application. The default application does not have a parent application and the parent of its `UserManager` is null.

If no user manager is specified, then the `UserManager` is determined according to the following rules.

- For the default application, a JAAS `UserManager` is created based on `jazn-data.xml` in the directory containing the `application.xml` file. If no `jazn-data.xml` is present in that directory, one is created. The default realm of the created `jazn-data.xml` file is `jazn.com`.
- At deployment time, if the `UserManager` of the parent application is the JAAS `UserManager`, then a JAAS `UserManager` is created based on `jazn-data.xml`. If necessary, a `jazn-data.xml` file is created in the same way as in the previous bullet. A `<jazn>` element is written into the `orion-application.xml` file associated with the application.
- At application deployment time, if the `UserManager` of the parent application is based on `principals.xml`, then the `UserManager` of the application will be a principals `UserManager`. If a `principals.xml` file is not present, then an empty file is created. A `<jazn>` element is written into the `orion-application.xml` associated with the application.
- If the `UserManager` of the parent application is user-written, then the `UserManager` of the parent will become the `UserManager` of the application.

Customizing RealmLoginModule

The `RealmLoginModule` class is the default `LoginModule` that is configured through the `jazn-data.xml` file. The `RealmLoginModule` class authenticates user login credentials before the user can access J2EE applications. Authentication is performed using OC4J container-based authentication (HTTP `BASIC`, `FORM`, and so on). You do not need to enable the `RealmLoginModule` class if your application uses OracleAS Single Sign-On authentication.

See Also:

- *Oracle Application Server Installation Guide* for OracleAS Single Sign-On configuration tasks

You can configure `RealmLoginModule` either using the JAZN Admintool or by editing `jazn-data.xml`. For details on using the Admintool, see "Adding and Removing Login Modules" on page 10-4.

The `<login-module>` element supports the following `<option>` values:

Table 4–2 RealmLoginModule Options

Name	Meaning	Default
`debug`	If set to `true`, prints debugging messages.	`false`
`addRoles`	If set to `true`, the `RealmLoginModule` adds all directly granted roles of the user to the subject after successful authentication.	`true`
`addAllRoles`	If set to `true`, the `RealmLoginModule` adds all directly or indirectly granted roles of the user to the subject after successful authentication.	`true`
`storePrivateCredentials`	If set to `true`, the `RealmLoginModule` adds all private credentials (for example, password credentials) to the subject after successful authentication.	`false`
`supportCSIv2`	If set to `true`, the `RealmLoginModule` supports CSIv2. See Chapter 15, "Configuring CSIv2" for details.	`false`
`supportNullPassword`	(LDAP-based provider **only**.) If set to `true`, the `RealmLoginModule` does not check to see if the supplied password is null or empty. If set to `false`, authentication fails if the supplied password is null or empty.	`false`

Enabling RealmLoginModule Using a Text Editor

Use a text editor to modify the login configuration file `jazn-data.xml` where needed.

The default configuration for the `RealmLoginModule` class setting in the `jazn-data.xml` file is as follows:

```
<!DOCTYPE jazn-data (View Source for full doctype...)>
<jazn-data>
    ...
<!--  Login Module Data -->
 <jazn-loginconfig>
   <application>
     <name>oracle.security.jazn.oc4j.JAZNUserManager</name>
     <login-modules>
       <login-module>
          <class>oracle.security.jazn.realm.RealmLoginModule</class>
          <control-flag>required</control-flag>
          <options>
             <option>
               <name>addRoles</name>
               <value>true</value>
             </option>
         </options>
```

```
        </login-module>
      </login-modules>
    </application>
  </jazn-loginconfig>
</jazn-data>
```

See Also:

- OracleAS JAAS Provider Javadoc: *JAAS Provider API Reference*
- "Adding and Removing Login Modules" on page 10-4

Specifying Authentication (auth-method)

You specify the authentication method (`auth-method`) in one of several configuration files, using either the `<jazn-web-app>` or `<login-config>` elements. You must edit the configuration files by hand.

Specifying auth-method in web.xml

To specify the authentication method at the Web module level, use the `<login-config>` element of `web.xml`. For example:

```
<web-app>
   ...
   <login-config>
      <auth-method>BASIC</auth-method>
   </login-config>
   ...
</web-app>
```

In `web.xml`, the `<auth-method>` element can have the values shown in Table 4–3. Note that these values can be overridden at the application level by using an attribute of the `<jazn-web-app>` element in `orion-application.xml`.

Table 4–3 Values for auth-method in web.xml

Setting	Meaning
`BASIC` (default)	The application uses basic authentication, the standard authentication. **Note**: Also use the `BASIC` setting if you want to use Oracle SSO or digest features, where the `web.xml` setting is overridden in a `<jazn-web-app>` setting. See "Specifying auth-method in orion-application.xml" immediately below. (This is how Oracle implements the optional digest mechanism, for the XML-based provider only, in the OC4J 10.1.2 implementation.)
`FORM`	The application uses form-based authentication.
`CLIENT-CERT`	The application requires the client to supply its own certificate for use with SSL.

Specifying auth-method in orion-application.xml

An `auth-method` attribute setting supplied in a `<jazn-web-app>` element overrides the `auth-method` setting in `web.xml`. The OC4J 10.1.2 implementation supports `SSO` or `DIGEST` settings in `<jazn-web-app>`.

Specifying auth-method SSO

The setting `auth-method="SSO"` in the `<jazn-web-app>` element of the `orion-application.xml` file specifies that the application uses OracleAS Single Sign-On. If your installation includes LDAP, then Oracle Enterprise Manager automatically sets `auth-method="SSO"`. If you stop using OracleAS Single Sign-On, you must edit the file to remove this method.

To use the SSO authentication mechanism, use the `auth-method` setting `BASIC` in `web.xml`, as follows:

```
<web-app>
   ...
   <login-config>
      <auth-method>BASIC</auth-method>
   </login-config>
   ...
</web-app>
```

Then override it with settings in `orion-application.xml` such as the following:

```
<jazn provider="LDAP">
   ...
   <jazn-web-app auth-method="SSO"/>
   ...
</jazn>
```

Specifying auth-method Digest

To use the digest authentication mechanism, use the `auth-method` setting `BASIC` in `web.xml`, as follows:

```
<web-app>
   ...
   <login-config>
      <auth-method>BASIC</auth-method>
   </login-config>
   ...
</web-app>
```

Then override this with the setting `auth-method="DIGEST"` in the `orion-application.xml` file, as follows:

```
<jazn>
   ...
   <jazn-web-app auth-method="DIGEST" />
   ...
</jazn>
```

Important: Be aware of the following:

- In the OC4J 10.1.2 implementation, the digest mechanism is supported only for the JAZN XML-based provider.
- Older Web browsers may not support the digest mechanism.
- The OC4J digest feature is unrelated to the Oracle HTTP Server `mod_digest` feature.

Configuring J2EE Authorization

J2EE defines a declarative authorization model that decouples applications from the underlying security infrastructure. This model allows the authorization policy of an application to be expressed in a portable manner in the application deployment descriptors. This model has proven to be hugely successful and suffices for most application needs.

In some advanced scenarios, however, the J2EE authorization model may seem too static and coarse-grained. In these cases, the JAAS authorization model can be used instead of (or in addition to) the J2EE security model. When compared to the J2EE authorization model, the JAAS authorization model is more powerful (fine-grained and dynamic) and more flexible (custom permission types supported). Such power and flexibility come at a cost, however, the JAAS authorization model is more complex to understand, deploy and administer than the J2EE authorization model.

Both models are fully supported in OC4J.

Servlets, runas-mode, and doasprivileged-mode

In most cases, URL authorization is sufficient for an application. However, if your application requires granular authorization, this extended authorization can be used to enforce JAAS Policy-based authorization at the method level.

If you want a servlet to be invoked using `subject.doAs()` or `subject.doAsPrivileged()`, you must set the `runas-mode` and `doasprivileged-mode` attributes of the `<jazn-web-app>` element contained in a `<jazn>` element in either the `orion-web.xml` or `orion-application.xml` file. To do this, you open the appropriate file in a text editor.

- The `subject.doAs()` method invokes the servlet using the privileges of a particular *subject*. A subject is defined by an instance of the `javax.security.auth.Subject` class and includes a set of facts regarding a single entity, such as a person. Such facts include identities and security-related attributes, such as passwords and cryptographic keys. The OracleAS JAAS Provider passes in the `Subject` instance in the method call.

  When the `doAs()` method is used, an `AccessControlContext` instance is retrieved from the current thread (from the server).

- The `subject.doAsPrivileged()` method uses the privileges of a particular subject without being limited by the access-control restrictions of the server.

  When the `doAsPrivileged()` method is used, the OracleAS JAAS Provider invokes the method with a null `java.security.AccessControlContext` instance, in order to start the action fresh and execute the servlet without the restrictions of the current server `AccessControlContext` instance.

The `runas-mode` and `doasprivileged-mode` attributes control whether the servlet is invoked with the `subject.doAsPrivileged()` method or the `subject.doAs()` method. By default, `runas-mode` is set to `false`, which means that neither `subject.doAsPrivileged()` nor `subject.doAs()` is invoked.

Note: The `runas-mode` attribute is unrelated to the `servlet.runAs()` method.

Table 4–4 Settings for runas-mode and doasprivileged-mode

If runas-mode is Set To	And doasprivileged-mode Is Set To	Then the servlet is invoked with:
`false` (default)	`true` or `false`	No special privileges
`true`	`true` (default)	`subject.doAsPrivileged()`
`true`	`false`	`subject.doAs()`

Thus, to have your servlet invoked using `subject.doAsPrivileged()`, you should have a `<jazn-web-app>` element that looks like this:

```
<jazn-web-app
    auth-method="SSO"
    runas-mode="true"
    doasprivileged-mode="true" />
```

Mapping Logical Roles to Security Roles

You sometimes deploy a servlet into a container that uses different role names than expected by the servlet. You do this by mapping the logical role (the role name used by the servlet) to the security role (the role name used by the container) in the `web.xml` file.

For example, the following entity maps the security role `corpmanagers` to the logical role `mgmt`:

```
<security-role-ref>
  <role-name>mgmt</role-name>
  <role-link>corpmanagers</role-link>
</security-role-ref>
```

In this example, if a servlet running as a user belonging to `corpmanagers` invokes `isUserInRole("mgmt")`, the method will return `true`. Whenever the container finds no `security-role-ref` matching a security role, the container checks the `<role-name>` against the entire list of security roles for the Web application.

Removing Realm Names from Authentication Principals

It is often desirable to avoid parsing the principal returned by various method calls. You can configure OracleAS JAAS Provider so that the returned principal contains no realm name. To do this, you add a `jaas.username.simple` property to the `<jazn>` element in the instance-level `jazn.xml` file, *`ORACLE_HOME`*`/j2ee/`*`instance_name`*`/config/jazn.xml`. If this property is set to `true`, returned principals contain no realm name; if it is set to `false`, the default, returned principals contain the complete realm name.

The `jaas.username.simple` property affects only the specified OC4J instance. Setting this property in `orion-application.xml` has no effect.

This property affects the return values of the following methods:

- `javax.servlet.http.HTTPServletRequest`: `getRemoteUser()` and `getUserPrincipal()` methods
- `javax.ejb.EJBContext`: `getCallerIdentity()` and `getCallerPrincipal()` methods

By default, the principal returned by these methods is in the format `realm_name/simple_name`, such as `jazn.com/john`. When you set `jaas.username.simple` to `true`, the returned principal is in the format `simple_name`, such as `john`.

You set `jaas.username.simple` as follows:

1. Locate the instance-level `jazn.xml` file, *`ORACLE_HOME`*`/j2ee/`*`instance_name`*`/config/jazn.xml`. Open the file in a text editor, and go to the `<jazn>` element within the file.
2. Search for a `<property>` subelement with `name="jaas.username.simple"` within the `<jazn>` element.
3. If the subelement exists, change `value` to `true` or `false`; if the subelement does not exist, create one. In either case, you should have a subelement that looks like this:

```
<jazn provider="XML" location="./jazn-data.xml">
   <property name="jaas.username.simple" value="true" />
</jazn>
```

Note: Do not edit any `<jazn>` properties except as specified in this chapter.

Configuring Third-Party LDAP Providers

See Chapter 9, "Configuring External LDAP Providers".

Permitting EJB RMI Client Access

To enable fat client access to EJBs using RMI, you must grant the correct permissions using the JAZN Admintool. (For general information on using the Admintool, see "Admintool Overview" on page 4-3.) You must grant `RMIPermission` of `login` to your user, role, or group.

```
java -jar jazn.jar -grantperm myrealm -role administrators \
  oracle.j2ee.server.rmi.RMIPermission login
```

Creating a Java 2 Policy File

The Java 2 policy file grants permissions to trusted code or applications that you run. This enables code or applications to access Oracle support for JAAS or JDK APIs requiring specific access privileges.

A preconfigured Java 2 policy (`java2.policy`) is provided in *`ORACLE_HOME`*`/j2ee/home/config`.

You need to modify the Java 2 policy file to grant permissions to trusted code or applications.

For example, the following section of a `java2.policy` file grants `java.security.AllPermission` to the trusted `jazn.jar`.

```
/* grant the JAZN library AllPermission */
grant codebase "file:${oracle.home}/j2ee/home/jazn.jar" {
    permission java.security.AllPermission;
};
```

The following example grants specific permissions to all applications running in the *ORACLE_HOME*/appdemo directory.

```
/* Assuming you are running your application demo in $ORACLE_HOME/appdemo/, */
/* Grant JAZN permissions to the demo to run JAZN APIs*/
grant codebase "file:/${oracle.ons.oraclehome}/appdemo/-" {
   permission oracle.security.jazn.JAZNPermission "getPolicy";
   permission oracle.security.jazn.JAZNPermission "getRealmManager";
   permission oracle.security.jazn.policy.AdminPermission
     "oracle.security.jazn.realm.RealmPermission$*$createRealm,dropRealm,
     createRole, dropRole,modifyRealmMetaData";
```

Using the <principals> Element and principals.xml

The `<principals>` element tells OC4J to use the `UserManager` described in a principals file, normally `principals.xml`. A `<principals>` element has one attribute, `<path>`, which specifies a path for the principals file, normally `principals.xml`.

For example:

```
<principals path="myprincipals.xml" />
```

A `principals.xml` file also contains a `<principals>` element; this contains two subelements, `<groups>` and `<users>`. The `<groups>` element contains one or more `<group>` elements, and the `<users>` element contains one or more `<user>` elements.

Note: The `XMLUserManager` class is deprecated, and is supported for backward compatibility only. Oracle will cease to support `XMLUserManager` and `principals.xml` in a future release.

Table 4–5 Elements in principals.xml

Element	Can Contain	Attributes	Description
`<principals>`	`<groups>, <users>`	NA	The top-level element for principals.
`<groups>`	`<group>`		A list of groups known to this user manager.
`<group>`	`<description>`, `<permission>`	name	Identifies a single user group; `name` attribute specifies group name.
`<description>`			Not used by OracleAS JAAS Provider, but displayed in some circumstances.
`<permission>`		name	A `java.security.Permission` that is granted to principals. There are two special values: ■ `administrator`, equivalent to `oracle.j2ee.security.AdministrationPermission()` ■ `rmi:login`, equivalent to `oracle.j2ee.server.rm.RMIPermission("login")`

Table 4–5 (Cont.) Elements in principals.xml

Element	Can Contain	Attributes	Description
`<users>`	`<user>`		List of users known to the `UserManager`.
`<user>`	`<description>`, `<group-membership>`	`username`, `password`, `deactivated`	The `username` is a string to identify the user. The `password` is a cleartext password to authenticate the user. There is no mechanism for obfuscating this password. The `deactivated` attribute is either `true` or `false`. If `true`, then this user will not be found in lookups and cannot be authenticated.
`<group-membership>`		`group`	Name attribute of a group that contains this user.

Groups in `principals.xml` correspond to roles in the OracleAS JAAS Provider. The `principals.xml` file does not support any equivalent of the OracleAS JAAS Provider concept of realms. Permissions granted to groups may be checked explicitly, and OC4J does check for the special permissions listed above. However, group permissions are not integrated with the usual `Permission` checking performed by a `SecurityManager`.

The following is a sample `principals.xml` file.

```
<?xml version="1.0" standalone='yes'?>
<!DOCTYPE principals PUBLIC "//Evermind - Orion Principals//"
 "http://xmlns.oracle.com/ias/dtds/principals.dtd">

<principals>
  <groups>
<group name="guests">
    <description>users</description>
  </group>
  <group name="administrators">
    <description>administrators</description>
    <permission name="administration" />
  </group>
  </groups>
  <users>
  <user username="SCOTT" password="TIGER">
  <group-membership group="guests" />
  </user>
  <user username="anonymous" password="">
    <description>The default guest/anonymous user</description>
    <group-membership group="guests" />
  </user>
  <user username="admin" password=""  deactivated="true">
    <description>The default administrator</description>
    <group-membership group="users" />
    <group-membership group="administrators" />
  </user>
  </users>
</principals>
```

5

Configuring the OC4J Instance

This chapter discusses instance-level OC4J configuration. All tasks in this chapter affect an entire OC4J instance and all applications running under that instance. This chapter contains the following sections:

The admin Account

Whenever you create a new OC4J instance, that instance is given an administrator account `admin` with password `welcome`. You should change this password immediately using Oracle Enterprise Manager 10*g* Application Server Control Console.

Here are the steps to change the password:

1. From the Application Server Home page, select the OC4J instance.
2. From the home page of the OC4J instance, choose Administration.
3. From the OC4J Administration page, choose Security.
4. From the Security page, choose `jazn.com/admin` (under Users) to edit the administrative user properties.

Instance-Level jazn.xml File

All of the tasks in this chapter rely on editing the OC4J instance-level `jazn.xml` file, which is read at instance startup. The instance-level `jazn.xml` file is *`ORACLE_HOME`*`/j2ee/`*`instance_name`*`/config/jazn.xml`. All changes to this file affect the entire OC4J instance. The properties listed in this section can be changed only in the instance-level `jazn.xml` file.

Note: You cannot change the `jazn.xml` file with Application Server Control Console; you must edit it using a text editor.

Specifying LDAP Connection Properties

There are two properties that change LDAP connection properties. They are listed in Table 5–1.

Table 5–1 LDAP Connection Properties

Property Name	Meaning	Default Value
`ldap.connect.max.retry`	Number of times the OracleAS JAAS Provider attempts to create an LDAP connection before giving up	5
`ldap.connect.sleep`	Number of milliseconds the OracleAS JAAS Provider waits before retrying a failed LDAP connection attempt	5000

To configure LDAP connection properties, use the following steps:

1. Open the `jazn.xml` file, *`ORACLE_HOME`*`/j2ee/`*`instance_name`*`/config/jazn.xml`, in a text editor and go to the `<jazn>` element within the file.
2. Locate the `<property>` subelement within the `<jazn>` element. The syntax of the `<property>` subelement is:

    ```
    <property name="propname" value="propvalue"/>
    ```

    If there is no `<property>` subelement corresponding to the property you want to change, create one.
3. Restart OC4J.

Specifying LDAP JNDI Connection Pool Size

There are two properties that change LDAP connection pool properties. They are listed in Table 5–2.

Table 5–2 LDAP JNDI Connection Pool Properties

Property Name	Meaning	Default Value
`jndi.ctx_pool.init_size`	Initial size for JNDI/LDAP connection pool	5
`jndi.ctx_pool.inc_size`	Pool increment size for JNDI/LDAP connection pool—number of connections added to pool whenever the supply of connections in the pool is exhausted	10

To specify the size of the connection pool used by JNDI:

1. Open the `jazn.xml` file, *`ORACLE_HOME`*`/j2ee/`*`instance_name`*`/config/jazn.xml`, in a text editor and go to the `<jazn>` element within the file.
2. Locate the `<property>` subelement within the `<jazn>` element. The syntax of the `<property>` subelement is:

    ```
    <property name="propname" value="propvalue"/>
    ```

    If there is no `<property>` subelement corresponding to the property you wish to change, create one. For example, a `<property>` subelement setting the initial size to 20 would look like:

```
<property name="jndi.ctx_pool.init_size" value="20">
```

Note: Do not edit any `<jazn>` properties except as specified in this documentation.

3. Restart OC4J.

Configuring LDAP Caching

The LDAP-based OracleAS JAAS Provider supports caching, providing improved performance and scalability. There are three separate caches:

- Policy cache, which stores grantees and permissions
- Realm cache, which stores realms, users and roles, and a role graph
- Session cache, which stores users and role graphs in an HTTP session object (available only to Web-based clients with cookies enabled)

The caching service maintains a global `HashMap`, which is used to store and retrieve cached objects. A daemon thread runs periodically in the background to invalidate and clean up expired objects in the `HashMap`. Objects in the cache expire based on a time-to-live algorithm; expiration time can be set with the cache properties, described in Table 5–3.

Note: Only the LDAP-based provider provides these caches. The XML-based provider defaults to caching the entire XML document.

Changing Session Cache Details

`HttpSession` objects persist for the duration of the server-side session. An application can terminate a session explicitly, by invoking `HttpSession.invalidate();` a container can terminate a session based on the `<session-timeout>` value.

Note: Objects stored in an `HttpSession` instance must implement the `java.io.Serializable` interface in order to be deployed with the `<distributable />` flag in `web.xml`.

See Also:

- The *Oracle HTTP Server Administrator's Guide* for more information about session support in OC4J

Disabling LDAP Caching

Caching is enabled by default. You should disable the caches when performing management and administrative tasks programmatically. In particular:

- Disable the policy cache when managing policy. If the policy cache is enabled, calling `Policy.grant()` or `Policy.revoke()` causes an `UnsupportedOperationException`.
- Disable the realm cache when managing realms. This includes adding realms, dropping realms, granting roles, and revoking roles.

- Disable the session cache when you disable HTTP session cookies.

Note: The JAZN Admintool automatically disables caching while it is in operation, then reenables caching when it finishes.

To disable the LDAP cache, use the following steps:

1. Open the `jazn.xml` file, `*ORACLE_HOME*/j2ee/*instance_name*/config/jazn.xml`, in a text editor and go to the `<jazn>` element within the file.
2. Edit the `<jazn>` element to appear as follows:

```
<jazn provider="LDAP">
   <property name="ldap.user"
             value="orclApplicationCommonName=jaznadmin1,cn=JAZNContext,
                    cn=products,cn=OracleContext"/>
   <property name="ldap.password"
             value="{903}3o4PTHbgMzVlzbVfKITIO5Bgio6KK9kD"/>
   <property name="ldap.cache.session.enable"
             value="false" />
   <property name="ldap.cache.realm.enable"
             value="false" />
   <property name="ldap.cache.policy.enable"
             value="false" />
</jazn>
```

3. Restart OC4J.

LDAP Cache Configuration

The properties that affect the LDAP cache are controlled by `<property>` subelements within the `<jazn>` element. To change these properties, you must edit the `jazn.xml` file and change the `<jazn>` element.

To configure LDAP cache properties, use the following steps:

1. Open the `jazn.xml` file, `*ORACLE_HOME*/j2ee/*instance_name*/config/jazn.xml`, in a text editor and go to the `<jazn>` element within the file.
2. Locate the `<property>` subelement within the `<jazn>` element. The syntax of the `<property>` subelement is:

```
<property name="propname" value="propvalue"/>
```

   If there is no `<property>` subelement corresponding to the property you wish to change, create one.
3. Restart OC4J.

Table 5–3 describes the LDAP cache properties and their default values. You can set these properties only at the instance level, in the `<jazn>` element in the `jazn.xml`.

Table 5–3 LDAP Cache Properties

Property	Description	Default
`ldap.cache.policy.enable` (see **Notes**)	If set to `true`, enables cache; if set to `false`, disables cache.	`true`

Table 5–3 (Cont.) LDAP Cache Properties

Property	Description	Default
`ldap.cache.realm.enable`	If set to `true`, enables cache; if set to `false`, disables cache.	`true`
`ldap.cache.session.enable`	If set to `true`, enables cache; if set to `false`, disables cache.	`true`
`ldap.cache.initial.capacity`	Initial capacity for the `HashMap`.	`20`
`ldap.cache.load.factor`	Load factor for the `HashMap`.	`0.7`
`ldap.cache.purge.initial.delay`	String containing an integer that represents the number of milliseconds the daemon thread waits before starts checking for expired objects.	`3600000`
`ldap.cache.purge.timeout`	The string representation of an integer that represents the number of milliseconds an object remains in cache before being invalidated and removed. It is also the sleep time for the daemon thread between each run looking for expired objects.	`3600000`

Notes:

- Do not edit any `<jazn>` properties except as specified in this documentation.
- The `ldap.cache.policy.enable` property replaces the deprecated `ldap.cache.enable` property.

A `jazn` element with all caches enabled, a cache size of 100, and a 10000-millisecond timeout would look like:

```
< jazn provider="LDAP" location="ldap://example.com:389" >
   < property name="ldap.cache.initial capacity" value="100" />
   < property name="ldap.cache.purget.timeout" value="10000" />
</jazn>
```

Configuring LDAP SSL Properties

The properties that affect SSL are controlled by `<property>` subelements within the `<jazn>` element. To change these properties, you must edit the file containing the `<jazn>` element.

To configure LDAP SSL properties, use the following steps:

1. Open the `jazn.xml` file, *`ORACLE_HOME`*`/j2ee/`*`instance_name`*`/config/jazn.xml`, in a text editor and go to the `<jazn>` element within the file.
2. Locate the `<property>` subelement within the `<jazn>` element. The syntax of the `<property>` subelement is:

   ```
   <property name="propname" value="propvalue"/>
   ```

   If there is no `<property>` subelement corresponding to the property you wish to change, create one.

3. Restart OC4J.

Table 5–4 lists the SSL properties.

Table 5–4 Values for <property> Subelement of <jazn> Element

Property Name	Value
`ldap.password`	Obfuscated password for the LDAP user name. For example: `{903}oZZYqmGc/iyCaDrD4qs2FHbXf3LAWtMN` See "Password Obfuscation in jazn-data.xml and jazn.xml" on page 14-1 for details on obfuscation.
`ldap.protocol`	The protocol to be used when communicating with LDAP using SSL.

`ldap.user`	LDAP user name or `DN`. This element is populated automatically; you should not change the contents. For example:
	`orclApplicationCommonName=jaznadmin1,cn=JAZNContext,`
	`cn=products,cn=OracleContext`

Note: Do not edit any `<jazn>` properties except as specified in this document.

Choosing SSL Authentication

This section discusses configuring the OracleAS JAAS Provider to use SSL with Oracle Internet Directory. For information on how to configure Oracle Internet Directory to use SSL, see the *Oracle Internet Directory Administrator's Guide* and *Oracle Application Server Containers for J2EE Servlet Developer's Guide* .

At 10*g* Release 2 (10.1.2), you must use `NULL` authentication when communicating with Oracle Internet Directory. `NULL` authentication means that data are encrypted with the Anonymous Diffie-Hellman cipher suite, but no certificates are used for authentication.

If you choose SSL at install time, SSL is enabled with `NULL` authentication in place. You must manually enable SSL only if you did not choose SSL as part of your installation. In that case, for `NULL` authentication, add a `<property>` element to the `<jazn>` element in `jazn.xml` to specify a protocol. (Note that you do not specify a wallet location or password, because `NULL` authentication does not use certificates.)

```
<jazn provider="LDAP" location="ldap://example.com:5000" default-realm="us">
   ...
   <property name="ldap.protocol" value="ssl"/>
   ...
</jazn>
```

Configuring LDAP Default Realm

The default realm is the realm used whenever an authentication or authorization request does not specify a realm explicitly. This attribute is automatically populated with the default Oracle Identity Management realm; you need to edit the attribute only if the default is incorrect for your application. To configure the LDAP default realm, use the following steps:

1. Open the `jazn.xml` file, `*ORACLE_HOME*/j2ee/*instance_name*/config/jazn.xml`, in a text editor and go to the `<jazn>` element within the file.

2. Edit the `default-realm` attribute of the `<jazn>` element. The syntax is:

   ```
   <jazn provider="LDAP" default-realm="myrealm">
      ...
   </jazn>
   ```

3. Restart OC4J.

   > **Note:** Do not edit any `<jazn>` properties except as specified in this documentation.

For example, a `<jazn>` element that set the default-realm to "Sales" would look like:

```
<jazn provider="LDAP" default-realm="Sales" ... more attributes ... >
   ...
</jazn>
```

6 Security Considerations During Application Deployment

This chapter discusses issues to be considered when deploying applications. It is divided into the following sections:

Selecting a User Manager

By default, if you associated your OC4J instance with infrastructure, the JAZN LDAP `UserManager` is used for your newly-deployed application; otherwise, the JAZN XML `UserManager` is used for your application. If for some reason you need to change the user manager of your application, you can do so from the Application Server Control Console. For details, see the Application Server Control Console help topic "Modifying the User Manager for All Applications".

Mapping Security Roles

You map security roles for your application using the Security page of the Application Server Control Console. Use the following steps:

1. Select your application from the Application Server Control Console, then click the Security link.
2. Select a role from the list titled Security Roles.
3. Click the button **Map Roles To Principals**. A new page appears headed Role: *yourrole*.
4. Click the checkbox next to the desired group or user. (There are two separate areas labeled Map Role to Groups and Map Role to Users.) Click **Apply**.
5. A confirmation page appears. Click **OK**.

Granting Permissions

There are two different ways to grant permissions.

- To grant RMI permission or `administration` permission, use Oracle Enterprise Manager 10*g* Application Server Control Console. For details, see "Granting RMI Permission or Administration Permission" immediately following.
- To grant any permissions other than RMI permission or `administration` permission, you use the JAZN Admintool. For details, see "Granting and Revoking All Other Permissions" below.

Granting RMI Permission or Administration Permission

You can grant RMI or `administration` permission to a group using Oracle Enterprise Manager 10*g* Application Server Control Console. To do this:

1. Select an application and navigate to the Security page.
2. Select the group name from the list of groups. The Add/Edit Group page appears.
3. Check whichever permissions you wish to add and click **Apply**.

Granting and Revoking All Other Permissions

You use the JAZN Admintool to grant and revoke user permissions. For basic information on running the JAZN Admintool, see "Admintool Overview" on page 4-3.

```
-grantperm {realm {-user user|-role role } | principal_class principal_parameters}
           permission_class [permission_parameters]
-revokeperm {realm {-user user|-role role} | principal_class principal_parameters}
            permission_class [permission_parameters]
-listperms {realm {-user user|-role role} | principal_class principal_parameters}
           permission_class [permission_parameters]
```

In this syntax, *`principal_class`* is the fully qualified name of a class that implements the principal interface (such as `com.sun.security.auth.NTDomainPrincipal`) and *`principal_parameters`* is a single `String` parameter.

The `-grantperm` option grants the specified permission to a user (when called with `-user`) or a role (when called with `-role)` or a principal. The `-revokeperm` option revokes the specified permission from a user or role or principal.

A *`permission_descriptor`* consists of the explicit class name of a permissions (for example, `oracle.security.jazn.realm.RealmPermission`), its action, and its action and target parameters (for `RealmPermission`, `realmname action`). Note that there may be multiple action and target parameters.

Note: If the Admintool gives the error message "`Permission class not found`", it means that the permission you wish to grant is not in the classpath. You must place the JAR containing the permission class in the `jdk/jre/lib/ext` directory so that the Admintool can locate it.

For example, to grant `FilePermission` with target `a.txt` and actions "`read, write`" to user `martha` in realm `foo`, type:

```
java -jar jazn.jar -grantperm foo -user martha java.io.FilePermission
     a.txt read,write
```

Admintool shell:

```
JAZN:> grantperm foo -user martha java.io.FilePermission a.txt read,write
```

Creating Users and Groups

See Chapter 7, "Configuring the LDAP-Based Provider" or Chapter 8, "Configuring the XML-Based Provider" for details on creating users and groups in each provider.

7 Configuring the LDAP-Based Provider

This chapter discusses configuring OC4J to use the Oracle Internet Directory (OID) LDAP-based provider. It contains the following sections:

- Preparing to Use LDAP
- Creating LDAP Users and Groups

Some LDAP properties affect the entire OC4J instance; these properties are discussed in "Specifying OracleAS JAAS Provider Settings" on page 4-5.

Preparing to Use LDAP

You normally associate OC4J with infrastructure at the time of installation.However, you can also associate OC4J with infrastructure using Oracle Enterprise Manager 10*g* Application Server Control Console. See the Oracle Enterprise Manager 10*g* help topic "Application Server Infrastructure Page".

When you associate an OC4J instance with an Oracle Application Server Infrastructure (including the Oracle Internet Directory), your application can leverage the LDAP-based provider for central management of users.

Creating Administrative Users and Groups

Before using the LDAP-based provider, you must set up certain users, groups, and permissions in Oracle Delegated Administration Services, and then grant these users and groups the appropriate permissions.

If you specify the LDAP-based provider globally in the `*ORACLE_HOME*/j2ee/*instance_name*/config/application.xml` configuration file, then you must also create an anonymous user, as discussed in "Creating an anonymous User Using ldapmodify" on page 7-2. Under normal conditions, you do not need to modify `application.xml`. The principal reason to do so is to configure the default application in an OC4J instance to use the LDAP-based provider as the user manager. The default application is a system application created by OC4J for internal use. (See the deployment and configuration overview in the *Oracle Application Server Containers for J2EE Servlet Developer's Guide* for information about the OC4J default application.)

Creating Users Using LoadOidData

You can set up the appropriate groups and users by using the tool `oracle.security.jazn.util.LoadOidData`, which is part of the `jazncore` library supplied in the *`ORACLE_HOME`* directory. You run the tool with the command line:

```
java -cp ./jazncore.jar oracle.security.jazn.util.LoadOidData
```

The syntax for this tool is:

```
LoadOidData [-h ldaphost] [-p ldapport] [-D binddn] [-w passwd]
            [-f filename [-oc4jAdminPwd passwd] [-ignoreError true|false]
```

The supported options are:

- `-h` *`ldaphost`* for the LDAP host name
- `-p` *`ldapport`* for the port of the LDAP server
- `-D` *`binddn`* for the distinguished name for the Oracle Internet Directory administrator
- `-w` *`password`* for the password of the Oracle Internet Directory administrator
- `-f` *`filename`* for the file containing the entries to be loaded, which is the following:

  `ORACLE_HOME``/j2ee/`*`instance_name`*`/jazn/install/oidConfigForOc4j.sbs`

- `-oc4jAdminPwd` *`password`* for the password that will be assigned to OC4J administrator
- `-ignoreError` *`boolean`* to specify whether the tool continues after reporting an error (if `true`) or stops as soon as it encounters an error (if `false`).

For example, assume the password for the Oracle database administrator is `welcome1` and the password for the OC4J administrative user is `welcome2`. The command line (assuming `$J2EE_HOME` is *`ORACLE_HOME`*`/j2ee/home`) would be:

```
java -cp $J2EE_HOME/jazncore.jar oracle.security.jazn.util.LoadOidData
     -h oidhost -p oidport -D cn=orcladmin -w welcome1
     -f $J2EE_HOME/jazn/install/oidConfigForOc4j.sbs -oc4jAdminPwd welcome2
```

After you run this tool, your default Oracle Identity Management realm will contain the following:

- An `administrators` group
- An administrative user that is a member of the `administrators` group

The `administrators` group will have the following permissions:

- `oracle.j2ee.server.AdministrationPermission ("administration")`
- `oracle.j2ee.server.rmi.RMIPermission("login")`

Finally, you must set the `ldap.user` property to `admin` and the `ldap.password` property to the appropriate password, as discussed in "Configuring LDAP SSL Properties" on page 5-5.

Creating an anonymous User Using ldapmodify

You create an anonymous user by creating an LDIF (lightweight directory interchange format) file, then supplying the LDIF file as an input to the `ldapmodify` tool. An appropriate LDIF file is shown in Example 7–1. Note that you must replace *yourDistinguishedName* by the distinguished name of the default identity management realm.

Example 7–1 An anony.ldif file to Create anonymous User

```
dn: cn=anonymous, cn=Users, yourDistinguishedName
changetype: add
uid: anonymous
givenName: anonymous
cn: anonymous
sn: anonymous
description: This entry is used as the identification for unauthenticated users.
orclisenabled: disabled
objectClass: top
objectclass: person
objectclass: organizationalPerson
objectClass: inetorgperson
objectClass: orcluser
objectClass: orcluserV2
```

After you have created your `anony.ldif` file, use the `ldapmodify` command to add the anonymous user. The syntax for this command is:

```
ORACLE_HOME/bin/ldapmodify -D cn=orcladmin -w password -h hostname -p port \
                           -f anony.ldif
```

When you issue this command, replace *password*, *hostname*, and *port* with the password, host name, and port for your installation.

Note: The `anonymous` account is a special user account created in the Oracle Internet Directory server for OC4J server usage purpose only. Because this account is created without a password, this account cannot be used by an end user to log in to the applications

LDAP-Based Provider Environment Variables

Before beginning development, you must ensure that the operating-system-specific environment variable controlling loading of dynamic libraries (for example, `LD_LIBRARY_PATH` in Solaris) is set appropriately. See Table 2–5, " Dynamic Library Path Settings" on page 2-9 for details.

When you manage OC4J with Oracle Enterprise Manager, it sets this variable automatically.

Creating LDAP Users and Groups

To create users and groups when using the LDAP-based provider, you use the Oracle Delegated Administration Services tools. For details, see *Oracle Identity Management Guide to Delegated Administration*.

8

Configuring the XML-Based Provider

This chapter discusses performing basic user, group, and role management tasks using Oracle Enterprise Manager 10*g* Application Server Control Console and the JAZN Admintool. It is divided into the following sections:

Note: This chapter uses the term "role" because that term is used by the JAZN Admintool. A "role" is the same as a "group", which is the more commonly used term.

Creating Users

To create users in the XML-based provider, use Enterprise Manager as follows:

1. Go to the Application Server Control Console.
2. Navigate to the Security page for the appropriate OC4J instance.
3. Click the **Add User** button and follow the instructions on the pages.

Note: Do not create user names that contain slash (`/`) characters, as in `a/b/c`.

Creating Roles (Groups)

To create roles (also known as groups) in the XML-based provider, use Enterprise Manager as follows:

1. Go to the Application Server Control Console.
2. Navigate to the Security page for the appropriate OC4J instance.
3. Click the **Add Group** button and follow the instructions on the pages.

Deleting Users

To delete users in the XML-based provider, use Enterprise Manager as follows:

1. Go to the Application Server Control Console.
2. Navigate to the Security page for the appropriate OC4J instance.
3. Select a user with the radio button.
4. Click the **Remove** button and follow the instructions on the pages.

Note: The instance-level `jazn-data.xml` file must contain accounts for "`admin`" and "`anonymous`". Do not remove these accounts; if you do, the OracleAS JAAS Provider will stop working.

Deleting Roles (Groups)

To delete roles (also known as groups) in the XML-based provider, use Enterprise Manager as follows:

1. Go to the Application Server Control Console.
2. Navigate to the Security page for the appropriate OC4J instance.
3. Select a group with the radio button.
4. Click the **Remove** button and follow the instructions on the pages.

Creating Realms

To add a realm, use the JAZN Admintool. See "Admintool Overview" on page 4-3 for details on using the Admintool.

The Admintool `-addrealm` option adds a realm. It takes as arguments the realm name, the administrator name, and the administrator password. The syntax is:

```
-addrealm realm admin adminpwd adminrole
```

For example, using the XML-based provider, the administrator `martha` with password `mypass` using role `hr` would add the realm `employees` as follows:

```
java -jar jazn.jar -addrealm employees martha mypass hr
```

Deleting Realms

To delete realms, use the JAZN Admintool. See "Admintool Overview" on page 4-3 for details on using the Admintool.

The Admintool `-remrrealm` option deletes a role from the realm. It takes one arguments, `realm`, the realm name. The syntax is:

```
-remrealm realm
```

To delete a realm `foo`, type:

```
java -jar jazn.jar -remrealm foo
```

Granting Permissions

See "Granting Permissions" on page 6-1.

Revoking Permissions

To revoke permissions, use the JAZN Admintool. See"Admintool Overview" on page 4-3 for details on using the Admintool.

The `-revokeperm` option revokes the specified permission from a user or role or principal. To supply multiple words in the *`permission`* argument, enclose it in quotation marks ("*`three word permission`*"). The syntax is:

```
-revokeperm {realm {-user user|-role role} | principal_class principal_parameters}
             permission_class [permission_parameters]
```

In this syntax, *`principal_class`* is the fully qualified name of a class that implements the principal interface (such as `com.sun.security.auth.NTDomainPrincipal`) and *`principal_paramters`* is a single `String` parameter.

To revoke the `perm1` permission:

```
java -jar jazn.jar -revokeperm foo -user martha java.io.FilePermission a.txt
     read,write
```

Granting Roles (Groups)

To grant roles in the XML-based provider, use Enterprise Manager as follows:

1. Go to the Application Server Control Console.
2. Navigate to the Security page of the chosen OC4J instance.
3. Select a user with the radio button.
4. Select the checkboxes that correspond to the roles you wish to grant.
5. Click the **Apply** button.

Revoking Roles (Groups)

To grant roles in the XML-based provider, use Enterprise Manager as follows:

1. Go to the Application Server Control Console.
2. Navigate to the Security page of the chosen OC4J instance.
3. Select a user with the radio button.
4. Select the checkboxes that correspond to the roles you wish to revoke.
5. Click the **Apply** button.

Setting Persistence Mode

Persistence mode governs when changes to data are written to `jazn-data.xml`. There are three possible values for persistence:

- `NONE`

  Do not write changes to `jazn-data.xml`.

- `ALL`

  Write changes after every modification.

- `VM_EXIT` (the default)

  Write changes when the Java Virtual Machine exits.

To configure the persistence mode in the XML-based provider, you must edit the `<jazn>` element in the `jazn.xml` file by hand.

See Also:

- "Locating jazn.xml, jazn-data.xml, and the <jazn> Element" on page 4-2 for details on locating `jazn.xml`

1. Open `jazn.xml` in your text editor and go to the `<jazn>` element.
2. Edit the `persistence` attribute of the `<jazn>` element. For example, to write changes after every modification, you should edit the `<jazn>` element to look like:

```
<jazn persistence="ALL" ... other attributes >
   ...
</jazn>
```

Note: Do not change the other attributes of the `<jazn>` element.

Configuring XML Default Realm

The default realm is the realm used whenever an authentication or authorization request does not specify a realm explicitly. This attribute is not needed if you have configured only one realm in the repository. To configure the XML default realm, use the following steps:

1. Locate the file containing the `<jazn>` element (as discussed in "Locating jazn.xml, jazn-data.xml, and the <jazn> Element" on page 4-2), open the file in a text editor, and go to the `<jazn>` element within the file.
2. Edit the `default-realm` attribute of the `<jazn>` element. The syntax is:

```
<jazn provider="XML" default-realm="myrealm" ... >
```

3. For example, a `<jazn>` element that set the default-realm to Sales would look like:

```
<jazn provider="XML" default-realm="Sales" ... more attributes ... >
   ...
</jazn>
```

Note: Do not edit any `<jazn>` properties except as specified in this chapter.

Migrating Principals from the principals.xml File

Use the JAZN Admintool to migrate your data out of the `principals.xml` file. For basic information on running the JAZN Admintool, see "Admintool Overview" on page 4-3.

```
-convert filename realm
```

The `-convert` option migrates the `principals.xml` file into the specified realm of the current OracleAS JAAS Provider. The *`filename`* argument specifies the path name of the input file (typically *`ORACLE_HOME`*`/j2ee/home/config/principals.xml`).

The migration converts `principals.xml` users to JAAS users and `principals.xml` groups to JAAS roles. All permissions that were previously granted to a `principals.xml` group are mapped to the JAAS role. Users that were deactivated at the time of migration are not migrated. This ensures that no users can inadvertently gain access through the migration.

An error (either `javax.naming.AuthenticationException:Invalid username/password` or `javax.naming.NamingException:Lookup Error`) is returned if the input file contains errors.

Before you convert `principals.xml`, you must make sure that you have an administrator user that is authorized to manage realms. To do this:

1. Activate the administrative user in `principals.xml`, which is deactivated by default. Be sure to create a password for the administrator.
2. Create the realm `principals.com` with a dummy user and a dummy role. For example, in the Admintool shell you would type:

   ```
   JAZN> addrealm principals.com u1 welcome r1
   ```

   Make sure that the administrator name you used to create the realm is different from the name of the administrator in `principals.xml`. This is necessary because the convert command does not migrate duplicate users, and migrates duplicate roles by overwriting the old one.
3. Migrate `principals.xml` to the `principals.com` realm, as in:

   ```
   java -jar jazn.jar -convert config/principals.xml principals.com
   ```

4. Change the `<default-realm>` to `principals.com`.

   See Also:

   - "Setting Persistence Mode" on page 8-4

5. Stop and restart OC4J.

9 Configuring External LDAP Providers

This chapter discusses how to configure OC4J to use non-Oracle LDAP servers. It is divided into the following sections:

Notes:

- You must use JDK1.4 or later to take advantage of non-Oracle LDAP servers.
- OC4J provides a JAAS login module to authenticate and authorize against non-Oracle LDAP servers. Do not configure the non-Oracle LDAP server JAAS login module to authenticate and authorize against the Oracle Internet Directory (OID). If you did that, you would lose optimizations and integrations that are available when you use the native LDAP provider. See Chapter 7, "Configuring the LDAP-Based Provider" for how to configure Oracle Internet Directory using the native LDAP provider.

Prerequisites

Before you configure OC4J, you must complete the following prerequisites:

1. Install and configure Sun Java System Application Server (formerly iPlanet) or Active Directory.
2. Install and configure OC4J.
3. Locate the `jazn-data.xml` file associated with your OC4J instance. This is normally in the *`ORACLE_HOME`*`/j2ee/`*`instance_name`*`/config` directory. You will be editing this file using a text editor.

Note: The `jazn-data.xml` file for the OC4J `home` instance, in *`ORACLE_HOME`*`/j2ee/home/config`, serves as the default repository for JAAS login modules.

---

4. Locate the `orion-application.xml` file that controls your application. This file will normally be located in the following directory:

```
ORACLE_HOME/j2ee/instance_name/application-deployments/application_name
```

You will be editing this file using a text editor.

Notes:

- Sample login module entries for Sun Java System Application Provider and Microsoft Active Directory are in the *ORACLE_HOME*`/j2ee/home/jazn/config` directory. A non-provider-specific login module entry is provided in the file `ldap_login_module.template` in the same directory.
- Be aware of the following when you make grants to principals, in `jazn-data.xml`, when using non-Oracle LDAP servers: If you specify the name of a principal using a full Distinguished Name (DN), you must specify the DN exactly as it appears in the LDAP server, and with no white space. For example:

```
cn=jdoe,dc=us,dc=example,dc=com
```

Creating a <login-module> Element in jazn-data.xml

Each option within a `<login-module>` element corresponds to a configuration setting in the LDAP provider. The supported options are listed in Table 9–1, Table 9–2 , and Table 9–3. Unless marked (optional), all options must be explicitly specified.

Table 9–1 Login Module Provider Options

Option name	Meaning
`oracle.security.jaas.ldap.provider.url`	The URL of the LDAP provider in the format *`hostname`*`:`*`portname`*.
`oracle.security.jaas.ldap.provider.principal`	The Distinguished Name (DN) of the LDAP user that is used to connect to the LDAP server. This user must be an administrator with privileges to search users and groups, and to invoke `ldapcompare` on a user password if the target directory supports this.
`oracle.security.jaas.ldap.provider.credential`	The credential (generally a password) used to authenticate the LDAP user defined in `principal`.
`oracle.security.jaas.ldap.provider.type`	(Optional) The product name of the LDAP provider. Supported values are `iplanet`, `active directory`, and `other`. If you supply `iplanet` or `active directory`, the login module is able to infer some LDAP properties (for example, the group object class for active directory is "group") and do some optimizations.
`oracle.security.jaas.ldap.provider.connect.pool`	(Optional) A boolean to determine whether connection pooling is enabled. A `true` setting enables connection pooling; `false` disables it.
`oracle.security.jaas.ldap.lm.cache_enabled`	(Optional) A boolean to determine whether login module caching is enabled. A `true` setting (default) enables caching; `false` disables it.

Table 9–2 Login Module User Options

Option name	Meaning
`oracle.security.jaas.ldap.user.name.attribute`	The name of the LDAP attribute that uniquely identifies the name of the user. In Sun Java System Application Server, `uid`; on Active Directory, `sAMAccountName`.
`oracle.security.jaas.ldap.user.objectclass`	A list of space-separated LDAP schema object classes to represent a use. On Sun Java System Application Server, `inetOrgPerson`.
`oracle.security.jaas.ldap.user.searchbase`	A list of space-separated base distinguished names (DN) in the LDAP directory that contains users. For example: `cn=users,dc=us,dc=abc,dc=com`
`oracle.security.jaas.ldap.user.searchscope`	Specifies how deeply into the LDAP directory tree to search for users. Supported values: `subtree`, `onelevel`.

Table 9–3 Login Module Role Options

Option name	Meaning
`oracle.security.jaas.ldap.role.name.attribute`	The name of the LDAP attribute that uniquely identifies the name of the role. In iPlanet, this would be `uniqueMember`; in Active Directory, it would be `member`.
`oracle.security.jaas.ldap.role.object.class`	A list of space-separated LDAP schema object classes that is used to represent a group. On Sun Java System Application Server, `groupOfUniqueNames`. On Active Directory, `group`.
`oracle.security.jaas.ldap.role.searchbase`	A list of space-separated distinguished names (DN) in the LDAP directory that contains group. For example: `cn=groups,dc=us,dc=abc,dc=com`
`oracle.security.jaas.ldap.role.searchscope`	Specifies how deeply into the LDAP directory tree to search for roles. Supported values: `subtree`, `onelevel`.
`oracle.security.jaas.ldap.role.membership.searchscope`	Specifies how deeply into the LDAP directory tree to search for role membership. Supported values: `direct`, `nested`.
`oracle.security.jaas.ldap.role.member.attribute`	The attribute of a static LDAP group object specifying the distinguished names (DN) of the members of the group. On Sun Java System Application Server, `uniqueMember`; on Active Directory, `member`.

Sample LDIF Description

Example 9–1 contains sample declarations for a user object and role object; each of the next two sections discusses how to map those objects to an LDAP provider.

Example 9–1 Sample LDIF Defining a User and Role

```
# An example user object entry
uid= jdoe,dc=us,dc=example,dc=com
uid= jdoe
givenName=John
sn=Doe
cn=John Doe
userPassword={SSHA}zD/44JbZY33osry4mzfLn0du7nBhIIAHKDG5Fg==
uidNumber=1
gidNumber=1
homeDirectory=c:\
objectClass=top
objectClass=person
objectClass=organizationalPerson
objectClass= inetOrgPerson
objectClass=posixAccount

# An example role object entry
cn=managers,ou=groups,dc=us,dc=example,dc=com
objectClass=top
objectClass= groupOfUniqueNames
cn=managers
uniqueMember=uid=jdoe,dc=us,dc=example,dc=com
```

Configuring Sun Java System Application Server as LDAP Provider

At this release, you must configure Sun Java System Application Server as your LDAP provider by editing the `jazn-data.xml` file to add a `<login-module>` element corresponding to the Sun product. This section discusses the necessary changes.

Note: A template file containing a sample login module entry for Sun Java System Application Server is provided in the file `sample_login_module.sun` in the *`ORACLE_HOME`*`/j2ee/home/jazn/config` directory.

---

1. Open your `jazn-data.xml` file (see "Prerequisites" on page 9-1) using a text editor.
2. Locate the `<application>` element representing your application. If there is no `<application>` element, create one.
3. Locate the `<login-modules>` section within the `<application>` element. If there is no `<login-modules>` element, create one.
4. Open your `orion-application.xml` file (see "Prerequisites" on page 9-1) using a text editor.
5. Locate the `<jazn>` element within `orion-application.xml`. Set the `provider` property to `"XML"` and add a `<property>` element setting `custom.ldap.provider` to `true`. The edited `<jazn>` element should look like this:

```
<jazn provider="XML">
    <property name="custom.ldap.provider" value="true"/>
</jazn>
```

6. Restart the OC4J instance using Enterprise Manager.

SunOne Example

Suppose that your Sun Java System Application Server installation is described by the set of LDIF entries shown in Example 9–1.

The corresponding `<jazn-loginconfig>` entity is shown in Example 9–2.

Example 9–2 JAAS Login Module Configuration Corresponding to Example 9–1

```
<jazn-loginconfig>
   <application>
      <name>callerInfo</name>
      <login-modules>
         <login-module>
            <class>oracle.security.jazn.login.module.LDAPLoginModule</class>
            <control-flag>required</control-flag>
            <options>
               ... irrelevant options omitted ...
               <option>
                  <name>oracle.security.jaas.ldap.user.name.attribute</name>
                  <value>uid</value>
               </option>
               <option>
                  <name>oracle.security.jaas.ldap.user.object.class</name>
                  <value>inetOrgPerson</value>
               </option>
               <option>
                  <name>oracle.security.jaas.ldap.user.searchbase</name>
                  <value>dc=us,dc=example,dc=com</value>
               </option>
               <option>
                  <name>oracle.security.jaas.ldap.role.name.attribute</name>
                  <value>cn</value>
               </option>
               <option>
                  <name>oracle.security.jaas.ldap.role.object.class</name>
                  <value>groupOfUniqueNames</value>
               </option>
               <option>
                  <name>oracle.security.jaas.ldap.role.searchbase</name>
                  <value>ou=groups,dc=us,dc=example,dc=com</value>
               </option>
               <option>
                  <name>oracle.security.jaas.ldap.member.attribute</name>
                  <value> uniqueMember </value>
               </option>
            </options>
         </login-module>
      </login-modules>
   </application>
</jazn-loginconfig>
```

Configuring Microsoft Active Directory as LDAP Provider

At this release, you must configure Microsoft Active Directory as your LDAP provider by editing the `jazn-data.xml` file to add a `<login-module>` element corresponding to the Microsoft product. This section discusses the necessary changes.

Note: A template file containing a sample login module entry for Active Directory is provided in the file `sample_login_module.ad` in the *`ORACLE_HOME`*`/j2ee/home/jazn/config` directory.

1. Locate the `<application>` element representing your application. If there is no `<application>` element, create one.
2. Locate the `<login-modules>` section within the `<application>` element. If there is no `<login-modules>` element, create one.
3. Edit the `<option>` elements to specify appropriate values for Microsoft Active Directory. Save the edited file.
4. Open your `orion-application.xml` file (see "Prerequisites" on page 9-1) using a text editor.
5. Locate the `<jazn>` element within `orion-application.xml`. Set the `provider` property to `"XML"` and add a `<property>` element that sets `custom.ldap.provider` to `true`. The edited `<jazn>` element should look like this:

```
<jazn provider="XML">
    <property name="custom.ldap.provider" value="true"/>
</jazn>
```

6. Restart the OC4J instance using Enterprise Manager.

10 Custom Login Modules

OC4J supplies a JAAS pluggable authentication framework that conforms to the JAAS standard. With this framework, an application server and any underlying authentication services remain independent from each other, and alternative authentication services can be plugged in through JAAS login modules without requiring modifications to the application server or application code.

This chapter discusses how to write and install a `LoginModule` to be used with the OracleAS JAAS Provider. The following topics are covered:

Notes:

- Because the JAAS specification does not cover user management, when you configure your application to use a custom `LoginModule`, the use of the `UserManager` API within your application is not supported. The J2EE API, however, will continue to function within your application.
- Because JAAS login module configuration is always stored in the OC4J `home` instance `jazn-data.xml` file, you should select the XML-based provider as the security provider for your application during deployment. Use of the LDAP-based provider in conjunction with custom login modules is not supported.
- In almost all cases, you should set the JAZN property `role.mapping.dynamic` to "`true`" when you configure your application to use custom login modules. This property is "`false`" by default. Also see the discussion of this property under the `<jazn>` element in "The orion-application.xml File" on page 10-8.

Integrating Custom JAAS Login Modules

A custom JAAS `LoginModule` may be desirable when Oracle Identity Management is not available and users and roles are defined in an external repository. You can configure a `LoginModule` using the XML-based provider type. When you create a custom `LoginModule`, the following preliminary questions need to be considered.

1. **Development**: Do you want to take advantage of J2EE security constraints?
2. **Development, packaging, and deployment**: Are you using the login modules that come with J2SE 1.4? Or are you deploying custom or third-party login modules?

Note: Custom login modules are supported only with the XML-based provider.

Developing a Login Module

You can use an any JAAS-compliant `LoginModule` within the OC4J framework.

See Also:

- For general information on developing login modules, the Sun JAAS documentation at:

  `http://java.sun.com/j2se/1.4.2/docs/guide/security/jaas/JAASLMDevGuide.html`

When developing a `LoginModule`, you must consider several important issues:

- Subject-Based Authorization
- J2EE Security Authorization
- Callback Support
- Debugging Tips

Each of these is discussed in detail in its own section.

Subject-Based Authorization

When you associate a custom `LoginModule` with an application, the subject and the principals it contains are used as the sole basis for all authorization tasks, including evaluating J2EE security constraints. To ensure that all relevant principals are considered during authorization, the `LoginModule` must add the relevant principals, including all roles and groups that the authenticated user participates in, to the subject during the commit phase of the JAAS authentication process.

J2EE Security Authorization

The OracleAS JAAS Provider custom `LoginModule` framework supports the J2EE declarative security model. This means that subject-based authorization enforces the J2EE security constraints declared in deployment descriptors (`web.xml` and `ejb-jar.xml`, for example). We encourage you to take advantage of the J2EE security model whenever possible.

Callback Support

The OracleAS JAAS Provider supports the standard `javax.security.auth.callback` name (`NameCallback`) and password (`PasswordCallback`) callbacks.

Debugging Tips

When debugging your secure application, bear the following issues in mind:

- Debug Logging
- Debugging Login Modules

Debug Logging

To turn on JAAS provider debug logging, set the system property `jazn.debug.log.enable` to `true` during Java Virtual Machine (JVM) startup.

You do this by modifying the `<java-options>` settings for your OC4J instance. In Oracle Application Server, you normally manage these settings using Oracle Enterprise Manager 10*g* Application Server Control Console, which stores these settings in `opmn.xml`. From the home page of your OC4J instance, do the following:

1. Choose Administration.
2. From the Administration page, choose Server Properties.
3. From the Server Properties page, under Command Line Options, enter the option for debug logging (as described immediately following).

When running OC4J outside Oracle Application Server, you set this property using JVM command-line options. For instance, you might start standalone OC4J with a command line such as:

```
java -Djazn.debug.log.enable=true -jar oc4j.jar
```

Or you might start the Admintool shell in debug mode with the command:

```
java -Djazn.debug.log.enable=true -jar jazn.jar -shell
```

When you turn on debug logging, the OracleAS JAAS Provider logs debugging output to the console. Under Oracle Application Server, debugging output is captured in the *`ORACLE_HOME`*`/opmn/logs` directory.

Debugging Login Modules

We encourage you to include debugging options in your custom `LoginModule`. For an example, see the default login module, `RealmLoginModule`, which provides diagnostic output if `debug` is set to `true`.

Accessing EJBs When Using Custom Login Modules

To access an EJB using a custom `LoginModule`, you must:

- Grant `login` permission to the user `JDOE_ENDUSER` in the OC4J `home` instance `jazn-data.xml` file.
- Grant namespace-access to the user `JDOE_ENDUSER` in `orion-application.xml`.

To grant login permission to `JDOE_ENDUSER`, use the JAZN Admintool, as in the following example:

```
java -jar jazn.jar -grantperm login -user JDOE_ENDUSER oracle.j2ee.server.rmi.RMIPermission
```

To grant namespace access to `JDOE_ENDUSER`, edit `orion-application.xml` to add a `<namespace-access>` element like the following:

```
<namespace-access>
  <read-access>
    <namespace-resource root="">
      <security-role-mapping>
        <user name="JDOE_ENDUSER" />
      </security-role-mapping>
    </namespace-resource>
  </read-access>
</namespace-access>
```

Adding and Removing Login Modules

You use the JAZN Admintool to add and remove login modules. For basic information on running the JAZN Admintool, see "Admintool Overview" on page 4-3.

```
java -jar jazn.jar -addloginmodule application_name login_module_name
     control_flag [optionname=value ...]
java -jar jazn.jar -remloginmodule application_name login_module_name
```

The `-addloginmodule` option configures a new `LoginModule` for the named application.

The *`control_flag`* setting must be one of `required`, `requisite`, `sufficient` or `optional`, as specified in `javax.security.auth.login.Configuration`. See Table 10–1.

Table 10–1 Login Module Control Flags

Flag	Meaning
`required`	The `LoginModule` must succeed. Whether or not it succeeds, authentication proceeds down the `LoginModule` list.
`requisite`	The `LoginModule` must succeed. If it succeeds, authentication continues down the `LoginModule` list. If it fails, control immediately returns to the application (authentication does not continue down the `LoginModule` list).
`sufficient`	The `LoginModule` is not required to succeed. If it succeeds, control immediately returns to the application and authentication does not proceed down the `LoginModule` list. If it fails, authentication continues down the `LoginModule` list.
`optional`	The `LoginModule` is not required to succeed. Whether or not it succeeds, authentication proceeds down the `LoginModule` list.

If the `LoginModule` accepts its own options, you specify each option and its value as an *`optionname=value`* pair. Each `LoginModule` has its own individual set of options.

For instance, to add `MyLoginModule` to the application `myapp` as a required module with `debug` set to `true`, specify:

```
java -jar jazn.jar -addloginmodule myapp MyLoginModule required debug=true
```

To delete `MyLoginModule` from `myapp`, specify:

```
java -jar jazn.jar -remloginmodule myapp MyLoginModule
```

Admintool shell:

```
JAZN:> addloginmodule myapp MyLoginModule required debug=true
JAZN: remloginmodule myapp MyLoginModule
```

Listing Login Modules

Use the JAZN Admintool to list login modules.

See Also:

- "Admintool Overview" on page 4-3 for basic information on running the JAZN Admintool

```
java -jar jazn.jar -listloginmodules [application_name [login_module_class]]
```

The `-listloginmodules` option displays all login modules either in the specified *`application_name`* , or, if no *`application_name`* is specified, in all applications. Specifying *`login_module_class`*, after *`application_name`*, displays information on only the specified class within the application.

For example, to display all login modules for the application `myapp`, specify:

```
java -jar jazn.jar -listloginmodules myapp
```

Admintool shell:

```
JAZN:> listloginmodules myapp
```

Packaging and Deploying

If you are using one or more of the default login modules provided with J2SE 1.3 and 1.4 (such as the J2SE1.4 `com.sun.security.auth.module.Krb5LoginModule`), then no additional configuration is needed. The OracleAS JAAS Provider can locate the default login modules.

If you are deploying your application with a custom login module, then you must deploy the login module and configure the OracleAS JAAS Provider properly so that the module can be found at runtime.

The following options are available when packaging and deploying your custom login modules:

- Deploying as Standard Extensions or Optional Packages
- Deploying within the J2EE Application
- Using the OC4J Classloading Mechanism

The remainder of this section discusses these options in greater detail.

Deploying as Standard Extensions or Optional Packages

If you deploy your login modules as standard extensions, the OracleAS JAAS Provider will be able to find them. No additional configuration is necessary. Deploying login modules as standard extensions allows multiple applications to share the deployed login modules.

For example, one way to deploy your login modules as standard extensions is to deploy them to the *`ORACLE_HOME`*`/j2ee/`*`instance_name`*`/lib/ext` directory.

See Also:

- For additional information:

  `http://java.sun.com/j2se/1.4.2/docs/guide/extensions`

Deploying within the J2EE Application

If your login module is used only by a single J2EE application rather than shared among multiple applications, then you can simply package your login module as part of your application, and the OracleAS JAAS Provider will be able to find it. No additional configuration is necessary.

If a later application needs the same `LoginModule`, you must repackage the login module and any relevant classes with the new application.

If you want to enable multiple applications to share the same `LoginModule` but you cannot deploy the `LoginModule` as an extension, then you can consider using the OC4J classloading mechanism.

Using the OC4J Classloading Mechanism

The OracleAS JAAS Provider is integrated with the OC4J classloading architecture. If you configure your application so that the deployed custom login modules are part of your application `classpath`, then the OracleAS JAAS Provider can locate them.

One way to accomplish this is using the `<library>` element in either of the following files:

- `application.xml` (instance-level)
- `orion-application.xml` (application-specific)

See Also:

- *Oracle Application Server Containers for J2EE Services Guide* for more information about the `<library>` element

Configuring Your Application

You modify the following files to configure your application to take advantage of custom login modules:

- The jazn-data.xml File
- The web.xml or ejb-jar.xml File
- The orion-application.xml File
- The oc4j-ra.xml File (J2EE Connector Architecture)

This section gives details on the configuration files.

Note: You must choose the XML-based provider when using custom login modules, as discussed in "Integrating Custom JAAS Login Modules" on page 10-2.

The jazn-data.xml File

All login module configuration information is stored in the OC4J `home` instance `jazn-data.xml` file. This file is usually located in the *`ORACLE_HOME`*`/j2ee/home/config` directory.

Note: The `home` instance `jazn-data.xml` must contain accounts for "`admin`" and "`anonymous`". Do not remove these accounts; if you do, the administrative functions of the OracleAS JAAS Provider will not work.

You must modify the `home` instance `jazn-data.xml` file whenever you deploy your application into a new OC4J instance. You edit this file using the JAZN Admintool.

The following sections discuss these XML elements:

- <jazn-loginconfig>
- <jazn-policy>

<jazn-loginconfig>

This element contains information that associates applications with login modules.

Example 10–1 Example <jazn-loginconfig> Element

```
<jazn-loginconfig>
  <application>
    <name>sampleLM</name>
    <login-modules>
       <login-module>
          <class>oracle.security.jazn.samples.SampleLoginModule</class>
          <control-flag>required</control-flag>
       </login-module>
    </login-modules>
  </application>
</jazn-loginconfig>
```

This fragment associates the application `sampleLM` with the login module `sample.SampleLoginModule`.

Note: Do not remove login configuration information on `RealmLoginModule`.

<jazn-policy>

This element contains information that associates grantees with permissions. If you want to make your fat client accessible to an EJB, you must explicitly make the permissions available. When you deploy a custom `LoginModule` in OC4J, you normally use custom principal classes or types. To grant or revoke permissions to these types, use the JAZN Admintool.

Note: These policies must be set in the `home` instance `jazn-data.xml` file, in the *`ORACLE_HOME`*`/j2ee/home/config` directory.

Example 10–2 Example <jazn-policy> Element

```
<jazn-policy>
  <grant>
   <grantee>
     <principals>
      <principal>
        <class>oracle.security.jazn.samples.SampleUser</class>
        <name>admin</name>
      </principal>
    </principals>
   </grantee>
   <permissions>
     <permission>
       <class>oracle.j2ee.server.rmi.RMIPermission</class>
       <name>login</name>
     </permission>
   </permissions>
  </grant>
</jazn-policy>
```

This fragment grants the permission `oracle.j2ee.server.rmi.RMIPermission` with target name `login` to the principal with class `oracle.security.jazn.samples.SampleUser` and name `admin`.

Note: Oracle recommends that you manage the contents of `jazn-data.xml` using the JAZN Admintool.

See Also:

- Chapter 8, "Configuring the XML-Based Provider" for more information about the JAZN Admintool

The web.xml or ejb-jar.xml File

To take advantage of J2EE declarative security in your application, you must configure the appropriate security constraints, either using your IDE or by hand-editing either the `web.xml` or `ejb-jar.xml` file.

See Also:

- For details on these files, the J2EE standard documentation at:

  `http://java.sun.com/j2ee`

The orion-application.xml File

This file is a container-specific deployment descriptor that is generated for each application deployed in OC4J. The following elements are relevant to writing custom login modules:

- <jazn>
- <security-role-mapping>

Note: This section discusses only elements relevant to security.

See Also:

- *Oracle Application Server Containers for J2EE User's Guide* for more information about the `orion-application.xml` file

<jazn>

The following `<jazn>` property is specific to `LoginModule` configuration:

- `role.mapping.dynamic`

  This property, when set to `true`, instructs the OracleAS JAAS Provider to base authorization checks on the authenticated `Subject` instance instead of basing checks on the users and roles defined in the application specific `jazn-data.xml` file.

  The `LoginModule` instance (or instances) must ensure that the appropriate principals (users, roles, or groups) are associated with the `Subject` instance during the commit phase of the authentication process, in order for the principals to be taken into consideration during the authorization process. This association of principals to the `Subject` instance is typically implemented using the standard JAAS API.

```
<jazn provider="XML" location="./jazn-data.xml">
   <property name="role.mapping.dynamic" value="true" />
</jazn>
```

See Also:

- "Locating the <jazn> Element" on page 4-2 for a discussion of how to locate the `<jazn>` element

<security-role-mapping>

When you set J2EE security constraints in the `web.xml` or `ejb-jar.xml` file, you must configure security role mapping.

The optional `<security-role-mapping>` element describes static security-role mapping information. If you set J2EE security constraints in your application deployment descriptors (`web.xml` or `ejb-jar.xml`), you must configure security role mapping.

See Also:

- "Authenticating and Authorizing EJB Applications" on page 12-2

<library>

This element adds to the classpath associated with your application. For example:

```
<library path="../../shared/lib/sample.jar"/>
<library path="../../shared/lib/samplemodule.jar"/>
```

The oc4j-ra.xml File (J2EE Connector Architecture)

Each `<connector-factory>` element in `oc4j-ra.xml` can specify a different JAAS login module, as in the following example. This also shows `<config-property>` setup to connect to a database through Oracle JDBC.

```
  <connector-factory connector-name="myBlackbox" location="eis/myEIS1">
     <config-property name="connectionURL"
                      value="jdbc:oracle:thin:@localhost:5521/myservice" />
     <security-config use="jaas-module">
```

```
        <jaas-module>
           <jaas-application-name>JAASModuleDemo</jaas-application-name>
        </jaas-module>
     </security-config>
  </connector-factory>
```

Simple Login Module J2EE Integration

Developing a simple `LoginModule` follows the standard development, packaging, and deployment cycle. The following sections discuss each step in the cycle.

Development

Develop a JAAS-compliant `LoginModule` according to the JAAS SPI.

See Also:

- Javadoc for `javax.security.auth.spi.LoginModule`:

  `http://java.sun.com/j2se/1.4.2/docs/api/`

Packaging

Package your `LoginModule` classes as part of your application EAR file. For Web applications, include the classes under the `WEB-INF/classes`.

Deployment

To deploy your `LoginModule` in the `home` instance `jazn-data.xml` file:

1. Register your application login module within the `<application>` element of the `jazn-data.xml` file.

   The following entry registers the login module `oracle.security.jazn.samples.SampleLoginModule` to be used for authenticating users accessing the `sampleLM` application.

   ```
   <application>
      <name>sampleLM</name>
      <login-modules>
         <login-module>
            <class>oracle.security.jazn.samples.SampleLoginModule</class>
            <control-flag>required</control-flag>
            <options>
               <option>
                  <name>debug</name>
                  <value>true</value>
               </option>
            </options>
         </login-module>
      </login-modules>
   </application>
   ```

2. **Optional**: Grant relevant permissions to your users and roles.

   For example, if the principal `admin` needs EJB access, then you must grant the permission `oracle.j2ee.rmi.RMIPermission` to `admin`.

   ```
   <grant>
     <grantee>
       <principals>
   ```

```
        <principal>
          <class>oracle.security.jazn.samples.SampleUser</class>
          <name>admin</name>
        </principal>
      </principals>
    </grantee>
    <permissions>
      <permission>
        <class>oracle.j2ee.server.rmi.RMIPermission</class>
        <name>login</name>
      </permission>
    </permissions>
  </grant>
```

To deploy your `LoginModule` in the application-specific `orion-application.xml` file:

1. Set the `<jazn>` property `role.mapping.dynamic` to `true`:

   ```
   <jazn provider="XML" location="./jazn-data.xml" >
     <property name="role.mapping.dynamic" value="true" />
   </jazn>
   ```

2. Create appropriate `<security-role-mapping>` entries:

   ```
   <security-role-mapping name="sr_developer">
     <user name="developer" />
   </security-role-mapping>
   <security-role-mapping name="sr_manager">
     <group name="managers" />
   </security-role-mapping>
   ```

Custom Login Module Example

This section gives source code for a simple custom `LoginModule` to be used by the `CallerInfo` example. You can find the complete source code for the revised example by searching the Oracle Technology Network:

`http://www.oracle.com/technology/index.html`

Example 10–3 SampleLoginModule.java

```
package oracle.security.jazn.samples;

import java.util.Set;
import java.util.Iterator;
import java.util.Map;
import java.security.Principal;
import javax.security.auth.Subject;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.login.LoginException;
import javax.security.auth.spi.LoginModule;

public class SampleLoginModule implements LoginModule {

    // initial state
    protected Subject _subject;
    protected CallbackHandler _callbackHandler;
```

```
protected Map _sharedState;
protected Map _options;

// configuration options
protected boolean _debug;

// the authentication status
protected boolean _succeeded;
protected boolean _commitSucceeded;

// username and password
protected String _name;
protected char[] _password;

protected Principal[] _authPrincipals;


/**
 * Initialize this <code>LoginModule</code>.
 * <p/>
 * <p/>
 *
 * @param subject         the <code>Subject</code> to be authenticated. <p>
 * @param callbackHandler a <code>CallbackHandler</code> for communicating
 *                        with the end user (prompting for usernames and
 *                        passwords, for example). <p>
 * @param sharedState     shared <code>LoginModule</code> state. <p>
 * @param options         options specified in the login
 *                        <code>Configuration</code> for this particular
 *                        <code>LoginModule</code>.
 */
public void initialize(Subject subject,
                       CallbackHandler callbackHandler,
                       Map sharedState,
                       Map options) {
    this._subject = subject;
    this._callbackHandler = callbackHandler;
    this._sharedState = sharedState;
    this._options = options;

    // initialize any configured options
    _debug = "true".equalsIgnoreCase((String) _options.get("debug"));

    if (debug()) {
        printConfiguration(this);
    }
}


final public boolean debug() {
    return _debug;
}


protected Principal[] getAuthPrincipals() {
    return _authPrincipals;
}


/**
```

```
     * Authenticate the user by prompting for a username and password.
     * <p/>
     * <p/>
     *
     * @return true if the authentication succeeded, or false if this
     *         <code>LoginModule</code> should be ignored.
     * @throws FailedLoginException if the authentication fails. <p>
     * @throws LoginException       if this <code>LoginModule</code>
     *                              is unable to perform the authentication.
     */
    public boolean login() throws LoginException {
        if (debug())
            System.out.println("\t\t[SampleLoginModule] login");

        if (_callbackHandler == null)
            throw new LoginException("Error: no CallbackHandler available " +
                    "to garner authentication information from the user");

        // Setup default callback handlers.
        Callback[] callbacks = new Callback[] {
            new NameCallback("Username: "),
            new PasswordCallback("Password: ", false)
        };

        try {
            _callbackHandler.handle(callbacks);
        } catch (Exception e) {
            _succeeded = false;
            throw new LoginException(e.getMessage());
        }


        String username = ((NameCallback)callbacks[0]).getName();
        String password =
                   new String(((PasswordCallback)callbacks[1]).getPassword());
        if (debug())
        {
            System.out.println("\t\t[SampleLoginModule] username : " + username);
        }

        // Authenticate the user. On successfull authentication add principals
        // to the Subject. The name of the principal is used for authorization by
        // OC4J by mapping it to the value of the name attribute of the group
        // element in the security-role-mapping for the application.
        if(username.equals("developer") && password.equals("welcome"))
        {
            _succeeded = true;
            _name = "developer";
            _password = password.toCharArray();
            _authPrincipals = new SamplePrincipal[2];
            //Adding username as principal to the subject
            _authPrincipals[0] = new SamplePrincipal("developer");
            //Adding role developers to the subject
            _authPrincipals[1] = new SamplePrincipal("developers");
        }

        if(username.equals("manager") && password.equals("welcome"))
        {
            _succeeded = true;
            _name = "manager";
            _password = password.toCharArray();
            _authPrincipals = new SamplePrincipal[3];
            //Adding username as principal to the subject
            _authPrincipals[0] = new SamplePrincipal("manager");
            //Adding roles developers and managers to the subject
            _authPrincipals[1] = new SamplePrincipal("developers");
            _authPrincipals[2] = new SamplePrincipal("managers");
        }


        ((PasswordCallback)callbacks[1]).clearPassword();
        callbacks[0] = null;
        callbacks[1] = null;

        if (debug())
        {
            System.out.println("\t\t[SampleLoginModule] success : " + _succeeded);
        }

        if (!_succeeded)
            throw new LoginException
                            ("Authentication failed: Password does not match");

        return true;
    }


    /**
     * <p> This method is called if the LoginContext's
     * overall authentication succeeded
     * (the relevant REQUIRED, REQUISITE, SUFFICIENT and OPTIONAL LoginModules
     * succeeded).
     * <p/>
     * <p> If this LoginModule's own authentication attempt
     * succeeded (checked by retrieving the private state saved by the
     * <code>login</code> method), then this method associates a
     * <code>Principal</code>
     * with the <code>Subject</code> located in the
     * <code>LoginModule</code>.  If this LoginModule's own
     * authentication attempted failed, then this method removes
     * any state that was originally saved.
     * <p/>
     * <p/>
     *
     * @return true if this LoginModule's own login and commit
     *         attempts succeeded, or false otherwise.
     * @throws LoginException if the commit fails.
     */
    public boolean commit()
            throws LoginException {
        try {

            if (_succeeded == false) {
                return false;
            }

            if (_subject.isReadOnly()) {
                throw new LoginException("Subject is ReadOnly");
            }
            // add authenticated principals to the Subject
            if (getAuthPrincipals() != null) {
                for (int i = 0; i < getAuthPrincipals().length; i++) {
                    if(!_subject.getPrincipals().contains(getAuthPrincipals()[i]))
{
                        _subject.getPrincipals().add(getAuthPrincipals()[i]);
                    }
                }
            }

            // in any case, clean out state
            cleanup();
            if (debug()) {
                printSubject(_subject);
            }

            _commitSucceeded = true;
            return true;

        } catch (Throwable t) {
            if (debug()) {
                System.out.println(t.getMessage());
                t.printStackTrace();
            }
            throw new LoginException(t.toString());
        }
    }


    /**
     * <p> This method is called if the LoginContext's
     * overall authentication failed.
     * (the relevant REQUIRED, REQUISITE, SUFFICIENT and OPTIONAL LoginModules
     * did not succeed).
     * <p/>
     * <p> If this LoginModule's own authentication attempt
     * succeeded (checked by retrieving the private state saved by the
     * <code>login</code> and <code>commit</code> methods),
     * then this method cleans up any state that was originally saved.
     * <p/>
     * <p/>
     *
     * @return false if this LoginModule's own login and/or commit attempts
     *         failed, and true otherwise.
     * @throws LoginException if the abort fails.
     */
    public boolean abort() throws LoginException {
        if (debug()) {
            System.out.println
                    ("\t\t[SampleLoginModule] aborted authentication attempt.");
        }

        if (_succeeded == false) {
            cleanup();
            return false;
        } else if (_succeeded == true && _commitSucceeded == false) {
            // login succeeded but overall authentication failed
            _succeeded = false;
            cleanup();
        } else {
            // overall authentication succeeded and commit succeeded,
            // but someone else's commit failed
            logout();
        }
        return true;
    }


    protected void cleanup() {
        _name = null;
        if (_password != null) {
            for (int i = 0; i < _password.length; i++) {
                _password[i] = ' ';
            }
            _password = null;
        }
    }


    protected void cleanupAll() {
        cleanup();

        if (getAuthPrincipals() != null) {
            for (int i = 0; i < getAuthPrincipals().length; i++) {
                _subject.getPrincipals().remove(getAuthPrincipals()[i]);
            }
        }
    }


    /**
     * Logout the user.
     * <p/>
     * <p> This method removes the <code>Principal</code>
     * that was added by the <code>commit</code> method.
     * <p/>
     * <p/>
     *
     * @return true in all cases since this <code>LoginModule</code>
     *         should not be ignored.
     * @throws LoginException if the logout fails.
     */
    public boolean logout() throws LoginException {
        _succeeded = false;
        _commitSucceeded = false;
        cleanupAll();
        return true;
    }

    // helper methods //

    protected static void printConfiguration(SampleLoginModule slm) {
        if (slm == null) {
            return;
        }
        System.out.println("\t\t[SampleLoginModule] configuration options:");
        if (slm.debug()) {
            System.out.println("\t\t\tdebug = " + slm.debug());
        }
    }
    protected static void printSet(Set s) {
        try {
            Iterator principalIterator = s.iterator();
            while (principalIterator.hasNext()) {
                Principal p = (Principal) principalIterator.next();
                System.out.println("\t\t\t" + p.toString());
            }
        } catch (Throwable t) {
        }
    }


    protected static void printSubject(Subject subject) {
        try {
            if (subject == null) {
                return;
            }
            Set s = subject.getPrincipals();
            if ((s != null) && (s.size() != 0)) {
                System.out.println
                     ("\t\t[SampleLoginModule] added the following Principals:");
                printSet(s);
            }

            s = subject.getPublicCredentials();
            if ((s != null) && (s.size() != 0)) {
                System.out.println
              ("\t\t[SampleLoginModule] added the following Public Credentials:");
                printSet(s);
            }
        } catch (Throwable t) {
        }
    }
}
```

The `Principal` that this `LoginModule` uses is in Example 10–4.

Example 10–4 SamplePrincipal example

```
package oracle.security.jazn.samples;

import java.security.Principal;

public class SamplePrincipal implements Principal {

    private String _name = null;


    public SamplePrincipal(String name) {
        _name = name;
    }


    public boolean equals(Object another) {
        return ((SamplePrincipal)another).getName().equals(_name);
    }
    public String getName() {
        return _name;
    }


    public int hashCode() {
        return _name.hashCode();
    }


    public String toString() {
        return "[SamplePrincipal] : " + _name;
    }

}
```

11 Configuring OC4J and SSL

OC4J supports Secure Socket Layer (SSL) communication between Oracle HTTP Server and OC4J in an Oracle Application Server environment, using secure AJP. This is the secure version of Apache JServ Protocol, the protocol that Oracle HTTP Server uses to communicate with OC4J. Note, however, that the secure AJP protocol used between Oracle HTTP Server and OC4J is not visible to the end user.

This chapter discusses only configuring OC4J to take advantage of SSL; for full information about configuring other Oracle Application Server components, see the *Oracle Application Server Administrator's Guide* .

The following sections provide details:

- Overview of SSL Keys and Certificates
- Using Keys and Certificates with OC4J and Oracle HTTP Server
- Enabling SSL in OC4J
- Requesting Client Authentication
- Resolving Common SSL Problems

Note: Secure communication between a client and Oracle HTTP Server is independent of secure communication between Oracle HTTP Server and OC4J. This chapter covers only secure communication between Oracle HTTP Server and OC4J.

This chapter assumes some prior knowledge of security and SSL concepts.

See Also:

For additional information about Oracle Application Server security and Oracle HTTP Server:

- *Oracle Application Server Security Guide*
- *Oracle HTTP Server Administrator's Guide*

Overview of SSL Keys and Certificates

In SSL communication between two entities, such as companies or individuals, the server has a *public key* and an associated *private key*. Each key is a number, with the private key of an entity being kept secret by that entity, and the public key of an entity being publicized to any other parties with which secure communication might be necessary. The security of the data exchanged is guaranteed by keeping the private key secret, and by the complex encryption algorithm. This system is known as *asymmetric encryption*, because the key used to encrypt data is not the same as the key used to decrypt data.

Asymmetric encryption has a performance cost due to its complexity. A much faster system is *symmetric encryption*, where the same key is used to encrypt and decrypt data. But the weakness of symmetric encryption is that the same key has to be known by both parties, and if anyone intercepts the exchange of the key, then the communication becomes unsecure.

SSL uses both asymmetric and symmetric encryption to communicate. An asymmetric key (*PKI public key*) is used to encode a symmetric encryption key (the *bulk encryption key*); the bulk encryption key is then used to encrypt subsequent communication. After both sides agree on the bulk encryption key, faster communication is possible without losing security and reliability.

When an SSL session is negotiated, the following steps take place:

1. The server sends the client its public key.
2. The client creates a bulk encryption key, often a 128 bit RC4 key, using a specified encryption suite.
3. The client encrypts the bulk key with the server public key, and sends the encrypted bulk key to the server.
4. The server decrypts the bulk encryption key using the server private key.

This set of operations is called *key exchange*. After key exchange has taken place, the client and the server use the bulk encryption key to encrypt all exchanged data.

Note: It is possible, but rare, for the client to have its own private and public keys as well.

In SSL the public key of the server is sent to the client in a data structure known as an X.509 certificate. This certificate, created by a *certificate authority* (CA), contains a public key, information concerning the owner of the certificate, and optionally some digital rights of the owner. Certificates are digitally signed by the CA which created them using the digital certificate public key of that CA.

In SSL, the CA signature is checked by the receiving process to ensure that it is on the *approved list* of CA signatures. This check is sometimes performed by analysis of certificate chains. This occurs if the receiving process does not have the signing CA public key on the approved list. In that case the receiving process checks to see if the signer of the certificate of the CA is on the approved list or the signer of the signer, and so on. This chain of certificate, signer of certificate, signer of signer of certificate, and so on is a *certificate chain*. The highest certificate in the chain (the original signer) is called the *root certificate* of the certificate chain.

The root certificate is often on the approved list of the receiving process. Certificates in the approve list are called *trust points* or trusted certificates. A root certificate can be signed by a CA or can be *self-signed*, meaning that the digital signature that verifies the root certificate is encrypted through the private key that corresponds with the public key that the certificate contains, rather than through the private key of a higher CA. (Note that certificates of the CAs themselves are always self-signed.)

Functionally, a certificate acts as a container for public keys and associated signatures. A single certificate file can contain one or multiple chained certificates, up to an entire chain. Private keys are normally kept separately to prevent them from being inadvertently revealed, although they can be included in a separate section of the certificate file for convenient portability between applications.

A *keystore* is used to store certificates, including the certificates of all trusted parties, for use by a program. Through its keystore, an entity such as OC4J (for example) can authenticate other parties as well as authenticate itself to other parties. The keystore password is obfuscated. Oracle HTTP Server has what is called a *wallet* for the same purpose. The Sun Microsystems SSL implementation introduces the notion of a *truststore*, which is a keystore file that includes the trusted certificate authorities that a client will implicitly accept during an SSL handshake.

In Java, a keystore is a `java.security.KeyStore` instance that you can create and manipulate using the `keytool` utility that is provided with the Sun Microsystems JDK. The underlying physical manifestation of this object is a file.

See Also:

- For information about `keytool`:

  `http://java.sun.com/j2se/1.3/docs/tooldocs/win32/keyto ol.html`

Using Keys and Certificates with OC4J and Oracle HTTP Server

The following steps describe using keys and certificates for SSL communication in OC4J. These are server-level steps, typically executed prior to deployment of an application that will require secure communication, perhaps when you first set up an Oracle Application Server instance.

Note that a *keystore* stores certificates, including the certificates of all trusted parties, for use by a program. Through its keystore, an entity such as OC4J (for example) can authenticate other parties, as well as authenticate itself to other parties. Oracle HTTP Server uses what is called a *wallet* for the same purpose.

In Java, a keystore is a `java.security.KeyStore` instance that you can create and manipulate using the `keytool` utility that is provided with the Sun Microsystems JDK. The underlying physical manifestation of this object is a file.

The Oracle Wallet Manager has functionality for Oracle wallets that is equivalent to the functionality of `keytool` for keystores.

See Also:

- *Oracle Application Server Administrator's Guide* for information on Oracle Wallet Manager

Here are the steps in using certificates between OC4J and Oracle HTTP Server:

1. Use `keytool` to generate a private key, public key, and unsigned certificate.You can place this information into either a new keystore or an existing keystore.
2. Obtain a signature for the certificate, using either of the following two approaches.

   Generate your own signature:

   a. Use `keytool` to "self-sign" the certificate. This is appropriate if your clients trust you as, in effect, your own certificate authority.

   Alternatively, obtain a signature from a recognized certificate authority:

   a. Using the certificate from Step 1, use `keytool` to generate a *certificate request*, which is a request to have the certificate signed by a certificate authority.

b. Submit the certificate request to a certificate authority.

c. Receive the signature from the certificate authority, and import it into the keystore, again using `keytool`. In the keystore, the signature is matched with the associated certificate.

Note: Oracle Application Server includes Oracle Application Server Certificate Authority (OCA). OCA enables customers to create and issue certificates for themselves and their users, although these certificates would probably be unrecognized outside a customer's organization without prior arrangements.

See Also:

- *Oracle Application Server Certificate Authority Administrator's Guide* for information about OCA

The process for requesting and receiving signatures is up to the particular certificate authority you use. Because that is outside the scope and control of Oracle Application Server, the documentation does not cover it. You can go to the Web site of any certificate authority for information. (Any browser should have a list of trusted certificate authorities.) Here are the Web addresses for VeriSign, Inc. and Thawte, Inc., for example:

`http://www.verisign.com/`

`http://www.thawte.com/`

For SSL communication between OC4J and Oracle HTTP Server, execute the preceding steps for Oracle HTTP Server, but use a wallet and Oracle Wallet Manager instead of a keystore and the `keytool` utility.

See Also:

- *Oracle Application Server Administrator's Guide* for information about wallets and the Oracle Wallet Manager

In addition to steps 1 and 2 above, execute the following steps as necessary:

1. **If the OC4J certificate is signed by an entity that Oracle HTTP Server does not yet trust**, obtain the certificate of the entity and import it into Oracle HTTP Server. The specifics depend on whether the OC4J certificate in question is self-signed, as follows.

   If OC4J has a self-signed certificate (essentially, Oracle HTTP Server does not yet trust OC4J):

   a. From OC4J, use `keytool` to export the OC4J certificate. This step places the certificate into a file that is accessible to Oracle HTTP Server.

   b. From Oracle HTTP Server, use Oracle Wallet Manager to import the OC4J certificate.

   Alternatively, if OC4J has a certificate that is signed by another entity (that Oracle HTTP Server does not yet trust):

   a. Obtain the certificate of the entity in any appropriate way, such as by exporting it from the entity. The exact steps vary widely, depending on the entity.

   b. From Oracle HTTP Server, use Oracle Wallet Manager to import the certificate of the entity.

2. **If the Oracle HTTP Server certificate is signed by an entity that OC4J does not yet trust, and OC4J is in a mode of operation that requires client authentication** (as "Requesting Client Authentication" on page 11-8 discusses):

   a. Obtain the certificate of the entity in any appropriate way, such as by exporting it from the entity. The exact steps vary widely, depending on the entity.

   b. From OC4J, use `keytool` to import the certificate of the entity.

> **Note:** During communications over SSL between Oracle HTTP Server and OC4J, all data on the communications channel between the two is encrypted. The following steps are executed:
>
> 1. The OC4J certificate chain is authenticated to Oracle HTTP Server during establishment of the encrypted channel.
> 2. Optionally, if OC4J is in client-authentication mode, Oracle HTTP Server is authenticated to OC4J. This process also occurs during establishment of the encrypted channel.
> 3. Any further communication after this initial exchange will be encrypted.

Example: Creating an SSL Certificate and Generating Your Own Signature This example corresponds to the step of obtaining a signature for the certificate, in the mode where you generate your own signature by using `keytool` to self-sign the certificate.

First, create a keystore with an RSA private/public keypair, using the `keytool` command. The following example (in which `%` is the system prompt) uses the RSA keypair algorithm to generate a keystore to reside in a file named `mykeystore`, which has a password of "`123456`" and is valid for 21 days:

```
% keytool -genkey -keyalg "RSA" -keystore mykeystore -storepass 123456
          -validity 21
```

Note the following:

- The `keystore` option specifies the name of the file in which the keys are stored.
- The `storepass` option sets the password for protecting the keystore.
- The `validity` option sets the number of days for which the certificate is valid.

The `keytool` prompts you for more information, as follows:

```
What is your first and last name?
  [Unknown]:  Test User
What is the name of your organizational unit?
  [Unknown]:  Support
What is the name of your organization?
  [Unknown]:  Oracle
What is the name of your City or Locality?
  [Unknown]:  Redwood Shores
What is the name of your State or Province?
  [Unknown]:  CA
What is the two-letter country code for this unit?
  [Unknown]:  US
Is <CN=Test User, OU=Support, O=Oracle, L=Reading, ST=Berkshire, C=GB> correct?
```

```
  [no]:  yes

Enter key password for <mykey>
        (RETURN if same as keystore password):
```

Note: To determine your two-letter country code, use the ISO country code list at the following URL:

http://www.bcpl.net/~j1m5path/isocodes.html

The `mykeystore` file is created in the current directory. The default alias of the key is `mykey`.

Enabling SSL in OC4J

For secure communication between Oracle HTTP Server and OC4J, configuration steps are required at each end, as discussed in the following section.

Configuring Oracle HTTP Server for SSL

In Oracle HTTP Server, verify proper SSL settings in the `mod_oc4j.conf` file for secure communication. SSL must be enabled, with a wallet file and password specified, as follows:

```
Oc4jEnableSSL on
Oc4jSSLWalletFile wallet_path
Oc4jSSLWalletPassword pwd
```

The *`wallet_path`* value is a directory path to the wallet file, without a file name. (The wallet file name is already known.) The *`pwd`* value is the wallet password.

See Also:

- *Oracle HTTP Server Administrator's Guide* for more information about the `mod_oc4j.conf` file

Example 11–1 Creating an SSL Certificate and Configuring HTTPS

The following example uses `keytool` to create a test certificate and shows all of the XML configuration necessary for HTTPS to work. To create a valid certificate for use in production environments, see the `keytool` documentation.

1. Install the correct JDK.

   Ensure that JDK 1.3.x is installed. This is required for SSL with OC4J. Set the *`JAVA_HOME`* to the JDK 1.3 directory. Ensure that the JDK 1.3.x *`JAVA_HOME/bin`* is at the beginning of your path. This may be achieved by doing the following.

   For UNIX:

   ```
   $ PATH=/usr/opt/java130/bin:$PATH
   $ export $PATH
   $ java -version
   java version "1.3.0"
   ```

   For Windows:

   ```
   set PATH=d:\jdk131\bin;%PATH%
   ```

Ensure that this JDK version is set as the current version in your Windows registry. In the Windows Registry Editor under `HKEY_LOCAL_MACHINE/SOFTWARE/JavaSoft/Java Development Kit`, set `CurrentVersion` to 1.3 (or later).

2. Request a certificate.

   a. Change to the *ORACLE_HOME*/`j2ee` directory.

   b. Create a keystore with an RSA private/public keypair using the `keytool` command. In this example, we generate a keystore to reside in a file named `mykeystore`, which has a password of "`123456`" and is valid for 21 days, using the RSA key pair generation algorithm with the following syntax:

```
keytool -genkey -keyalg "RSA" -keystore mykeystore -storepass 123456
        -validity 21
```

In this tool:

- The `keystore` option sets the filename where the keys are stored.
- The `storepass` option sets the password for protecting the keystore.
- The `validity` option sets number of days the certificate is valid.

The `keytool` prompts you for more information, as follows:

```
What is your first and last name?
  [Unknown]:  Test User
What is the name of your organizational unit?
  [Unknown]:  Support
What is the name of your organization?
  [Unknown]:  Oracle
What is the name of your City or Locality?
  [Unknown]:  Redwood Shores
What is the name of your State or Province?
  [Unknown]:  CA
What is the two-letter country code for this unit?
  [Unknown]:  US
Is <CN=Test User, OU=Support, O=Oracle, L=Reading, ST=Berkshire, C=GB> correct?
  [no]:  yes

Enter key password for <mykey>
        (RETURN if same as keystore password):
```

Note: To determine your two-letter country code, use the ISO country code list at the following URL:

`http://www.bcpl.net/~j1m5path/isocodes.html`

The `mykeystore` file is created in the current directory. The default alias of the key is `mykey`.

3. If you do not have a `secure-web-site.xml` file, then copy the `default-web-site.xml` file to create a *ORACLE_HOME*/`j2ee/home/config/secure-web-site.xml` file.

4. Edit `secure-web-site.xml` with the following elements:

   a. Add `secure="true"` to the `<web-site>` element, as follows:

```
<web-site port="8888"
     display-name="Default OracleAS Containers for J2EE Web Site"
```

```
        secure="true">
```

b. Add the following new line inside the `<web-site>` element to define the keystore and the password.

```
<ssl-config keystore="<yourkeystore>"
            keystore-password="<yourpassword>" />
```

Where *`yourkeystore`* is the full path to the keystore and *`yourpassword`* is the keystore password. In our example, this is as follows:

```
<!-- Enable SSL -->
<ssl-config keystore="../../keystore" keystore-password="123456"/>
```

Note: The keystore path is relative to where the XML file resides.

---

c. Change the web-site port number, to use an available port. For example, the default for SSL ports is 443, so change the Web site port attribute to port="4443". To use the default of 443, you have to be a super user.

d. Now save the changes to `secure-web-site.xml`.

5. Edit `server.xml` to point to the `secure-web-site.xml` file (if it does not already).

a. Uncomment or add the following line in the file `server.xml` so that the `secure-web-site.xml` file is read:

```
<web-site path="./secure-web-site.xml" />
```

Note: Even on Windows, you use a forward slash, not a backslash, in the XML files.

---

b. Save the changes to `server.xml`.

6. Stop and restart OC4J to initialize the `secure-web-site.xml` file additions. Test the SSL port by accessing the site in a browser on the SSL port. If successful, you will be asked to accept the certificate, because it is not signed by an accepted authority.

When completed, OC4J listens for SSL requests on one port and non-SSL requests on another. You can disable either SSL requests or non-SSL requests by commenting out the appropriate `*-web-site.xml` file in the `server.xml` configuration file:

```
<!-- Comment out the following to remove SSL -->
<web-site path="./secure-web-site.xml" />
<!-- Comment out the following to remove non-SSL -->
<default-site path="./default-web-site.xml" />
```

Requesting Client Authentication

OC4J supports a *client authentication* mode in which the server explicitly requests authentication from the client before the server communicates with the client. In an Oracle Application Server environment, Oracle HTTP Server acts as the client to OC4J.

For client authentication, Oracle HTTP Server must have its own certificate and must authenticate itself by sending a certificate and a certificate chain that ends with a root certificate. You can configure OC4J to accept only root certificates from a specified list in establishing a chain of trust back to a client.

A certificate that OC4J trusts is called a *trust point*. In the certificate chain from Oracle HTTP Server, the trust point is the first certificate OC4J encounters that matches one in its own keystore. There are three ways to establish trust:

- The client certificate is in the keystore.
- One of the intermediate CA certificates in the certificate chain from Oracle HTTP Server is in the keystore.
- The root CA certificate in the certificate chain from Oracle HTTP Server is in the keystore.

OC4J verifies that the entire certificate chain, up to and including the trust point, is valid to prevent any forged certificates.

If you request client authentication with the setting `needs-client-auth="true"` in the `<ssl-config>` element, perform the following steps:

1. Decide which of the certificates in the chain from Oracle HTTP Server is to be your trust point. Ensure that you either have control over the issuance of certificates using this trust point or that you trust the certificate authority as an issuer.
2. Import the intermediate or root certificate in the server keystore as a trust point for authentication of the client certificate.

   Note: If you do not want OC4J to accept certain trust points, make sure these trust points are not in the keystore.

3. Execute the steps to create the client certificate (as discussed in "Using Keys and Certificates with OC4J and Oracle HTTP Server" on page 11-3). The client certificate includes the intermediate or root certificate that is installed in the server. If you wish to trust another certificate authority, obtain a certificate from that authority.
4. Save the certificate in a file on Oracle HTTP Server.

   Note: If you are running standalone OC4J, save the certificate on the client.

5. Provide the certificate.
   a. If you are running Oracle HTTP Server, then provide the certificate for the Oracle HTTP Server initiation of the secure AJP connection.
   b. If you are running OC4J in a standalone environment:
      – If the client is a browser, set the certificate in the client browser security area.
      – If the client is a Java client, you must programmatically present the client certificate and the certificate chain when initiating the HTTPS connection.

During secure communication between the client and OC4J, the following functionality is executed:

- The link (all communications) between the two is encrypted.

- OC4J is authenticated to the client. A "secret key" is securely exchanged and used for the encryption of the link.
- Optionally, if OC4J is in client-authentication mode, the client is authenticated to OC4J.

See Also:

- *Oracle Application Server Administrator's Guide* for information about Oracle Wallet Manager, PKI, and security fundamentals
- Documentation for JSSE:

  `http://java.sun.com/products/jsse/doc/guide/API_users_guide.html`
- Documentation for the `java.net` package:

  `http://java.sun.com/j2se/1.4.2/docs/api/`

Resolving Common SSL Problems

This section discusses some common SSL errors and their causes and remedies, followed by a brief discussion of general SSL debugging.

Common SSL Errors and Solutions

The following errors may occur when using SSL certificates:

Keytool Error: java.security.cert.CertificateException: Unsupported encoding

Cause: There is trailing white space, which the `keytool` utility does not allow.

Action: Delete all trailing white space. If the error still occurs, add a newline in your certificate reply file.

Keytool Error: KeyPairGenerator not available

Cause: You are probably using the `keytool` utility from an older JDK.

Action: Use the `keytool` utility from the latest supported JDK. To ensure that you are using the latest JDK, specify the full path for this JDK.

Keytool Error: Failed to establish chain from reply

Cause: The `keytool` utility cannot locate the root CA certificates in your keystore, and therefore cannot build the certificate chain from your server key to the trusted root certificate authority.

Action: Execute the following command:

```
keytool -keystore keystore -import -alias cacert -file cacert.cer
(keytool -keystore keystore -import -alias intercert -file inter.cer)
```

If you use an intermediate CA `keytool` utility, then execute this command:

```
keystore keystore -genkey -keyalg RSA -alias serverkey
keytool -keystore keystore -certreq -file my.host.com.csr
```

Get the certificate from the Certificate Signing Request (CSR), then execute the following command:

```
keytool -keystore keystore -import -file my.host.com.cer -alias serverkey
```

No available certificate corresponds to the SSL cipher suites that are enabled

Cause: Something is wrong with your certificate.

Action: Determine and rectify the problem.

General SSL Debugging

While you are developing in OC4J standalone, you can display verbose debug information from the Java Secure Socket Extension (JSSE) implementation. To get a list of options, start OC4J as follows (where `%` is the system prompt):

```
% java -Djavax.net.debug=help -jar oc4j.jar
```

Start it as follows to enable full verbosity:

```
% java -Djavax.net.debug=all -jar oc4j.jar
```

This will display the browser request header, server HTTP header, server HTTP body, content length (before and after encryption), and SSL version.

12 Configuring EJB Security

This chapter discusses security issues affecting EJBs. It discusses the following topics:

- EJB JNDI Security Properties
- Configuring Security

For full information about EJBs, see the *Oracle Application Server Containers for J2EE Enterprise JavaBeans Developer's Guide.*

EJB JNDI Security Properties

There are two JNDI properties that are specific to security. You can either set these within the `jndi.properties` file or within your EJB implementation.

JNDI Properties in jndi.properties

If setting the JNDI properties within the `jndi.properties` file, set the properties as follows. Make sure that this `jndi.properties` file is accessible from the classpath.

When you access EJBs in a *remote* container, you must pass valid credentials to this container. Stand-alone clients define their credentials in the `jndi.properties` file deployed with the client code.

```
java.naming.security.principal=username
java.naming.security.credentials=password
```

JNDI Properties within Code Implementation

As in the preceding section, set the principal user name and credentials, but in your Java code. For example, JavaBeans running within the container pass their credentials within the `InitialContext`, which is created to look up the remote EJBs.

For instance, to pass JNDI security properties within the `Hashtable` environment, set these as shown in the following example:

```
Hashtable env = new Hashtable();
env.put("java.naming.provider.url", "ormi://myhost/ejbsamples");
env.put("java.naming.factory.initial",
      "oracle.j2ee.server.ApplicationClientInitialContextFactory");
env.put(Context.SECURITY_PRINCIPAL, "guest");
env.put(Context.SECURITY_CREDENTIALS, "welcome");
Context ic = new InitialContext (env);
Object homeObject = ic.lookup("java:comp/env/employeeBean");

// Narrow the reference to a TemplateHome.
EmployeeHome empHome =
   (EmployeeHome) PortableRemoteObject.narrow(homeObject,
   EmployeeHome.class);
```

Note: `ApplicationClientInitialContextFactory` is in the file `oc4jclient.jar`.

Configuring Security

EJB security involves two realms: granting permissions if you download into a browser and configuring your application for authentication and authorization. This section covers the following:

- Granting Permissions in Browser
- Authenticating and Authorizing EJB Applications
- Specifying Credentials in EJB Clients

Granting Permissions in Browser

If you download the EJB application as a client where the security manager is active, you must grant the following permissions before you can execute:

```
permission java.net.SocketPermission "*:*", "connect,resolve";
permission java.lang.RuntimePermission "createClassLoader";
permission java.lang.RuntimePermission "getClassLoader";
permission java.util.PropertyPermission "*", "read";
permission java.util.PropertyPermission "LoadBalanceOnLookup", "read,write";
```

Authenticating and Authorizing EJB Applications

For EJB authentication and authorization, you define the principals under which each method executes by configuring of the EJB deployment descriptor. The container enforces that the user who is trying to execute the method is the same as defined within the deployment descriptor.

The EJB deployment descriptor enables you to define security roles under which each method is allowed to execute. These methods are mapped to users or groups in the OC4J-specific deployment descriptor. The users and groups are defined within your designated security user managers, which uses either the JAZN or XML user manager.

See Also:

- *Oracle Application Server Containers for J2EE User's Guide* and *Oracle Application Server Containers for J2EE Services Guide* for a description of security user managers

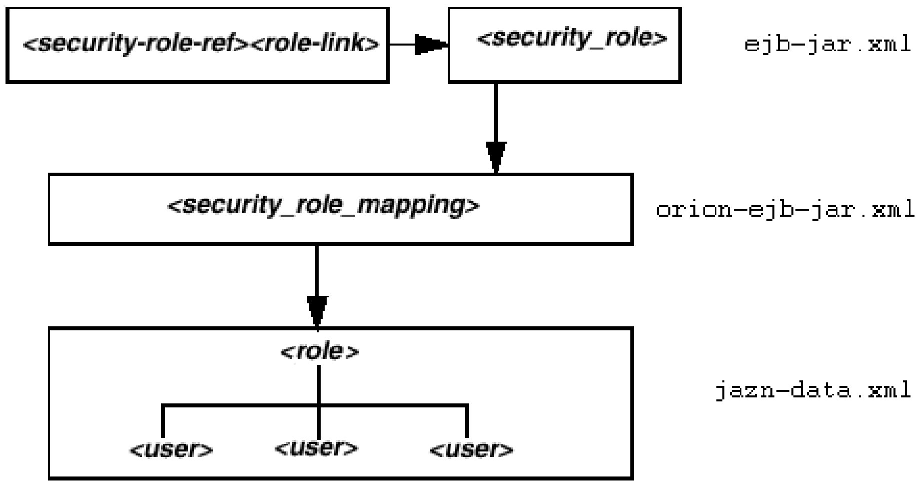
For authentication and authorization, this section focuses on XML configuration within the EJB deployment descriptors. EJB authorization is specified within the EJB and OC4J-specific deployment descriptors. You can manage the authorization piece of your security within the deployment descriptors, as follows:

- The EJB deployment descriptor describes access rules using logical roles.
- The OC4J-specific deployment descriptor maps the logical roles to concrete users and groups, which are defined either the JAZN or XML user managers.

Users and groups are identities known by the container. Roles are the *logical* identities each application uses to indicate access rights to its different objects. The user name / password pairs can be digital certificates and, in the case of SSL, private key pairs.

Thus, the definition and mapping of roles is demonstrated in Figure 12–1.

Figure 12–1 Role Mapping



Defining users, groups, and roles are discussed in the following sections:

- Specifying Users and Groups
- Specifying Logical Roles in the EJB Deployment Descriptor
- Specifying Unchecked Security for EJB Methods
- Specifying the run-as Security Identity
- Mapping Logical Roles to Users and Groups
- Specifying a Default Role Mapping for Undefined Methods
- Specifying Users and Groups by the Client

Specifying Users and Groups

OC4J supports the definition of users and groups—either shared by all deployed applications or specific to given applications. You define shared or application-specific users and groups within either the JAZN or XML user managers.

See Also:

- *Oracle Application Server Containers for J2EE User's Guide* and *Oracle Application Server Containers for J2EE Services Guide* for specific instructions

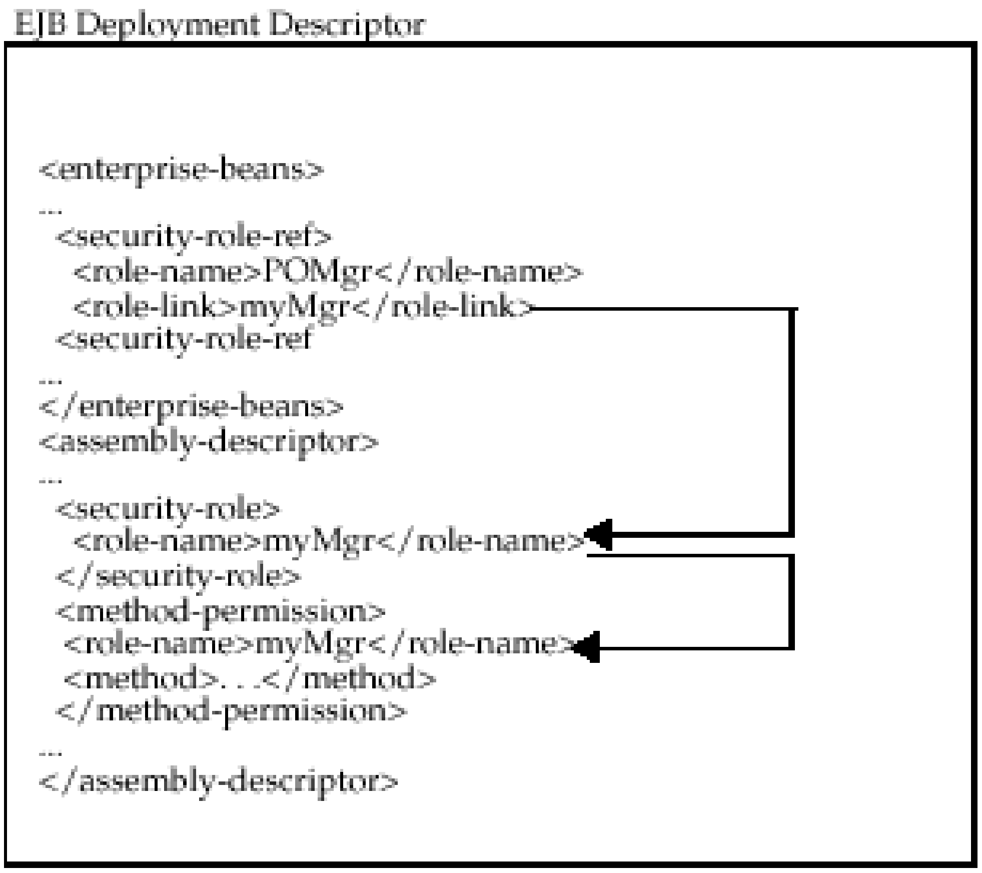
Specifying Logical Roles in the EJB Deployment Descriptor

As shown in Figure 12–2, you can use a logical name for a role within your bean implementation, and map this logical name to the correct security role or user. The mapping of the logical name to a database role is specified in the OC4J-specific deployment descriptor.

See Also:

- "Mapping Logical Roles to Users and Groups" on page 12-7

Figure 12–2 Security Mapping



```
<enterprise-beans>
...
 <security-role-ref>
   <role-name>POMgr</role-name>
   <role-link>myMgr</role-link>
 <security-role-ref
...
</enterprise-beans>
<assembly-descriptor>
...
 <security-role>
   <role-name>myMgr</role-name>
 </security-role>
 <method-permission>
  <role-name>myMgr</role-name>
  <method>. . .</method>
 </method-permission>
...
</assembly-descriptor>
```

If you use a logical name for a database role within your bean implementation for methods such as `isCallerInRole`, you can map the logical name to an actual database role by doing the following:

1. Declare the logical name within the `<enterprise-beans>` section `<security-role-ref>` element. For example, to define a role used within the purchase order example, you may have checked, within the bean implementation, to see if the caller had authorization to sign a purchase order. Thus, the caller would have to be signed in under a correct role. In order for the bean to not need to be aware of database roles, you can check `isCallerInRole` on a logical name, such as `POMgr`, because only purchase order managers can sign off on the order. Thus, you would define the logical security role, `POMgr` in the `<role-name>` element within the `<enterprise-beans>` section, as follows:

```
<enterprise-beans>
...
  <security-role-ref>
    <role-name>POMgr</role-name>
    <role-link>myMgr</role-link>
  </security-role-ref>
</enterprise-beans>
```

The <role-link> element within the <security-role-ref> element can be the actual database role, which is defined further within the <assembly-descriptor> section. Alternatively, it can be another logical name, which is still defined more in the <assembly-descriptor> section and is mapped to an actual database role within the Oracle-specific deployment descriptor.

Note: The <security-role-ref> element is not generally required. You specify it when using security context methods within your bean.

---

2. Define the role and the methods that it applies to. In the purchase order example, any method executed within the PurchaseOrder bean must have authorized itself as myMgr. Note that PurchaseOrder is the name declared in the <ejb-name> element.

   Thus, the following defines the role as myMgr, the EJB as PurchaseOrder, and all methods by denoting the "*" symbol.

Note: The myMgr role in the <security-role> element is the same as the <role-link> element within the <enterprise-beans> section. This ties the logical name of POMgr to the myMgr definition.

```
<assembly-descriptor>
  <security-role>
    <description>Role needed purchase order authorization</description>
    <role-name>myMgr</role-name>
  </security-role>
  <method-permission>
    <role-name>myMgr</role-name>
    <method>
      <ejb-name>PurchaseOrder</ejb-name>
      <method-name>*</method-name>
    </method>
  </method-permission>
...
</assembly-descriptor>
```

After performing both steps, you can refer to POMgr within the bean implementation and the container translates POMgr to myMgr.

Note: If you define different roles within the <method-permission> element for methods in the same EJB, the resulting permission is a union of all the method permissions defined for the methods of this bean.

The <method> subelement of the <method-permission> element is used to specify the security role for one or more methods within an interface or implementation. According to the EJB specification, this definition can be of one of the following forms:

1. Defining all methods within a bean by specifying the bean name and using the "*" character to denote all methods within the bean, as follows:

```
<method-permission>
  <role-name>myMgr</role-name>
  <method>
    <ejb-name>EJBNAME</ejb-name>
    <method-name>*</method-name>
  </method>
</method-permission>
```

2. Defining a specific method that is uniquely identified within the bean. Use the appropriate interface name and method name, as follows:

```
<method-permission>
  <role-name>myMgr</role-name>
  <method>
    <ejb-name>myBean</ejb-name>
    <method-name>myMethodInMyBean</method-name>
  </method>
</method-permission>
```

> **Note:** If there are multiple methods with the same overloaded name, the element of this style refers to all the methods with the overloaded name.

3. Defining a method with a specific signature among many overloaded versions, as follows:

```
<method-permission>
  <role-name>myMgr</role-name>
  <method>
    <ejb-name>myBean</ejb-name>
    <method-name>myMethod</method-name>
    <method-params>
      <method-param>javax.lang.String</method-param>
      <method-param>javax.lang.String</method-param>
    </method-params>
  </method>
</method-permission>
```

The parameters are the fully-qualified Java types of the input parameters of the method. If the method has no input arguments, the `<method-params>` element contains no elements.

Specifying Unchecked Security for EJB Methods

If you want certain methods to not be checked for security roles, you define these methods as unchecked, as follows:

```
<method-permission>
  <unchecked/>
  <method>
     <ejb-name>EJBNAME</ejb-name>
     <method-name>*</method-name>
  </method>
</method-permission>
```

Instead of a `<role-name>` element defined, you define an `<unchecked>` element. When executing any methods in the `EJBNAME` bean, the container does not check for security. Unchecked methods always override any other role definitions.

Specifying the run-as Security Identity

You can specify that all methods of an EJB execute under a specific identity. That is, the container does not check different roles for permission to run specific methods; instead, the container executes all of the EJB methods under the specified security identity. You can specify a particular role or the caller identity as the security identity.

Specify the run-as security identity in the `<security-identity>` element, which is contained in the `<enterprise-beans>` section. The following XML demonstrates that the `POMgr` is the role under which all the entity bean methods execute.

```
<enterprise-beans>
  <entity>
  ...
    <security-identity>
       <run-as>
         <role-name>POMgr</role-name>
       </run-as>
    </security-identity>
  ...
  </entity>
</enterprise-beans>
```

Alternatively, the following XML example demonstrates how to specify that all methods of the bean execute under the identity of the caller:

```
<enterprise-beans>
  <entity>
    ...
    <security-identity>
       <use-caller-identity/>
    </security-identity>
    ...
  </entity>
</enterprise-beans>
```

Mapping Logical Roles to Users and Groups

You can use logical roles or actual users and groups in the EJB deployment descriptor. However, if you use logical roles, you must map them to the actual users and groups defined either in the JAZN or XML user managers.

Map the logical roles defined in the application deployment descriptors to JAZN or XML user manager users or groups through the `<security-role-mapping>` element in the OC4J-specific deployment descriptor.

- The `name` attribute of this element defines the logical role that is to be mapped.
- The `<group>` or `<user>` element maps the logical role to a group or user name. This group or user must be defined in the JAZN or XML user manager configuration.

See Also:

- *Oracle Application Server Containers for J2EE User's Guide* and *Oracle Application Server Containers for J2EE Services Guide* for a description of the JAZN and XML user managers

Example 12–1 Mapping Logical Role to Actual Role

This example maps the logical role `POMGR` to the `managers` group in the `orion-ejb-jar.xml` file. Any user that can log in as part of this group is considered to have the `POMGR` role; thus, it can execute the methods of `PurchaseOrderBean`.

```
<security-role-mapping name="POMGR">
  <group name="managers" />
</security-role-mapping>
```

Note: You can map a logical role to a single group or to several groups.

To map this role to a specific user, do the following:

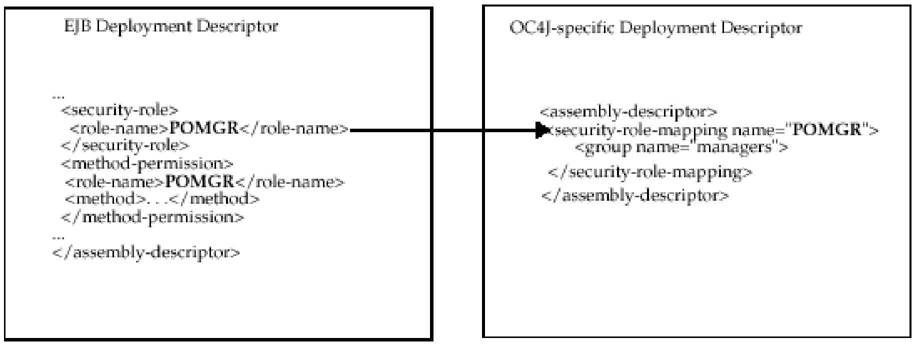
```
<security-role-mapping name="POMGR">
  <user name="guest" />
</security-role-mapping>
```

Lastly, you can map a role to a specific user within a specific group, as follows:

```
<security-role-mapping name="POMGR">
  <group name="managers" />
  <user name="guest" />
</security-role-mapping>
```

As shown in Figure 12–3, the logical role name for `POMGR` defined in the EJB deployment descriptor is mapped to `managers` within the OC4J-specific deployment descriptor in the `<security-role-mapping>` element.

Figure 12–3 Security Mapping



Notice that the `<role-name>` in the EJB deployment descriptor is the same as the name attribute in the `<security-role-mapping>` element in the OC4J-specific deployment descriptor. This is what identifies the mapping.

Specifying a Default Role Mapping for Undefined Methods

If any methods have not been associated with a role mapping, they are mapped to the default security role through the `<default-method-access>` element in the `orion-ejb-jar.xml` file. The following is the automatic mapping for any unsecure methods:

```
<default-method-access>
   <security-role-mapping name="&lt;default-ejb-caller-role&gt;"
                          impliesAll="true" />
</default-method-access>
```

The default role is `<default-ejb-caller-role>` and is defined in the `name` attribute. You can replace this string with any name for the default role. The `impliesAll` attribute indicates whether any security role checking occurs for these methods. This attribute defaults to true, which states that no security role checking occurs for these methods. If you set this attribute to false, the container will check for this default role on these methods.

If the `impliesAll` attribute is set to "`false`", you must map the default role defined in the `name` attribute to a JAZN or XML user or group through the `<user>` and `<group>` elements. The following example shows how all methods not associated with a method permission are mapped to the "`others`" group.

```
<default-method-access>
   <security-role-mapping name="default-role" impliesAll="false">
      <group name="others" />
   </security-role-mapping>
</default-method-access>
```

Specifying Users and Groups by the Client

In order for the client to access methods that are protected by users and groups, the client must provide the correct user or group name with a password that the JAZN or XML user manager recognizes. And the user or group must be the same one as designated in the security role for the intended method.

See Also:

- "Specifying Credentials in EJB Clients" immediately following

Specifying Credentials in EJB Clients

When you access EJBs in a *remote* container, you must pass valid credentials to this container.

- Stand-alone clients define their credentials in the `jndi.properties` file deployed with the EAR file.
- Servlets or JavaBeans running within the container pass their credentials within the `InitialContext`, which is created to look up the remote EJBs.

Credentials in JNDI Properties

Indicate the user name (principal) and password (credentials) to use when looking up remote EJBs in the `jndi.properties` file.

For example, if you want to access remote EJBs as `POMGR/welcome`, define the following properties. The `factory.initial` property indicates that you will use the Oracle JNDI implementation:

```
java.naming.security.principal=POMGR
java.naming.security.credentials=welcome
java.naming.factory.initial=
   oracle.j2ee.server.ApplicationClientInitialContextFactory
java.naming.provider.url=ormi://myhost/ejbsamples
```

In your application program, authenticate and access the remote EJBs, as shown in the following example:

```
InitialContext ic = new InitialContext();
CustomerHome =
   (CustomerHome)ic.lookup("java:comp/env/purchaseOrderBean");
```

Credentials in the InitialContext

To access remote EJBs from a servlet or JavaBean, pass the credentials in the `InitialContext` object, as follows:

```
Hashtable env = new Hashtable();
env.put("java.naming.provider.url", "ormi://myhost/ejbsamples");
env.put("java.naming.factory.initial",
   "oracle.j2ee.server.ApplicationClientInitialContextFactory");
env.put(Context.SECURITY_PRINCIPAL, "POMGR");
env.put(Context.SECURITY_CREDENTIALS, "welcome");
Context ic = new InitialContext (env);
CustomerHome =
   (CustomerHome)ic.lookup("java:comp/env/purchaseOrderBean")
```

13
Oracle HTTPS for Client Connections

This chapter describes the OC4J implementation of HTTPS that provides SSL functionality to client HTTP connections. It discusses how to use the Secure Sockets Layer protocol to communicate securely between networked applications, and the use of Oracle HTTPS and JSSE. The following topics are included:

- Oracle HTTPS and Clients
- Overview of Oracle HTTPS Features
- Specifying Default System Properties
- Oracle HTTPS Example
- Using HTTPClient with JSSE

This chapter assumes that you have already obtained keys and certificates.

See Also:

- Chapter 11, "Configuring OC4J and SSL" for a general overview of configuring OC4J to use the Secure Sockets Layer

---

Note: Secure communication between a client and Oracle HTTP Server is independent of secure communication between Oracle HTTP Server and OC4J. (Also note that the secure AJP protocol used between Oracle HTTP Server and OC4J is not visible to the end user.) This section covers only secure communication between OC4J and the client.

Oracle HTTPS and Clients

HTTPS is vital to securing client/server interactions. For many server applications, HTTPS is handled by the Web server. However, any application that acts as a client, such as servlets that initiate connections to other Web servers, needs its own HTTPS implementation to make requests and to receive information securely from the server. Java application developers who are familiar with either the HTTP package, `HTTPClient`, or who are familiar with the Sun Microsystems, Inc., `java.net` package can easily use Oracle HTTPS to secure client interactions with a server.

Oracle HTTPS extends the `HTTPConnection` class of the `HTTPClient` package, which provides a complete HTTP client library. To support client HTTPS connections, several methods have been added to the `HTTPConnection` class that use the OracleSSL class, `OracleSSLCredential`.

Note: Oracle `HTTPClient` supports two different SSL implementations: the Java Secure Socket Extension (JSSE) and OracleSSL. This documentation discusses the two implementations separately.

HTTPConnection Class

The `HTTPConnection` class is used to create new connections that use HTTP, with or without SSL. To provide support for PKI (public key infrastructure) digital certificates and wallets, the methods described in "Oracle HTTPS Example" on page 13-7 have been added to this class.

See Also:

- `HTTPClient` Javadoc: *Oracle Application Server HTTPClient API Reference*

OracleSSLCredential Class (OracleSSL Only)

Security credentials are used to authenticate the server and the client to each other. Oracle HTTPS uses the Oracle Java SSL package, `OracleSSLCredential`, to load user certificates and trustpoints from base64 or DER-encoded certificates. (DER, part of the X.690 ASN.1 standard, stands for Distinguished Encoding Rules.)

The API for Oracle Java SSL requires that security credentials be passed to the HTTP connection before the connection is established. The `OracleSSLCredential` class is used to store these security credentials. Typically, a wallet generated by Oracle Wallet Manager is used to populate the `OracleSSLCredential` object. Alternatively, individual certificates can be added by using an `OracleSSLCredential` class API. After the credentials are complete, they are passed to the connection with the `setCredentials()` method.

Overview of Oracle HTTPS Features

Oracle HTTPS supports HTTP 1.0 and HTTP 1.1 connections between a client and a server. To provide SSL functionality, new methods have been added to the `HTTPConnection` class of this package. These methods are used in conjunction with Oracle Java SSL to support cipher suite selection, security credential management with Oracle Wallet Manager, security-aware applications, and other features that are described in the following sections. Oracle HTTPS uses the Oracle Java SSL class, `OracleSSLCredential`, and it extends the `HTTPConnection` class of the `HTTPClient` package. `HTTPClient` supports two SSL implementations, OracleSSL and JSSE.

In addition to the functionality included in the `HTTPClient` package, Oracle HTTPS supports the following:

- Multiple cryptographic algorithms
- Certificate and key management with Oracle Wallet Manager
- Limited support for the `java.net.URL` framework
- Both the OracleSSL and JSSE SSL implementations

In addition, Oracle HTTPS uses the `HTTPClient` package to support:

- HTTP tunneling through proxies

- HTTP proxy authentication

The following sections describe Oracle HTTPS features in detail:

- SSL Cipher Suites
- SSL Cipher Suites Supported by OracleSSL
- SSL Cipher Suites Supported by JSSE
- Security-Aware Applications Support
- Framework Support for java.net.URL

SSL Cipher Suites

Before data can flow through an SSL connection, both sides of the connection must negotiate common algorithms to be used for data transmission. A set of such algorithms combined to provide a mix of security features is called a *cipher suite*. Selecting a particular cipher suite lets the participants in an SSL connection establish the appropriate level for their communications.

`HTTPClient` supports two different SSL implementations, each of which supports different cipher suites. These are discussed in this section.

Choosing a Cipher Suite

In general, you should prefer:

- RSA to Diffie-Hellman, because RSA defeats many security attacks
- 3DES or RC4 128 to other encryption methods, because 3DES and RC4 128 have strong keys
- SHA1 digest to MD5, because SHA1 produces a stronger digest

SSL Cipher Suites Supported by OracleSSL

OracleSSL supports the cipher suites listed in Table 13–1. Note that with `NULL` encryption, SSL is only used for authentication and data integrity purposes.

Table 13–1 Cipher Suites Supported by OracleSSL

Cipher Suite	Authentication	Encryption	Hash Function (Digest)
`SSL_RSA_WITH_3DES_EDE_CBC_SHA`	RSA	3DES EDE CBC	SHA1
`SSL_RSA_WITH_RC4_128_SHA`	RSA	RC4 128	SHA1
`SSL_RSA_WITH_RC4_128_MD5`	RSA	RC4 128	MD5
`SSL_RSA_WITH_DES_CBC_SHA`	RSA	DES CBC	SHA1
`SSL_RSA_EXPORT_WITH_RC4_40_MD5`	RSA	RC4 40	MD5
`SSL_RSA_EXPORT_WITH_DES40_CBC_SHA`	RSA	DES40 CBC	SHA1
`SSL_DH_anon_WITH_3DES_EDE_CBC_SHA`	DH anon	3DES EDE CBC	SHA1
`SSL_DH_anon_WITH_RC4_128_MD5`	DH anon	RC4 128	MD5
`SSL_DH_anon_WITH_DES_CBC_SHA`	DH anon	DES CBC	SHA1
`SSL_DH_anon_EXPORT_WITH_RC4_40_MD5`	DH anon	RC4 40	MD5

Table 13–1 (Cont.) Cipher Suites Supported by OracleSSL

Cipher Suite	Authentication	Encryption	Hash Function (Digest)
`SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA`	DH anon	DES40 CBC	SHA1
`SSL_RSA_WITH_NULL_SHA`	RSA	NULL	SHA1
`SSL_RSA_WITH_NULL_MD5`	RSA	NULL	MD5

SSL Cipher Suites Supported by JSSE

JSSE supports the cipher suites listed in Table 13–2. Note that with `NULL` encryption, SSL is only used for authentication and data integrity purposes.

Table 13–2 Cipher Suites Supported by JSSE

Cipher Suite	Authentication	Encryption	Hash Function (Digest)
`SSL_RSA_WITH_3DES_EDE_CBC_SHA`	RSA	3DES EDE CBC	SHA1
`SSL_RSA_WITH_RC4_128_SHA`	RSA	RC4 128	SHA1
`SSL_RSA_WITH_RC4_128_MD5`	RSA	RC4 128	MD5
`SSL_RSA_WITH_DES_CBC_SHA`	RSA	DES CBC	SHA1
`SSL_RSA_EXPORT_WITH_RC4_40_MD5`	RSA	RC4 40	MD5
`SSL_DH_anon_WITH_3DES_EDE_CBC_SHA`	DH anon	3DES EDE CBC	SHA1
`SSL_DH_anon_WITH_RC4_128_MD5`	DH anon	RC4 128	MD5
`SSL_DH_anon_WITH_DES_CBC_SHA`	DH anon	DES CBC	SHA1
`SSL_DH_anon_EXPORT_WITH_RC4_40_MD5`	DH anon	RC4 40	MD5
`SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA`	DH anon	DES40 CBC	SHA1
`SSL_RSA_WITH_NULL_SHA`	RSA	NULL	SHA1
`SSL_RSA_WITH_NULL_MD5`	RSA	NULL	MD5
`SSL_DHE_DSS_WITH_DES_CBC_SHA`	DH	DES CBC	SHA1
`SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA`	DH	3DES EDE CBC	SHA1
`SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA`	DH	DES40 CBC	SHA1

Access Information About Established SSL Connections

Users can access information about established SSL connections using the `getSSLSession()` method of Oracle HTTPS. After a connection is established, users can retrieve the cipher suite used for the connection, the peer certificate chain, and other information about the current connection.

Security-Aware Applications Support

Oracle HTTPS uses Oracle Java SSL to provide security-aware applications support. When security-aware applications do not set trust points, Oracle Java SSL allows them to perform their own validation letting the handshake complete successfully only if a complete certificate chain is sent by the peer. When applications authenticate to the trustpoint level, they are responsible for authenticating individual certificates below the trustpoint.

After the handshake is complete, the application must obtain the SSL session information and perform any additional validation for the connection.

Security-unaware applications that need the trust point check must ensure that trust points are set in the HTTPS infrastructure.

See Also:

- *Oracle Advanced Security Administrator's Guide* for information about Oracle Java SSL

Framework Support for java.net.URL

The `HTTPClient` package provides basic support for the `java.net.URL` framework with the `HTTPClient.HttpUrlConnection` class. However, many of the Oracle HTTPS features are supported through system properties only.

Features that are only supported through system properties are:

- Cipher suites selection option
- Confidentiality-only option
- Server authentication option
- Mutual authentication option
- Security credential management with Oracle Wallet Manager

Note: If the `java.net.URL` framework is used, then set the `java.protocol.handler.pkgs` system property to select the `HTTPClient` package as a replacement for the JDK client, as follows:

```
java.protocol.handler.pkgs=HTTPClient
```

See Also:

- "Specifying Default System Properties" below for information about configuring your client to use JSSE
- *Oracle Application Server Administrator's Guide* for information about wallets and Oracle Wallet Manager

Specifying Default System Properties

For many users of HTTPS it is desirable to specify some default properties in a non-programmatic way. The best way to accomplish this is through Java system properties which are accessible through the `java.lang.System` class. These properties are the only way for users of the `java.net.URL` framework to set security credential information. Oracle HTTPS recognizes the following properties:

- The javax.net.ssl.KeyStore Property
- The javax.net.ssl.KeyStorePassword Property
- The Oracle.ssl.defaultCipherSuites Property (OracleSSL Only)

The following sections describe how to set these properties.

The javax.net.ssl.KeyStore Property

This property can be set to point to the text wallet file exported from Oracle Wallet Manager that contains the credentials that are to be used for a specific connection. For example:

```
javax.net.ssl.KeyStore=/etc/ORACLE/WALLETS/Default/default.txt
```

In this syntax, *`default.txt`* is the name of the text wallet file that contains the credentials.

If no other credentials have been set for the HTTPS connection, then the file indicated by this property is opened when a handshake first occurs. If any errors occur while reading this file, then the connection fails and an `IOException` is thrown.

If you do not set this property, the application is responsible for verifying that the certificate chain contains a certificate that can be trusted. However, `HTTPClient` using Oracle SSL does verify that all of the certificates in the certificate chain, from the user certificate to the root CA, have been sent by the server and that all of these certificates contain valid signatures.

The javax.net.ssl.KeyStorePassword Property

This property can be set to the password that is necessary to open the wallet file. For example:

```
javax.net.ssl.KeyStorePassword=welcome1
```

In this syntax, *`welcome1`* is the password that is necessary to open the wallet file.

Important: Storing the wallet file password as a Java system property can result in a security risk in some environments. To avoid this risk, use one of the following alternatives:

- If mutual authentication is not required for the application, use a text wallet that contains no private key. No password is needed to open these wallets.
- If a password is necessary, then do not store it in a cleartext file. Instead, load the property dynamically before the `HTTPConnection` is started by using `System.setProperty()`. Unset the property after the handshake is completed.

The Oracle.ssl.defaultCipherSuites Property (OracleSSL Only)

This property can be set to a comma-delimited list of cipher suites. For example:

```
Oracle.ssl.defaultCipherSuites=
          SSL_RSA_WITH_DES_CBC_SHA,\
          SSL_RSA_EXPORT_WITH_RC4_40_MD5,\
          SSL_RSA_WITH_RC4_128_MD5
```

The cipher suites that you set this property to are used as the default cipher suites for new HTTPS connections.

See Also:

- Table 13–1 on page 13-3 for a complete list of the cipher suites that are supported by OracleSSL

Oracle HTTPS Example

The following is a simple program that uses Oracle HTTPS, `HTTPClient`, and OracleSSL to connect to a Web server, send a `GET` request, and fetch a Web page. The complete code for this program is presented here followed by sections that explain how Oracle HTTPS is used to set up secure connections.

```
import HTTPClient.HTTPConnection;
import HTTPClient.HTTPResponse;
import oracle.security.ssl.OracleSSLCredential;
import java.io.IOException;

public class HTTPSConnectionExample
{
    public static void main(String[] args)
    {
        if(args.length < 4)
        {
            System.out.println(
            "Usage: java HTTPSConnectionTest [host] [port] " +
            "[wallet] [password]");
            System.exit(-1);
        }

        String hostname = args[0].toLowerCase();
        int port = Integer.decode(args[1]).intValue();
        String walletPath = args[2];
        String password = args[3];

        HTTPConnection httpsConnection = null;
        OracleSSLCredential credential = null;

        try
        {
            httpsConnection = new HTTPConnection("https", hostname, port);
        }
        catch(IOException e)
        {
            System.out.println("HTTPS Protocol not supported");
            System.exit(-1);
        }

        try
        {
            credential = new OracleSSLCredential();
            credential.setWallet(walletPath, password);
        }
        catch(IOException e)
        {
            System.out.println("Could not open wallet");
            System.exit(-1);
        }
        httpsConnection.setSSLCredential(credential);

        try
        {
            httpsConnection.connect();
        }
        catch (IOException e)
        {
            System.out.println("Could not establish connection");
            e.printStackTrace();
            System.exit(-1);
        }

        javax.servlet.request.X509Certificate[] peerCerts = null;
        try
        {
            peerCerts =
              (httpsConnection.getSSLSession()).getPeerCertificateChain();
        }
        catch(javax.net.ssl.SSLPeerUnverifiedException e)
        {
            System.err.println("Unable to obtain peer credentials");
            System.exit(-1);
        }

        String peerCertDN =
            peerCerts[peerCerts.length -1].getSubjectDN().getName();
        peerCertDN = peerCertDN.toLowerCase();
        if(peerCertDN.lastIndexOf("cn="+hostname) == -1)
        {
            System.out.println("Certificate for " + hostname + " is issued to "
              + peerCertDN);
            System.out.println("Aborting connection");
            System.exit(-1);
        }

        try
        {
            HTTPResponse rsp = httpsConnection.Get("/");
            System.out.println("Server Response: ");
            System.out.println(rsp);
        }
        catch(Exception e)
        {
            System.out.println("Exception occured during Get");
            e.printStackTrace();
            System.exit(-1);
        }
    }
}
```

Initializing SSL Credentials in OracleSSL

This program example uses a wallet created by Oracle Wallet Manager to set up credential information. First the credentials are created and the wallet is loaded as follows:

```
credential = new OracleSSLCredential();
credential.setWallet(walletPath, password);
```

After the credentials are created, they are passed to `HTTPSConnection`:

```
httpsConnection.setSSLCredential(credential);
```

The private key, user certificate, and trust points located in the wallet can now be used for the connection.

Verifying Connection Information

Although SSL verifies that the certificate chain presented by the server is valid and contains at least one certificate trusted by the client, that does not prevent impersonation by malicious third parties. An HTTPS standard that addresses this problem requires that HTTPS servers have certificates issued to their host name. Then it is the responsibility of the client to perform this validation after the SSL connection is established.

To perform this validation in this sample program, `HTTPSConnectionExample` establishes a connection to the server without transferring any data using the following:

```
httpsConnection.connect();
```

After the connection is established, the connection information, in this case the server certificate chain, is obtained with the following:

```
peerCerts = (httpsConnection.getSSLSession()).getPeerCertificateChain();
```

Finally the server certificate common name is obtained with the following:

```
String peerCertDN = peerCerts[peerCerts.length -1].getSubjectDN().getName();
peerCertDN = peerCertDN.toLowerCase();
```

If the certificate name is not the same as the host name used to connect to the server, then the connection is aborted with the following:

```
if(peerCertDN.lastIndexOf("cn="+hostname) == -1)
{
    System.out.println("Certificate for " + hostname + " is issued to " +
        peerCertDN);
    System.out.println("Aborting connection");
        System.exit(-1);
}
```

Transferring Data Using HTTPS

It is important to verify the connection information before data is transferred from the client or from the server. The data transfer is performed in the same way for HTTPS as it is for HTTP. In this sample program a `GET` request is made to the server using the following:

```
HTTPResponse rsp = httpsConnection.Get("/");
```

Using HTTPClient with JSSE

Oracle Application Server supports HTTPS client connections using the Java Secure Socket Extension (JSSE). A client can configure `HTTPClient` to use JSSE as the underlying SSL provider.

Note: The JSSE SSL implementation is not thread-safe; if you need to use SSL in a threaded application, use OracleSSL.

See Also:

- For full information on JSSE, Sun documentation at:

  `http://java.sun.com/products/jsse/`

`HTTPClient` still uses OracleSSL as the default provider, but the developer can easily change this by setting the `SSLSocketFactory` on the `HTTPConnection` class. Example 13–1 demonstrates how a client could configure HTTPClient to use JSSE for SSL communication.

Example 13–1 Using JSSE with HTTPClient

```
public void obtainHTTPSConnectionUsingJSSE() throws Exception
{
// set the trust store to the location of the client's trust store file
      // this value specifies the certificate authorities the client accepts
   System.setProperty("javax.net.ssl.trustStore", KEYSTORE_FILE);
   // creates the HTTPS URL
   URL testURL = new URL("https://" + HOSTNAME + ":" + HTTPS_PORTNUM);
   // call SSLSocketFactory.getDefault() to obtain the default JSSE implementation
  // of an SSLSocketFactory
   SSLSocketFactory socketFactory =
           (SSLSocketFactory)SSLSocketFactory.getDefault();
   HTTPConnection connection = new HTTPConnection(testURL);

   // configure HTTPClient to use JSSE as the underlying
   // SSL provider
   connection.setSSLSocketFactory(socketFactory);
   // call connect to setup SSL handshake
   try
   {
       connection.connect();
   }
   catch (IOException e)
   {
       e.printStackTrace();    }

   HTTPResponse response = connection.Get("/index.html");

}
```

Configuring HTTPClient to Use JSSE

The steps required to use JSSE with `HTTPClient` are as follows:

1. Create a truststore using the keytool.
2. Set the truststore property. A client wishing to use JSSE must specify the client truststore location in `javax.net.ssl.trustStore`. Unlike OracleSSL, the client does not need to set the `javax.net.ssl.keyStore` property.
3. Obtain the JSSE `SSLSocketFactory` by calling `SSLSocketFactory.getDefault()`.
4. Create an `HTTPClient` connection.
5. Configure the `HTTPClient` connection to use the JSSE implementation of SSL. `HTTPClient` can be configured to use JSSE in one of two ways:
   a. (**For each connection**) The client calls the `setSSLSocketFactory(SSLSocketFactory factory)` method on the `HTTPConnection` instance.
   b. (**Entire VM**) The client calls the static method `setDefaultSSLSocketFactory(SSLSocketFactory factory)` on the

`HTTPConnection` class. This static method must be called before instantiating any `HTTPConnection` instances.

6. Call `HTTPConnection.connect()` before sending any HTTPS data. This allows the connection to verify the SSL handshaking that must occur between client and server before any data can be encrypted and sent.
7. Use the `HTTPConnection` instance normally. At this point, the client is set up to use `HTTPClient` with JSSE. There is no additional configuration necessary and basic usage is the same.

Note: The JSSE implementation of SSL has some subtle differences from the Oracle implementation. Unlike in OracleSSL, if no truststore is set, the JDK default truststore will be used. This default will accept known certificate authorities, such as Verisign and Thawte. Many self-signed certificates will be rejected by this default.

See Also:

- For details of using the keytool:

  `http://java.sun.com/j2se/1.3/docs/tooldocs/win32/keytool.html`

14 Password Management

This chapter discusses managing passwords within XML files. It contains the following sections:

Introduction

Many OC4J components require passwords for authentication. Embedding these passwords into deployment and configuration files poses a security risk, especially if the permissions on the files allow them to be read by any user. To avoid this problem, OC4J provides two solutions:

- *Password obfuscation*: This replaces passwords stored in cleartext files with an encrypted version of the password, and is discussed in "Password Obfuscation in jazn-data.xml and jazn.xml" below.
- *Password indirection*: This replaces cleartext passwords with information necessary to look up the password in another location, and is discussed in "Creating an Indirect Password" on page 14-2.

Password Obfuscation in jazn-data.xml and jazn.xml

The JAAS configuration files, `jazn.xml` and `jazn-data.xml`, contain user names and passwords for JAAS authorization. To protect these files, OC4J uses password obfuscation.

Whenever you update `jazn.xml` or `jazn-data.xml`, OC4J reads the file, then rewrites it with obfuscated (encrypted) versions of all passwords. In all other OC4J configuration files, you can avoid exposing password cleartext by using password indirection, as explained in "Creating an Indirect Password" on page 14-2.

The OracleAS JAAS Provider does not obfuscate passwords in `orion-application.xml`. This means that you should not embed passwords within a `<jazn>` element that is stored in `orion-application.xml`.

Note: For security reasons, credentials stored in Oracle Internet Directory cannot usually be retrieved in decrypted (cleartext) format, which means the LDAP-based JAAS provider cannot be used as a password manager for your application. To resolve this, you can specify the XML-based JAAS provider as your application password manager even when your application uses the LDAP-based JAAS provider as the user manager.

To do this, add the following entry to `application.xml`:

```
<password-manager>
  <jazn provider="XML"
   location="ORACLE_HOME/j2ee/instance_name/config/jazn-data.xml"/>
</password-manager>
```

Otherwise, passwords are not obfuscated.

Editing jazn-data.xml

If you prefer, you can directly edit `jazn-data.xml` with a text editor. The next time OC4J reads `jazn-data.xml`, it will rewrite the file with all passwords obfuscated and unreadable.

Setting the `clear` attribute of the `<credentials>` element to `"true"` enables you to use clear (human-readable) passwords in the `jazn-data.xml` file:

```
<credentials clear="true">welcome</credentials>
<credentials>!welcome</credentials>
```

Creating an Indirect Password

The following OC4J XML configuration and deployment files support password indirection in one or more entities:

- `data-sources.xml`: `password` attribute of `<data-source>` element
- `ra.xml`: `<res-password>` element
- `rmi.xml`: `password` attribute of `<cluster>` element
- `application.xml`: `password` attribute of `<resource-provider>` and `<commit-coordinator>` elements
- `jms.xml`: `<password>` element
- `internal-settings.xml`: `<sep-property>` element, with `name="keystore-password"` and `name="truststore-password"`

To make any of these passwords indirect, replace the literal password string with a string containing "`->`" followed by either the user name or by the realm and user name separated by a slash ("`/`").

Note: To begin a literal (non-indirect) password with the string "`->`", precede the password by "`->!`". For example, you would represent the direct password "`->silly`" as "`->!->silly`".

Indirect Password Examples

- `<data-source password="->Scott">`: Use `JAZNUserManager` to look up `Scott` in the `JAZNUserManager`, and use the password stored there.
- `<res-password="->customers/Scott">`: Use `JAZNUserManager` to look up `Scott` in the `customers` realm, and use the password stored there.
- `<cluster password="martha">`: The literal string "`martha`" is the password; the password is not indirect.

Specifying a User Manager in application.xml

The `<password-manager>` element in `application.xml` specifies the user manager that the global application uses to look up indirect passwords. If this element is omitted, the user manager of the global application is used for authentication and authorization of indirect passwords. The `<jazn>` element within a `<password-manager>` element can be different from the `<jazn>` element at the top level.

The `<password-manager>` element should always contain the path name of the instance-level `jazn-data.xml` file.

For example, you can use an LDAP-based user manager for the regular user manager, but use an XML-based user manager to authenticate indirect passwords. This is the only way to use indirect passwords in LDAP.

Note: It is possible to use pluggable user managers as password managers. However, if you use `XMLUserManager` as your password manager, `principals.xml` will not have passwords obfuscated.

See Also:

- "Creating an Indirect Password" on page 14-2
- "Specifying User Managers" on page 4-5

15 Configuring CSIv2

OC4J supports the Common Secure Interoperability Version 2 protocol (CSIv2). CSIv2 specifies different conformance levels; OC4J complies with the EJB specification, which requires conformance level 0.

This chapter covers the following topics:

Note: If your application uses JAAS, you must configure the OracleAS JAAS Provider to use CSIv2, as discussed in Table 4–2, " RealmLoginModule Options" on page 4-8.

Introduction to CSIv2 Security Properties

CSIv2 is an Object Management Group (OMG) standard for a secure interoperable wire protocol that supports authorization and identity delegation. You configure CSIv2 properties in three different locations:

- `internal-settings.xml` (discussed in "CSIv2 Security Properties in internal-settings.xml" on page 15-3)
- `orion-ejb-jar.xml` (discussed in "CSIv2 Security Properties in orion-ejb-jar.xml" on page 15-5)
- `ejb_sec.properties` (discussed in "EJB Client Security Properties in ejb_sec.properties" on page 15-6)

EJB Server Security Properties in internal-settings.xml

You specify server security properties in `internal-settings.xml`.

Note: You cannot edit `internal-settings.xml` with Enterprise Manager.

This file specifies certain properties as values within `<sep-property>` entities. Table 15–1 contains a list of properties.

The table refers to keystore and truststore files, which use the Java Key Store (JKS), a JDK-specified format, to store keys and certificates. A keystore stores a map of private keys and certificates. A truststore stores trusted certificates for the certificate authorities (CAs, such as VeriSign and Thawte).

Table 15–1 EJB Server Security Properties

Property	Meaning
`port`	IIOP port number (defaults to `5555`).
`ssl`	This is "`true`" if IIOP/SSL is supported, "`false`" otherwise.
`ssl-port`	IIOP/SSL port number (defaults to `5556`). This port is used for server-side authentication only. If your application uses client and server authentication, you must also set `ssl-client-server-auth-port`.
`ssl-client-server-auth-port`	Port used for client and server authentication (defaults to `5557`). This is the port on which OC4J listens for SSL connections that require both client and server authentication. If this is not set, OC4J will listen on `ssl-port` + 1 for client-side authentication.
`keystore`	Name of keystore (used only if `ssl` is "`true`").
`keystore-password`	Keystore password (used only if `ssl` is "`true`").
`trusted-clients`	Comma-delimited list of hosts whose identity assertions can be trusted. Each entry in the list can be an IP address, a host name, or a host name pattern (such as "`*.example.com`" or "`*`", where "`*`" alone means that all clients are trusted). The default is to trust no clients.
`truststore`	Name of truststore. If you do not specify a truststore for a server, OC4J uses the keystore as the truststore (used only if `ssl` is "`true`").
`truststore-password`	Truststore password (can be set only if `ssl` is "`true`").

Be aware of the following:

- If OC4J is started by the Oracle Process Management Notification service (OPMN) in an Oracle Application Server (as opposed to standalone) environment, then ports specified in `internal-settings.xml` are overridden. (Note that the IIOP SSL ports may fail to start if the keystore or truststore location or password is missing or incorrect. In such a case, you are advised to look at the log *`ORACLE_HOME`*`/opmn/logs/OC4J-home-default_island-1` to see the exact nature of the failure.)
- If OPMN is configured to disable IIOP for a particular OC4J instance, then even though IIOP may be enabled through `internal-settings.xml` (as pointed to by `server.xml`), IIOP is not enabled.
- Keystore and truststore settings are supported in `internal-settings.xml` for both standalone OC4J and a full Oracle Application Server environment. In Oracle Application Server, there are no OPMN options to set these values, so they must be configured manually.

- When running in Oracle Application Server with OPMN, absolute path names are required for the keystore and truststore.

The following example shows an `internal-settings.xml` file:

```
<server-extension-provider name="IIOP"
        class="com.oracle.iiop.server.IIOPServerExtensionProvider">
    <sep-property name="port" value="5555" />
    <sep-property name="host" value="localhost" />
    <sep-property name="ssl" value="false" />
    <sep-property name="ssl-port" value="5556" />
    <sep-property name="ssl-client-server-auth-port" value="5557" />
    <sep-property name="keystore" value="keystore.jks" />
    <sep-property name="keystore-password" value="123456" />
    <sep-property name="truststore" value="truststore.jks" />
    <sep-property name="truststore-password" value="123456" />
    <sep-property name="trusted-clients" value="*" />
</server-extension-provider>
```

Note: Although the default value of `port` is one less than the default value for `ssl-port`, this relationship is not required.

Here is the DTD for `internal-settings.xml`:

```
<!-- A server extension provider that is to be plugged in to the server.
-->
<!ELEMENT server-extension-provider (sep-property*) (#PCDATA)>
<!ATTLIST server-extension-provider name class CDATA #IMPLIED>
<!ELEMENT sep-property (#PCDATA)>
<!ATTLIST sep-property name value CDATA #IMPLIED>
<!-- This file contains internal server configuration settings. -->
<!ELEMENT internal-settings (server-extension-provider*)>
```

CSIv2 Security Properties in internal-settings.xml

This section discusses the semantics of the values you set within the `<sep-property>` element in `internal-settings.xml`. Syntax is discussed in the previous section, "EJB Server Security Properties in internal-settings.xml".

To use the CSIv2 protocol with OC4J, you must both set `ssl` to "`true`" and specify an IIOP/SSL port (`ssl-port`).

- If you do not set `ssl` to "`true`", then CSIv2 is not enabled. Setting `ssl` to "`true`" permits clients and servers to use CSIv2, but does not require them to communicate using SSL.
- If you do not specify an `ssl-port`, then no CSIv2 component element is inserted by the server into the IOR, even if you configure an `<ior-security-config>` element in `orion-ejb-jar.xml`.

When IIOP/SSL is enabled on the server, OC4J listens on two different sockets—one for server authentication alone and one for server and client authentication. Specify the server authentication port within the `<sep-property>` element; the server and client authentication listener uses the port number immediately following.

For SSL clients using server authentication alone, you can specify any of the following:

- Truststore only
- Both keystore and truststore

- Neither

If you specify neither keystore nor truststore, the handshake may fail if there are no default truststores established by the security provider.

SSL clients using client-side authentication must specify both a keystore and a truststore. The certificate from the keystore is used for client authentication.

CSIv2 Security Properties in ejb_sec.properties

If the client does not use client-side SSL authentication, you must set `client.sendpassword` in the `ejb_sec.properties` file in order for the client runtime to insert a security context and send the user name and password. You must also set `server.trustedhosts` to include your server.

Note: Server-side authentication takes precedence over a user name and password.

If the client does use client-side SSL authentication, the server extracts the `DistinguishedName` from the client certificate and then looks it up in the corresponding user manager; it does not perform password authentication.

Trust Relationships

Two types of trust relationships exist:

- Clients trusting servers to transmit user names and passwords using non-SSL connections
- Servers trusting clients to send *identity assertions*, which delegate an originating client identity

Clients list trusted servers in the EJB property `oc4j.iiop.trustedServers`. Servers list trusted clients in the `trusted-client` property of the `<sep-property>` element in `internal-settings.xml`.

See Also:

- "EJB Client Security Properties in ejb_sec.properties" on page 15-6
- "EJB Server Security Properties in internal-settings.xml" on page 15-1

Conformance level 0 of the EJB standard defines two ways of handling trust relationships:

- *Presumed trust*, in which the server presumes that the logical client is trustworthy, even if the logical client has not authenticated itself to the server, and even if the connection is not secure
- *Authenticated trust*, in which the target trusts the intermediate server based on authentication either at the transport level or in the `trusted-client` list or both

Note: You can also configure the server to both require SSL client-side authentication and also specify a list of trusted client (or intermediate) hosts that are allowed to insert identity assertions.

OC4J provides both kinds of trust. Configure trust using the `<ior-security-config>` element for the bean in `orion-ejb-jar.xml`, as discussed in "CSIv2 Security Properties in orion-ejb-jar.xml" immediately following.

CSIv2 Security Properties in orion-ejb-jar.xml

This section discusses the CSIv2 security properties for an EJB. You configure CSIv2 security policies for each individual bean in its `orion-ejb-jar.xml`. The CSIv2 security properties are specified within `<ior-security-config>` elements. Each element contains a `<transport-config>` element, an `<as-context>` element, and a `<sas-context>` element.

DTD for <ior-security-config>

Here is the DTD for the `<ior-security-config>` element:

```
<!ELEMENT ior-security-config (transport-config?, as-context?
sas-context?) >
<!ELEMENT transport-config (integrity, confidentiality,
establish-trust-in-target, establish-trust-in-client) >
<!ELEMENT as-context (auth-method, realm, required) >
<!ELEMENT sas-context (caller-propagation) >
<!ELEMENT integrity (#PCDATA) >
<!ELEMENT confidentiality (#PCDATA)>
<!ELEMENT establish-trust-in-target (#PCDATA) >
<!ELEMENT establish-trust-in-client (#PCDATA) >
<!ELEMENT auth-method (#PCDATA) >
<!ELEMENT realm (#PCDATA) >
<!ELEMENT required (#PCDATA)> <!-- Must be true or false -->
<!ELEMENT caller-propagation (#PCDATA) >
```

<transport-config>

This element specifies the transport security level. Each element within `<transport-config>` must be set to `supported`, `required`, or `none`—`none` means that the bean neither supports nor uses that feature; `supports` means that the bean permits the client to use the feature; `required` means that the bean insists that the client use the feature. The elements are:

- `<integrity>`: Is there a guarantee that all transmissions are received exactly as they were transmitted?
- `<confidentiality>`: Is there a guarantee that no third party was able to read transmissions?
- `<establish-trust-in-target>`: Does the server authenticate itself to the client? This element may be set to either `supported` or `none`; it cannot be set to `required`.
- `<establish-trust-in-client>`: Does the client authenticate itself to the server? This element must be set to `supported`, `required`, or `none`.

Notes: If you set `<establish-trust-in-client>` to `required`, this overrides specifying `username_password` in `<as-context>`. If you do this, you must also set the `<required>` node value in the `<as-context>` section to `false`; otherwise access permission issues will arise.

Setting any of the `<transport-config>` properties to `required` means that the bean will use RMI/IIOP/SSL to communicate.

<as-context>

This element specifies the message-level authentication properties.

- `<auth-method>`: Must be set to either `username_password` or `none`. If set to `username_password`, beans use user names and passwords to authenticate the caller.
- `<realm>`: Must be set to `default` in the OC4J 10.1.2 implementation.
- `<required>`: If set to `true`, the bean requires the caller to specify a user name and password.

<sas-context>

This element specifies the identity delegation properties. It has one element, `<caller-propagation>`, which can be set to `supported`, `required`, or `none`. If the `<caller-propagation>` element is set to `supported`, then this bean accepts delegated identities from intermediate servers. If it is set to `required`, then this bean requires all other beans to transmit delegated identities. If set to `none`, this bean does not support identity delegation.

An example:

```
<ior-security-config>
  <transport-config>
    <integrity>supported</integrity>
    <confidentiality>supported</confidentiality>
    <establish-trust-in-target>supported</establish-trust-in-target>
    <establish-trust-in-client>supported</establish-trust-in-client>
  </transport-config>
  <as-context>
    <auth-method>username_password</auth-method>
    <realm>default</realm>
    <required>true</required>
  </as-context>
  <sas-context>
    <caller-propagation>supported</caller-propagation>
  </sas-context>
</ior-security-config>
```

EJB Client Security Properties in ejb_sec.properties

Any client, whether running inside a server or not, has EJB security properties. Table 15–2 lists the EJB client security properties controlled by the `ejb_sec.properties` file. By default, OC4J searches for this file in the current directory when running as a client or in `*ORACLE_HOME*/j2ee/*instance_name*/config` when running in the server. You can specify the file location explicitly as follows:

```
-Dejb_sec_properties_location=pathname
```

Table 15–2 EJB Client Security Properties

Property	Meaning
`# oc4j.iiop.keyStoreLoc`	The path name for the keystore.
`# oc4j.iiop.keyStorePass`	The password for the keystore.
`# oc4j.iiop.trustStoreLoc`	The path name for the truststore.
`# oc4j.iiop.trustStorePass`	The password for the truststore.
`# oc4j.iiop.enable.clientauth`	Whether the client supports client-side authentication. If this property is set to `true`, you must specify a keystore location and password.

`oc4j.iiop.ciphersuites`	Which cipher suites are to be enabled. The valid cipher suites are:
	`TLS_RSA_WITH_RC4_128_MD5`
	`SSL_RSA_WITH_RC4_128_MD5`
	`TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA`
	`SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA`
	`TLS_RSA_EXPORT_WITH_RC4_40_MD5`
	`SSL_RSA_EXPORT_WITH_RC4_40_MD5`
	`TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA`
	`SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA`

`nameservice.useSSL`	Whether to use SSL when making the initial connection to the server.
`client.sendpassword`	Whether to send user name and password in clear form (unencrypted) in the service context when not using SSL. If this property is set to `true`, the user name and password are sent only to servers listed in the `trustedServer` list.
`oc4j.iiop.trustedServers`	A list of servers that can be trusted to receive passwords sent in clear form. This has no effect if `client.sendpassword` is set to `false`. The list is comma-delimited. Each entry in the list can be an IP address, a host name, or a host name pattern (such as "`*.example.com`" or "`*`", where "`*`" alone means that all servers are trusted).

Notes:

- The `ejb_sec.properties` file is used for client-side operations, where `internal-settings.xml` would not be present.
- The properties marked with a "#" can be set either in `ejb_sec.properties` or as system properties. The settings in `ejb_sec.properties` always override settings specified as system properties.

16 Troubleshooting Security Issues

This chapter discusses techniques for locating security problems in your OC4J application. It is divided into the following sections:

Alternative jazn.xml Locations

When the OracleAS JAAS Provider starts, it searches for a `jazn.xml` file. The `jazn.xml` file can be in a variety of locations, but is normally in the *`ORACLE_HOME`*`/j2ee/home/config` directory. However, if you specify the location of this file in a system property, the file in the system property takes precedence.

When the OracleAS JAAS Provider starts, it searches for `jazn.xml` in order through the directories specified by:

1. `oracle.security.jazn.config` (system property)
2. `java.security.auth.policy` (system property)
3. *`J2EE_HOME`*`/config` (*`J2EE_HOME`* is specified by the system property `oracle.j2ee.home`)
4. *`ORACLE_HOME`*`/j2ee/home/config` (*`ORACLE_HOME`* is specified by the system property `oracle.home`)
5. `./config`

The OracleAS JAAS Provider stops searching after locating a `jazn.xml` file. If no file is found, you receive the error message "`JAZN has not been properly configured`".

JAZN Admintool

Before using the Admintool, you must set the environment variable controlling loading of dynamic libraries (for example, `LD_LIBRARY_PATH` in Solaris). See Table 2–5, " Dynamic Library Path Settings" on page 2-9 for details.

Caution: The Admintool does not require authentication when used with the LDAP-based provider; anyone who runs the tool is granted all rights. This means that it is vital to secure the Admintool in production environments; you normally do this by using file system properties. If you specify the `-user` and `-password` options when using LDAP, they are ignored.

If you are attempting to grant a permission and the Admintool gives the error message "`Permission class not found`", it means that the permission you wish to grant is not in the classpath. You must place the JAR containing the permission class in the `jdk/jre/lib/ext` directory so that the Admintool can locate it.

Custom Login Modules

When writing a custom `LoginModule`, you should be aware of the following issues:

- Subject-Based Authorization
- J2EE Security Integration

Subject-Based Authorization

When an application uses a custom login module, the subject (and the principals it contains) are used as the sole basis for authorization, including the evaluation of J2EE security constraints. To ensure that all relevant principals are taken into consideration during authorization, the login module should add the relevant principals (including any roles/groups that the authenticated user belongs to) to the subject during the `commit` phase of the JAAS authentication process.

J2EE Security Integration

The custom `LoginModule` framework supports the J2EE security declarative security model. That is, the J2EE security constraints declared in application deployment descriptors such as `web.xml` and `ejb-jar.xml` are enforced using subject-based authorization.

We encourage J2EE developers to take advantage of the J2EE security model whenever possible, rather than writing their own security implementation; this ensures forward compatibility with future releases.

LDAP-Based Provider Issues

The following issues are important when troubleshooting the LDAP-based provider:

- Checking JAZN-LDAP Configuration
- Enabling and Disabling Caching

Checking JAZN-LDAP Configuration

When you associate an Oracle Application Server instance with Oracle Application Server Infrastructure, either during installation or using Enterprise Manager, the instance is automatically configured to use the LDAP-based provider. The Oracle Internet Directory location and port are determined by the `ORACLE_HOME/config/ias.properties` file.

To verify that the LDAP-based provider has been configured property, do the following:

1. Use Enterprise Manager to verify that the user manager is set to "LDAP".
2. Issue the JAZN Admintool `-listrealms` command to verify that the LDAP-based provider can retrieve data from Oracle Internet Directory:

   ```
   java -jar jazn.jar -listrealms
   ```

3. If the Admintool responds with the message "`Communication Error`", then it is likely that Oracle Internet Directory is down.
4. If the Admintool responds with the message "`Invalid Credentials`", then the LDAP users and credentials are incorrectly configured.

Enabling and Disabling Caching

LDAP caching is enabled by default; caching is per-JVM, not per-application. Before using JAAS Admintool management commands, such as granting permissions or roles, you must disable caching. After you use the Admintool, you should re-enable caching.

See Also:

- "Configuring LDAP Caching" on page 5-3 for details on enabling and disabling caching

Servlets, runas-mode, and doasprivileged-mode

If you want a servlet to be invoked using `subject.doAs()` or `subject.doAsPrivileged()`, you must set the `runas-mode` and `doasprivileged-mode` attributes of the `<jazn-web-app>` element in the `orion-web.xml` or `orion-application.xml` file.

See Also:

- "Configuring J2EE Authorization" on page 4-11

Creating Realms

It is important to use the appropriate tool to create realms. In general, if you use the LDAP-based provider or Oracle Application Server Single Sign-On, use Oracle Delegated Administration Services to create realms. If you use the XML-based provider, create realms with the JAAS Admintool. The realms you create with the JAAS Admintool are external or application realms; they are located in a different place in the realm tree than identity management realms.

Removing Realm Names from Principals

In some applications, you prefer to avoid parsing the principal returned by various method calls. You can configure the OracleAS JAAS Provider so that the returned principal contains no realm name. To do this, you add the property `jaas.username.simple` to the `<jazn>` element in the instance-level *`ORACLE_HOME`*`/j2ee/`*`instance_name`*`/config/jazn.xml` file, such as in the following example:

```
<property name="jaas.username.simple" value="true" />
```

This property affects the return values of the following methods:

- In `javax.servlet.http.HTTPServletRequest`, the `getRemoteUser()` and `getUserPrincipal()` methods
- In `javax.ejb.EJBContext`, the `getCallerIdentity()` and `getCallerPrincipal()` methods

Specifying the JAAS Provider

Consider an exception and stack trace similar to the following:

```
Exception in thread "main" java.lang.SecurityException: Unable to locate a login
configuration
at com.sun.security.auth.login.ConfigFile.<init>(ConfigFile.java:97)
at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
at sun.reflect.NativeConstructorAccessorImpl.newInstance
```

In this circumstance, you have probably failed to specify the OracleAS JAAS Provider as the JAAS policy provider, as discussed in "Specifying an Alternate Policy Provider (Optional)" on page 4-4.

17 Security Tips

This chapter provides tips in the following areas:

- HTTPS Tips
- Overall Security Tips
- JAAS Tips

The tips come from the Security Best Practices document, available from Oracle Technology Network at the following location:

`http://www.oracle.com/technology/index.html`

Check the OTN Web site for updates.

HTTPS Tips

Oracle HTTP Server (OHS) has several features that provide security to an application without requiring you to modify the application. You should evaluate and leverage these features before coding similar features yourself. HTTP security features include:

- Authentication: OHS can authenticate users and pass the authenticated user ID to an application in a standard manner (`REMOTE_USER`). It also supports single sign-on, thus reusing existing login mechanisms.
- Authorization: OHS has directives that can allow access to your application only if the end user is authenticated and authorized. Again, no code change is required.
- Encryption: OHS can provide transparent SSL communication to end customers without any code change on the application.

Other suggestions for securing HTTPS:

- *Configure Oracle Application Server to fail attempts to use weak encryption.* You can configure Oracle Application Server to use only specified encryption ciphers for HTTPS connections. For example, your application could reject connections from non-128-bit client-side SSL libraries. This ability is especially useful for banks and other financial institutions because it provides server-side control of the encryption strength for each connection.
- *Use HTTPS to HTTP appliances for accelerating HTTP over SSL.* Use HTTPS everywhere you need to. However, the performance overhead of HTTPS forces a trade-off in some situations.

  For a relatively low cost, HTTPS-to-HTTP appliances can change throughput on a 500MHz UNIX machine from 20-30 transactions per second to 6000 transactions per second, making this trade-off decision easier.

Moreover, these appliances provide much better solutions than adding mathematics or cryptography cards to UNIX, Windows, or Linux systems.

- *Ensure that sequential HTTPS transfers are requested through the same Web server.* Expect 40 to 50 milliseconds of CPU time to initiate SSL sessions on a 500 MHz machine. Most of this CPU time is spent in the key exchange logic, where the bulk encryption key is exchanged. Caching the bulk encryption will significantly reduces CPU overhead on subsequent accesses, provided that the accesses are routed to the same Web server.
- *Keep secure pages and pages not requiring security on separate servers.* Although it may be easier to place all pages for an application on one HTTPS server, this strategy has enormous performance costs. Reserve your HTTPS server for pages needing SSL, and put the pages not needing SSL on an HTTP server.

  If secure pages are composed of many GIF, JPEG, or other files to be displayed on the same screen, it is probably not worth the effort to segregate secure from nonsecure static content. The SSL key exchange (a major consumer of CPU cycles) is likely to be called exactly once in any case, and the overhead of bulk encryption is not that high.

Overall Security Tips

Suggestions for overall security:

- *When assigning privileges to modules, use the lowest levels that are adequate to perform the modules function(s).* Using low-level privileges provides "fault containment"—if security is compromised, it is contained within a small area of the network and cannot invade the entire intranet.
- *Tune the SSLSessionCacheTimeout directive if you are using SSL.* The Apache server in Oracle Application Server caches the SSL session information of a client by default. With session caching, only the first connection to the server incurs high latency.

  In a simple test to connect and disconnect to an SSL-enabled server, the elapsed time for 5 connections was 11.4 seconds without SSL session caching; with session caching enabled, the elapsed time was 1.9 seconds.

  The default `SSLSessionCacheTimeout` is 300 seconds. Note that the duration of an SSL session is unrelated to the use of HTTP persistent connections. You can change the `SSLSessionCacheTimeout` directive in the `httpd.conf` file to meet your application needs.

JAAS Tips

Suggestions for JAAS:

- *Migrate your user management from principals.xml to the OracleAS JAAS Provider.* In earlier releases of Oracle Application Server, the J2EE application server component stored all user information in a file called `principals.xml` (including storing passwords in cleartext). The OracleAS JAAS Provider provides a similar simple security model as a default, without storing passwords in cleartext. The OracleAS JAAS Provider also offers tight integration with Oracle Application Server Infrastructure Infrastructure (including OracleAS Single Sign-On and Oracle Internet Directory) out of the box.
- *Avoid writing custom user managers; instead, extend the OracleAS JAAS Provider, OracleAS Single Sign-On, and Oracle Internet Directory.* The OC4J container

continues to supply several methods and levels of extending security providers. Although you can extend the `UserManager` class to build a custom user manager, leveraging the rich functionality provided by the OracleAS JAAS Provider, OracleAS Single Sign-On, and Oracle Internet Directory gives you more time to focus on actual business logic instead of infrastructure code. Both OracleAS Single Sign-On and Oracle Internet Directory provide APIs to integrate with external authentication servers and directories respectively.

- *Use OracleAS Single Sign-On as the authentication mechanism with the OracleAS JAAS Provider.* The OracleAS JAAS Provider allows different authentication options. However, we strongly recommend leveraging the OracleAS Single Sign-On server whenever possible, for the following reasons:
  - It is the default mechanism for most Oracle Application Server components, such as Portal, Forms, Reports, and Wireless.
  - It is easy to set up in a declarative fashion and does not require any custom programming.
  - It provides seamless PKI integration.
- *Use the OracleAS JAAS Provider declarative features to reduce programming.* Because most of the features in the OracleAS JAAS Provider are controlled declaratively, particularly in the area of authentication, developers can postpone setup until deployment time. This not only reduces the programming tasks for integrating a JAAS based application, it enables the deployer to use environment-specific security models for that application.
- *Use the fine-grained access control offered by the OracleAS JAAS Provider and the Java permission model.* Unlike the "coarse-grained" J2EE authorization model as it exists today, the OracleAS JAAS Provider integrated with OC4J allows any protected resource to be modeled using Java permissions. The Java permission model (and associated Permission class) is extensible and allows a flexible way to define fine-grained access control.
- *Use Oracle Internet Directory as the central repository for the OracleAS JAAS Provider in production environments.* Although the OracleAS JAAS Provider supports a flat-file XML-based repository useful for development and testing environments, it should be configured to use Oracle Internet Directory for production environments. Oracle Internet Directory provides LDAP standard features for modeling administrative metadata and is built on the Oracle database platform, inheriting all the database properties of scalability, reliability, manageability, and performance.
- *Take advantage of the authorization features of the OracleAS JAAS Provider.* In addition to the authorization functionality defined in the JAAS 1.0 specification, the OracleAS JAAS Provider supports the following:
  - Hierarchical, role-based access control (RBAC)
  - The ability to partition security policy by subscriber (that is, for each user community)

  These extensions provide a more scalable and manageable framework for security policies covering a large user population.

A OracleAS JAAS Provider Samples

This appendix provides supplemental samples and standards. It contains the following samples:

- Sample: jazn-data.xml Configuration
- Sample: Modifying User Permissions

Sample: jazn-data.xml Configuration

This section presents a sample `jazn-data.xml` file which illustrates the specific standards that XML files must conform to. This `jazn-data.xml` file contains a realm, `jazn.com`, users, and roles.

Example A–1 Sample jazn-data.xml File

```
<?xml version="1.0" encoding="UTF-8" standalone='yes'?>
<!DOCTYPE jazn-data PUBLIC "JAZN-XML Data"
"http://xmlns.oracle.com/ias/dtds/jazn-data-9_04.dtd">
<jazn-data>

<!-- JAZN Realm Data -->
<jazn-realm>
  <realm>
    <name>jazn.com</name>
    <users>
      <user>
        <name>anonymous</name>
        <description>The default guest/anonymous user</description>
      </user>
      <user>
        <name>SCOTT</name>
        <display-name>SCOTT</display-name>
        <credentials>!TIGER</credentials>
      </user>
      <user>
        <name>admin</name>
        <display-name>OC4J Administrator</display-name>
        <description>OC4J Administrator</description>
        <credentials>!welcome</credentials>
      </user>
      <user>
        <name>user</name>
        <description>The default user</description>
        <credentials>!456</credentials>
      </user>
            <!-- users used for password hiding -->
      <user>
        <name>pwForScott</name>
        <description>Password for database user Scott</description>
        <credentials>!TIGER</credentials>
      </user>
      <user>
        <name>pwForSSL</name>
        <description>Password for ssl key and trust stores</description>
        <credentials>!123456</credentials>
      </user>
      <user>
        <name>pwForSystem</name>
        <description>Password for database system user </description>
        <credentials>!manager</credentials>
      </user>
    </users>
    <roles>
      <role>
        <name>administrators</name>
        <display-name>Realm Admin Role</display-name>
        <description>Administrative role for this realm.</description>
        <members>
          <member>
            <type>user</type>
            <name>admin</name>
          </member>
        </members>
      </role>
      <role>
        <name>users</name>
        <members>
          <member>
            <type>user</type>
            <name>user</name>
          </member>
          <member>
            <type>user</type>
            <name>SCOTT</name>
          </member>
          <member>
            <type>role</type>
            <name>administrators</name>
          </member>
        </members>
      </role>
      <role>
        <name>guests</name>
        <members>
          <member>
            <type>user</type>
            <name>anonymous</name>
          </member>
          <member>
            <type>role</type>
            <name>users</name>
          </member>
        </members>
      </role>
```

```
      <role>
        <name>jmxusers</name>
        <display-name>JMX users</display-name>
        <description>
            Allows access to application level user defined MBeans
        </description>
        <members>
        </members>
      </role>
    </roles>
  </realm>
</jazn-realm>

<!-- JAZN Policy Data -->
<jazn-policy>
  <grant>
    <grantee>
      <principals>
        <principal>
          <realm-name>jazn.com</realm-name>
          <type>role</type>
          <class>oracle.security.jazn.spi.xml.XMLRealmRole</class>
          <name>jazn.com/administrators</name>
        </principal>
      </principals>
    </grantee>
    <permissions>
      <permission>
        <class>oracle.security.jazn.policy.AdminPermission</class>
        <name>
           oracle.security.jazn.realm.RealmPermission$jazn.com$createrealm
        </name>
      </permission>
      <permission>
        <class>oracle.security.jazn.realm.RealmPermission</class>
        <name>jazn.com</name>
        <actions>createrealm</actions>
      </permission>
      <permission>
        <class>oracle.security.jazn.policy.AdminPermission</class>
        <name>oracle.security.jazn.realm.RealmPermission$jazn.com$droprealm</name>
      </permission>
      <permission>
        <class>oracle.security.jazn.policy.AdminPermission</class>
        <name>
           oracle.security.jazn.realm.RealmPermission$jazn.com$createrole<
        /name>
      </permission>
      <permission>
        <class>oracle.security.jazn.policy.AdminPermission</class>
        <name>oracle.security.jazn.policy.RoleAdminPermission$jazn.com/*$</name>
      </permission>
      <permission>
        <class>oracle.j2ee.server.AdministrationPermission</class>
        <name>administration</name>
        <actions>administration</actions>
      </permission>
      <permission>
        <class>oracle.security.jazn.realm.RealmPermission</class>
        <name>jazn.com</name>
```

```
          <actions>droprealm</actions>
        </permission>
        <permission>
          <class>oracle.security.jazn.realm.RealmPermission</class>
          <name>jazn.com</name>
          <actions>dropuser</actions>
        </permission>
        <permission>
          <class>oracle.security.jazn.policy.RoleAdminPermission</class>
          <name>jazn.com/*</name>
        </permission>
        <permission>
          <class>oracle.j2ee.server.rmi.RMIPermission</class>
          <name>login</name>
        </permission>
        <permission>
          <class>oracle.security.jazn.policy.AdminPermission</class>
          <name>
             oracle.security.jazn.realm.RealmPermission$jazn.com$modifyrealmmetadata
          </name>
        </permission>
        <permission>
          <class>oracle.security.jazn.realm.RealmPermission</class>
          <name>jazn.com</name>
          <actions>modifyrealmmetadata</actions>
        </permission>
        <permission>
          <class>oracle.security.jazn.policy.AdminPermission</class>
          <name>oracle.security.jazn.realm.RealmPermission$jazn.com$droprole</name>
        </permission>
      </permissions>
    </grant>
    <grant>
      <grantee>
        <principals>
          <principal>
            <realm-name>jazn.com</realm-name>
            <type>role</type>
            <class>oracle.security.jazn.spi.xml.XMLRealmRole</class>
            <name>jazn.com/users</name>
          </principal>
        </principals>
      </grantee>
      <permissions>
        <permission>
          <class>oracle.j2ee.server.rmi.RMIPermission</class>
          <name>login</name>
        </permission>
      </permissions>
    </grant>
    <grant>
      <grantee>
        <principals>
          <principal>
            <realm-name>jazn.com</realm-name>
            <type>role</type>
            <class>oracle.security.jazn.spi.xml.XMLRealmRole</class>
            <name>jazn.com/jmxusers</name>
          </principal>
        </principals>
      </grantee>
      <permissions>
        <permission>
          <class>oracle.j2ee.server.rmi.RMIPermission</class>
          <name>login</name>
        </permission>
      </permissions>
    </grant>

  </jazn-policy>

  <!-- Permission Class Data -->
  <jazn-permission-classes>
  </jazn-permission-classes>

  <!-- Principal Class Data -->
  <jazn-principal-classes>
  </jazn-principal-classes>

  <!-- Login Module Data -->
  <jazn-loginconfig>
    <application>
      <name>oracle.security.jazn.oc4j.JAZNUserManager</name>
      <login-modules>
        <login-module>
          <class>oracle.security.jazn.realm.RealmLoginModule</class>
          <control-flag>required</control-flag>
          <options>
            <option>
              <name>addAllRoles</name>
              <value>true</value>
            </option>
          </options>
        </login-module>
      </login-modules>
    </application>
    <application>
      <name>oracle.security.jazn.tools.Admintool</name>
      <login-modules>
        <login-module>
          <class>oracle.security.jazn.realm.RealmLoginModule</class>
          <control-flag>required</control-flag>
          <options>
            <option>
              <name>addAllRoles</name>
              <value>true</value>
            </option>
            <option>
              <name>debug</name>
              <value>false</value>
            </option>
          </options>
        </login-module>
      </login-modules>
    </application>
    <application>
      <name>oracle.security.jazn.oc4j.DigestAuthenticator</name>
      <login-modules>
        <login-module>
          <class>oracle.security.jazn.login.module.digest.DigestLoginModule</class>
```

```
          <control-flag>required</control-flag>
          <options>
            <option>
              <name>debug</name>
              <value>false</value>
            </option>
            <option>
              <name>addAllRoles</name>
              <value>true</value>
            </option>
          </options>
        </login-module>
      </login-modules>
    </application>
  </jazn-loginconfig>

</jazn-data>
```

Sample: Modifying User Permissions

Example A–2 demonstrates granting `java.io.FilePermission` to a user named `Jane.Smith`. The objects to be modified are presented in bold.

Table A–1 lists the objects in Example A–2.

Table A–1 Objects in Sample Modifying User Permissions Code

Objects	Names	Comments
`RealmUser user`	`Jane.Smith`	
`codesource cs`	`file:/home/task.jar`	
File path	report.data	Path is the path name of the file.
Sample organization	`abc.com`	`abc.com` does not appear in this code directly.
Sample external realm	`abcRealm`	

Example A–2 Modifying User Permissions

```
import oracle.security.jazn.*;
import oracle.security.jazn.policy.*;
import oracle.security.jazn.realm.*;
import java.lang.*;
import java.security.*;
import java.util.*;
import java.net.*;
import java.io.*;

public class Init {

    public static void main(String[] args) {

    try {
            JAZNConfig _jc = JAZNConfig.getJAZNConfig();
            RealmManager realmMgr = _jc.getRealmManager();
            Realm realm = realmMgr.getRealm("abcRealm");
            UserManager userMgr = realm.getUserManager();
            RoleManager roleMgr = realm.getRoleManager();
            final JAZNPolicy policy = _jc.getPolicy();
            final RealmUser user = userMgr.getUser("Jane.Smith");

            AccessController.doPrivileged (new PrivilegedAction() {
                    public Object run() {

                try {

                  CodeSource cs =
                        new CodeSource(new URL("file:/home/task.jar"), null);
                    HashSet prop = new HashSet();
                    prop.add((Principal) user);

                    // assign permission to principals
                    policy.grant(new Grantee(prop, cs), new
                             FilePermission("report.data", "read"));

                    return null;
                        } catch (JAZNException e1) {
                            e1.printStackTrace();
                        } catch (java.net.MalformedURLException e2) {
                            e2.printStackTrace();
                        }
                    return null;
                    }
                }
             );

        } catch (JAZNException e) {
            e.printStackTrace();
        }
    }
}
```

This sample code grants a user, `Jane.Smith`, permission to use the sample application, `AccessTest1`, as follows:

The name `cs` is assigned to the `file:/home/task.jar`, which includes the sample application `AccessTest1`:

```
CodeSource cs = new CodeSource(new URL("file:/home/task.jar"), null);
```

`Jane.Smith` is the user added to the `HashSet prop`:

```
HashSet prop = new HashSet();
prop.add((Principal) user);
```

`Jane.Smith` is granted permission, on the `Codesource cs`, to read the file `report.data`.

```
policy.grant(new Grantee(prop, cs), new FilePermission("report.data", "read"));
```

B JAZN Admintool Reference

The JAZN Admintool can manage both XML-based and LDAP-based JAAS configurations and data from the command prompt. The JAZN Admintool is a flexible Java console application, with functions that can be called directly from the command line or through an interactive shell. The JAZN Admintool is located in *ORACLE_HOME*`/j2ee/home/jazn.jar.`

Note: The JAZN Admintool manages only XML-based roles and users. To manage LDAP-based users and roles, use the Delegated Administration Service (DAS); see the *Oracle Internet Directory Administrator's Guide* for details.

This appendix is a central reference for how to perform common administration tasks using the JAZN Admintool (in some cases repeating information from earlier in this manual). It is divided into the following sections:

- Listing Principal Class Information
- Listing Realms
- Listing Roles
- Listing Users
- Migrating Principals from the principals.xml File
- Setting Passwords (XML-Based Provider Only)
- Using the JAZN Admintool Shell

Introduction to JAZN Admintool Command-Line Options and Syntax

This section introduces JAZN Admintool options, grouped according to function, and their syntax. The options are described in greater detail later in the appendix.

The overall syntax for the Admintool is as follows:

```
java -jar jazn.jar [-user username -password password -clustersupport oracle_home]
                   [otheroptions]
```

Note: If you use one or more of the `-user`, `-password`, and `-clustersupport` options, you must specify them before all other options on the command line.

The tool prints error messages if the syntax or parameters are incorrect. You can list all the options and their syntax with the `-help` option, as in the following example:

```
java -jar jazn.jar -help
```

Admintool Authentication (XML-Based Provider Only)

```
-user username password password
```

See Also:

- "Authentication and the JAZN Admintool (XML-Based Provider Only)" on page B-4

Clustering Operations

```
-clustersupport oracle_home
```

See Also:

- "Adding Clustering Support (XML-Based Provider Only)" on page B-6

Configuration Operations

```
-getconfig
```

See Also:

- "Configuration Operations" on page B-9

Interactive Shell

```
-shell
```

See Also:

- "Using the JAZN Admintool Shell" on page B-14

Login Modules

```
-addloginmodule application_name login_module_name control_flag [options]
-listloginmodules [application_name] [login_module_class]
-remloginmodule application_name login_module_name
```

See Also:

- "Adding and Removing Login Modules (XML-Based Provider Only)" on page B-6
- "Listing Login Modules" on page B-10

Migration Operations

```
-convert filename realm
```

See Also:

- "Migrating Principals from the principals.xml File" on page B-13

Password Management

```
-checkpasswd realm user [-pw password]
-setpasswd realm user old_pwd new_pwd
```

See Also:

- "Checking Passwords (XML-Based Provider Only)" on page B-9
- "Setting Passwords (XML-Based Provider Only)" on page B-14

Policy Operations

```
-addperm permission permission_class action target [description]
-addprncpl principlename principle_class parameters [description]
-grantperm {realm {-user user|-role role} | principal_class principal_params}
            permission_class [permission_params]
-listperms {realm {-user user|-role role} | principal_class principal_params}
            permission_class [permission_params]
-listperm permission
-listprncpls [principal_name]
-listprncpl principal_name
-remperm permission
-remprncpl principal_name
-revokeperm {realm {-user user|-role role} | principal_class principal_params}
             permission_class [permission_params]
```

See Also:

- "Adding and Removing Policy Permissions (XML-Based Provider Only)" on page B-5
- "Adding and Removing Principals (XML-Based Provider Only)" on page B-7
- "Granting and Revoking Permissions" on page B-9
- "Listing Permissions" on page B-11
- "Listing Permission Information" on page B-11
- "Listing Principal Classes" on page B-11
- "Listing Principal Class Information" on page B-12

Realm Options

```
-addrealm realm admin {adminpwd adminrole | adminrole
    userbase rolebase realmtype }
-addrole realm role
-adduser realm username password
-grantrole role realm {user | -role to_role}
-listrealms realm
-listroles [realm [user | -role role]]
-listusers [realm [-role role | -perm permission]]
-remrealm realm
-remrole realm role
-remuser realm user
-revokerole role realm {user | -role from_role}
```

See Also:

- "Adding and Removing Realms" on page B-7
- "Adding and Removing Roles (XML-Based Provider Only)" on page B-8
- "Adding and Removing Users (XML-Based Provider Only)" on page B-8
- "Granting and Revoking Roles" on page B-10
- "Listing Realms" on page B-12
- "Listing Roles" on page B-12
- "Listing Users" on page B-13

Help

```
-help  [command_name]
```

To display help for a specific command.

Authentication and the JAZN Admintool (XML-Based Provider Only)

If you use the XML-based provider, you must authenticate yourself to the JAZN Admintool before making administrative changes. You authenticate yourself in one of two ways:

- Supply the `-user` and `-password` switches, as in:

```
java -jar jazn.jar -user username -password password -listrealms
```

Note: If you use the `-user` and `-password` options, they must appear before all other options on the command line.

- Supply a user name and password when prompted by the Admintool, as in:

```
java -jar jazn.jar -listrealms
>RealmLoginModule username: martha
>RealmLoginModule password: mypass
```

Cautions:

- Although it is possible to supply the user name and password on the command line with `-user` and `-password`, we recommend that you avoid this; specifying passwords on command lines creates security vulnerabilities.
- Because of Java limitations, the password you type to the Admintool may be visible in the command window. Be sure to close the command window after using the Admintool.
- The Admintool does not require authentication when used with the LDAP-based provider; anyone who runs the tool can perform Admintool operations against the Oracle Internet Directory server. This means that it is vital to secure access to any production machine on which OC4J uses the LDAP-based provider. If you specify the `-user` and `-password` options when using the LDAP-based provider, they are ignored.

Adding and Removing Policy Permissions (XML-Based Provider Only)

```
-addperm permission permission_class action target [description]
-remperm permission
```

The `-addperm` option registers a permission with the JAAS provider `PermissionClassManager`. The `-remperm` option removes registration for the specified permission class. To supply multiple words in the *`permission`* or *`description`* arguments, enclose them in quotation marks ("*`three word permission`*").

If you add a permission that already exists, the Admintool updates the action and target lists of the permission.

For instance, to create permission to drop a realm:

```
java -jar jazn.jar -addperm perm1 oracle.security.jazn.realm.RealmPermission
     droprealm "permission to drop a realm"
```

To delete the `droprealm` permission:

```
java -jar jazn.jar -remperm perm1
```

Admintool shell:

```
JAZN:> addperm perm1 oracle.security.jazn.realm.RealmPermission droprealm -null
      "permission to drop a realm"
JAZN: remperm perm1
```

Adding Clustering Support (XML-Based Provider Only)

```
-clustersupport oracle_home
```

Specifying this option instructs the Admintool to propagate all JAAS configuration changes throughout a cluster. The *oracle_home* argument specifies the absolute path name of *ORACLE_HOME*, the Oracle home directory. You can combine -clustersupport with the -shell option.

Note: If you use the -clustersupport option, it must appear before all other options on the command line.

For example:

```
java -jar jazn.jar -clustersupport /oracle_home -shell
```

Adding and Removing Login Modules (XML-Based Provider Only)

Use the JAZN Admintool to add and remove login modules.

```
java -jar jazn.jar -addloginmodule application_name login_module_name
     control_flag [optionname=value ...]
java -jar jazn.jar -remloginmodule application_name login_module_name
```

The -addloginmodule option configures a new LoginModule for the named application.

The *control_flag* must be one of required, requisite, sufficient or optional, as specified in javax.security.auth.login.Configuration. See Table B–1.

See Also:

- "Admintool Overview" on page 4-3 for basic information on running the JAZN Admintool

Table B–1 Login Module Control Flags

Flag	Meaning
required	The LoginModule must succeed. Whether or not it succeeds, authentication proceeds down the LoginModule list.
requisite	The LoginModule must succeed. If it succeeds, authentication continues down the LoginModule list. If it fails, control immediately returns to the application (authentication does not continue down the LoginModule list).
sufficient	The LoginModule is not required to succeed. If it succeeds, control immediately returns to the application and authentication does not proceed down the LoginModule list. If it fails, authentication continues down the LoginModule list.
optional	The LoginModule is not required to succeed. Whether or not it succeeds, authentication proceeds down the LoginModule list.

If the `LoginModule` accepts its own options, you specify each option and its value as an *`optionname=value`* pair. Each `LoginModule` has its own individual set of options.

For example, to add `MyLoginModule` to the application `myapp` as a required module with `debug` set to `true`:

```
java -jar jazn.jar -addloginmodule myapp MyLoginModule required debug=true
```

To delete `MyLoginModule` from `myapp`:

```
java -jar jazn.jar -remloginmodule myapp MyLoginModule
```

Admintool shell:

```
JAZN:> addloginmodule myapp MyLoginModule required debug=true
JAZN: remloginmodule myapp MyLoginModule
```

Adding and Removing Principals (XML-Based Provider Only)

```
-addprncpl principal_name principal_class parameters [description]
-remprncpl principal_name
```

The `-addprncpl` option registers a principal with the JAAS provider `PrincipalClassManager`. The `-remprncpl` option removes registration for the specified principal class. To supply multiple words in the *`principal_name`* and *`description`* arguments, enclose them in quotation marks ("*`three word description`*").

If you add a principal that already exists, the Admintool updates the parameter list of the principal.

For example, to add the principal `staff`:

```
java -jar jazn.jar -addprincpl staff oracle.security.jazn.spi.xml.XMLRealmUser
  "a staff user"
```

Admintool shell:

```
JAZN:> addprincpl staff oracle.security.jazn.spi.xml.XMLRealmUser -null "a staff
user"
```

Adding and Removing Realms

```
-addrealm realm admin {adminpwd adminrole | adminrole userbase rolebase realmtype}
-remrealm realm
```

The `-addrealm` option creates a realm of the specified type with the specified name, and `-remrealm` deletes a realm.

For example, using the XML-based provider, the administrator `martha` with password `mypass` using role `hr` would add the realm `employees` as follows:

```
java -jar jazn.jar -addrealm employees martha mypass hr
```

Using the LDAP-based provider, the administrator `martha` using role `hr` would add the realm `employees` to userbase `ub` and rolebase `rb` in an external realm, as follows:

```
java -jar jazn.jar -addrealm employees martha hr ub rb external
```

Note: The *`realmtype`* argument is required only when using the LDAP-based provider. The possible values for *`realmtype`* are `external` or `application`.

In either environment, the administrator would delete `employees` as follows:

```
java -jar jazn.jar -remrealm employees
```

Adding and Removing Roles (XML-Based Provider Only)

-**addrole** *realm role*
-**remrole** *realm role*

The `-addrole` option creates a role in the specified realm; the `-remrole` option deletes a role from the realm.

Note: If you use the LDAP-based provider, `-addrole` and `-remrole` are supported only for application realms; they are not supported for external or identity management realms.

For example, to add the role `roleFoo` to the realm `foo`:

```
java -jar jazn.jar -addrole foo fooRole
```

To delete the role from the realm:

```
java -jar jazn.jar -remrole foo fooRole
```

Admintool shell:

```
JAZN:> remrole foo fooRole
```

Adding and Removing Users (XML-Based Provider Only)

-**adduser** *realm username password*
-**remuser** *realm user*

The `-adduser` option adds a user to a specified realm; the `-remuser` option deletes a user from the realm. For example, to add the user `martha` to the realm `foo` with the password `mypass`:

```
java -jar jazn.jar -adduser foo martha mypass
```

Notes:

- To insert a user with no password, end the command line with the `-null` option, as follows:

  ```
  jazn -jar jazn.jar -adduser foo martha -null
  ```

- If you use the LDAP-based provider, these commands will not work.

To delete `martha` from the realm, type:

```
java -jar jazn.jar -remuser foo martha
```

Admintool shell:

```
JAZN:> adduser foo martha mypass
```

Checking Passwords (XML-Based Provider Only)

-**checkpasswd** *realm user* [-pw *password*]

The `-checkpasswd` option indicates whether the given user requires a password for authentication.

When you specify `-checkpasswd` alone, the Admintool responds `"A password exists for this principal"` if the user has a password, or "`No password exists for this principal`" if the user has no password.

When you specify `-checkpasswd` together with the `-pw` option, the Admintool responds "`Successful verification of user/password pair`" if the user name and password pair are correct, or "`Unsuccessful verification of user/password pair`" if the user name or password is incorrect.

For example, to check whether the user `martha` in realm `foo` uses the password `Hello`, type:

```
java -jar jazn.jar -checkpasswd foo martha -pw Hello
```

Admintool shell:

```
JAZN:> checkpasswd foo martha -pw Hello
```

Configuration Operations

-**getconfig**

The `-getconfig` option displays the current configuration setting in `jazn.xml`. For example, to check the configuration settings for the realm `foo`:

```
java -jar jazn.jar -getconfig
```

Admintool shell:

```
JAZN:> getconfig foo
```

Granting and Revoking Permissions

-**grantperm** {*realm* {-user *user*|-role *role*} | *principal_class principal_parameters*}
permission_class [*permission_parameters*]
-**revokeperm** {*realm* {-user *user*|-role *role*} | *principal_class principal_parameters*}
permission_class [*permission_parameters*]
-**listperms** {*realm* {-user *user*|-role *role*} | *principal_class principal_parameters*}
permission_class [*permission_parameters*]

In this syntax, *`principal_class`* is the fully qualified name of a class that implements the principal interface (such as `com.sun.security.auth.NTDomainPrincipal`) and *`principal_parameters`* is a single String parameter.

The `-grantperm` option grants the specified permission to a user (when called with `-user`) or a role (when called with `-role)` or a principal. The `-revokeperm` option revokes the specified permission from a user or role or principal.

A *permission_descriptor* consists of the explicit class name of a permission (for example, `oracle.security.jazn.realm.RealmPermission`), its action, and its action and target parameters (for `RealmPermission`, `realmname action`). Note that there may be multiple action and target parameters.

For example, to grant `FilePermission` with target `a.txt` and actions "`read, write`" to user `martha` in realm `foo`:

```
java -jar jazn.jar -grantperm foo -user martha java.io.FilePermission
     a.txt read, write
```

Admintool shell:

```
JAZN:> grantperm foo -user martha java.io.FilePermission a.txt read, write
```

Granting and Revoking Roles

-**grantrole** *role realm* {*user*|-role *to_role*}
-**revokerole** *role realm* {*user*|-role *from_role*}

The `-grantrole` option grants the specified role to a user (when called with a user name) or a role (when called with `-role`). The `-revokerole` option revokes the specified role from a user or role.

Note: If you are using the LDAP-based provider, `-grantrole` and `-revokerole` are supported only for application realms; they are not supported for external or identity management realms.

For example, to grant the role `editor` to the user `martha` in realm `foo`:

```
java -jar jazn.jar -grantrole editor foo martha
```

Admintool shell:

```
JAZN:> grantrole editor foo martha
```

Listing Login Modules

-**listloginmodules** [*application_name*] [*login_module_class*]

Use the JAZN Admintool to list login modules.

See Also:

- "Admintool Overview" on page 4-3 for basic information on running the JAZN Admintool

java -jar jazn.jar -**listloginmodules** [*application_name* [*login_module_class*]]

The `-listloginmodules` option displays all login modules, either in the specified *application_name* , or, if no *application_name* is specified, in all applications. Specifying *login_module_class* after *application_name* displays information on only the specified class within the application.

For example, to display all login modules or the application `myapp`:

```
java -jar jazn.jar -listloginmodules myapp
```

Admintool shell:

```
JAZN:> listloginmodules myapp
```

Listing Permissions

-**listperms** {*realm* {-user *user*|-role *role*} | *principal_class principal_parameters*}
permission_class [*permission_parameters*]

The `-listperms` option displays all permissions that match the list criteria. This option lists the following:

- All permissions registered with the JAAS provider `PermissionClassManager`
- Permissions that are granted to a role when the `-role` option is used
- Permissions that are granted to a `principal`

For example, to display all permissions for the user `martha` in realm `foo`:

```
java -jar jazn.jar -listperms foo -user martha
```

Admintool shell:

```
JAZN:> listperms foo -user martha
```

Listing Permission Information

-**listperm** *permission*

The `-listperm` option displays detailed information about the specified permission, including the display name, class, description, actions, and targets.

For example, to list all information about the permission `perm1`:

```
java -jar jazn.jar -listperm perm1
```

Typical output might look like the following:

```
Name:
perm1

Class:
oracle.security.jazn.realm.RealmPermission

Description:
permission to drop realm

Targets:

Actions:
droprealm  <no description available>
```

Admintool shell:

```
JAZN:> listperm perm1
```

Listing Principal Classes

-**listprncpls** *principal_name*

The `-listprncpls` option lists all principal classes registered with the `PrincipalClassManager`. If the *`principal_name`* argument is present, only the named principal class is listed.

For example:

```
java -jar jazn.jar -listprncpls
```

Admintool shell:

```
JAZN:> listprncpls
```

Listing Principal Class Information

-listprncpl *principal_name*

The `-listprncpl` option displays detailed information about the specified principal, including the display name, class, description, and actions.

For example, to list all information about the principal `martha`:

```
java -jar jazn.jar -listprncpl martha
```

In this example, the output would be:

```
Name:
martha
Class:
oracle.security.jazn.spi.xml.XMLRealmUser
Description:
a staff user
Parameters:
```

Admintool shell:

```
JAZN:> listprncpl martha
```

Listing Realms

-listrealms [*realm*]

The `-listrealms` option displays all realms in the current JAAS environment; if an argument is specified, it lists only the specified realm.

For example, to list all realms:

```
java -jar jazn.jar -listrealms
```

Admintool shell:

```
JAZN:> listrealms
```

Listing Roles

-listroles [*realm* [*user* | -role *role*]]

The `-listroles` option displays a list of roles that match the list criteria. This option lists:

- All roles in all realms, when called without any parameters
- All roles granted to a user, when called with a realm name and user name
- Roles that are granted the specified *`role`*, when called with a realm name and the option `-role`

For example, to list all roles in realm `foo`:

```
java -jar jazn.jar -listroles foo
```

Admintool shell:

```
JAZN:> listroles foo
```

Listing Users

-**listusers** [*realm* [-role *role* | -perm *permission*]]

The `-listusers` option displays a list of users that match the list criteria. This option lists:

- All users in all realms, when called without any parameters
- All users in a realm, when called with a realm name
- Users that are granted a certain role or permission, when called with a realm name and the option `-role` or `-perm`

For example, to list all users in realm `foo`:

```
java -jar jazn.jar -listusers foo
```

For example, to list all users in realm `foo` using permission `bar`:

```
java -jar jazn.jar -listusers foo -perm bar
```

The Admintool lists users one to a line:

```
scott
admin
anonymous
```

Admintool shell:

To list all users in the realm `foo`:

```
JAZN:> listusers foo
```

Migrating Principals from the principals.xml File

Use the JAZN Admintool to migrate your data out of the `principals.xml` file. For basic information on running the JAZN Admintool, see "Admintool Overview" on page 4-3.

-**convert** *filename realm*

The `-convert` option migrates the `principals.xml` file into the specified realm of the current OracleAS JAAS Provider. The *`filename`* argument specifies the path name of the input file (typically *`ORACLE_HOME`*`/j2ee/home/config/principals.xml`).

The migration converts `principals.xml` users to JAAS users and `principals.xml` groups to JAAS roles. All permissions that were previously granted to a `principals.xml` group are mapped to the JAAS role. Users that were deactivated at the time of migration are not migrated. This ensures that no users can inadvertently gain access through the migration.

An error (either `javax.naming.AuthenticationException:Invalid username/password` or `javax.naming.NamingException:Lookup Error`) is returned if the input file contains errors.

Before you convert `principals.xml`, make sure that you have an administrator user that is authorized to manage realms. To do this:

1. Activate the administrative user in `principals.xml`, which is deactivated by default. Be sure to create a password for the administrator.

   Make sure that the administrator name you used to create the realm is different from the name of the administrator in `principals.xml`. This is necessary because the `convert` command does not migrate duplicate users, and migrates duplicate roles by overwriting the old one.

2. Create the realm `principals.com` with a dummy user and a dummy role. For example, in the Admintool shell:

   ```
   JAZN> addrealm principals.com u1 welcome r1
   ```

3. Migrate `principals.xml` to the `principals.com` realm, as in the following example:

   ```
   java -jar jazn.jar -convert config/principals.xml principals.com
   ```

4. Change the `<default-realm>` to `principals.com`, as discussed in "Setting Persistence Mode" on page 8-4.

5. Stop OC4J and restart it.

Setting Passwords (XML-Based Provider Only)

`-`**`setpasswd`** *`realm user old_pwd new_pwd`*

The `-setpasswd` option enables administrators to reset the password of a user given the old password. For example, to change the user `martha` in realm `foo` from password `mypass` to password `a2d3vn`:

```
java -jar jazn.jar -setpasswd foo martha mypass a2d3vn
```

Admintool shell:

```
JAZN:> setpasswd foo martha mypass a2d3vn
```

Using the JAZN Admintool Shell

`-`**`shell`**

The `-shell` option starts a JAZN Admintool shell. The JAZN Admintool shell provides interactive administration of JAAS principals and policies through a UNIX-derived interface.

```
java -jar jazn.jar -user martha -password mypass -shell
JAZN:>
```

The shell responds with the `JAZN:>` prompt. To leave the interface shell, type `exit`.

Note: Multiple-word arguments must be enclosed in quotes. For example:

```
java -jar jazn.jar -user 'Oracle DBA' ...
```

If you are using the XML-based provider you must supply a user name and password to the Admintool, as discussed in "Authentication and the JAZN Admintool

(XML-Based Provider Only)" on page B-4. If you use the LDAP-based provider, you do not need to specify the `-user` and `-password` arguments.

JAZN Admintool Shell Commands

The Admintool shell supports UNIX-like commands for navigating within a JAZN structure. All the Admintool commands support relative and absolute paths. This section documents command syntax.

See Also:

- "Admintool Shell Directory Structure" on page B-16 for a complete discussion of the Admintool directory structure

The Admintool navigation commands are: `add`, `cd`, `clear`, `exit`, `help`, `ls`, `man`, `pwd`, `rm`, and `set`.

add: Creating Provider Data

```
add directory_name [other_parameter]
mkdir directory_name [other_parameter]
mk directory_name [other_parameter]
```

The `add`, `mkdir`, and `mk` commands are synonyms—they create a subdirectory or node in the current directory. For example, if the current directory is the root, then `mk` creates a realm. If the current directory is `/realm/users`, then `mk` creates a user. The effect of `add` depends upon the current directory. Some commands require additional parameters in addition to the name.

cd: Navigating Provider Data

```
cd path
```

The `cd` command enables users to navigate the directory tree. Relative and absolute path names are supported. To navigate to a directory `mydir`:

```
cd mydir
```

The command "`cd /`" returns the user to the root node. An error message is displayed if the specified directory does not exist.

clear: Clearing the Screen

```
clear
```

The `clear` command clears the terminal screen by displaying 80 blank lines.

exit: Exiting the JAZN Shell

```
exit
```

The `exit` command exits the JAZN shell.

help: Listing JAZN Admintool Shell Commands

```
help
```

The `help` command displays a list of all valid commands.

ls: Listing Data

```
ls [path]
```

The `ls` command lists the contents of the current directory or node. For example, if the current directory is the root, then `ls` lists all realms. If the current directory is `/realm/users`, then `ls` lists all users in the realm. The results of the listing depends on the current directory. The `ls` command can operate with the "`*`" wildcard.

man: Viewing JAZN Admintool Man Pages

`man` *`command_option`*
`man` *`shell_command`*

The `man` command displays detailed usage information for the specified shell command or JAZN Admintool command option. Where information presented by the `man` page and this document conflict, this document contains the correct usage for the command.

pwd: Displaying the Working Directory

`pwd`

The `pwd` command displays the current location of the user in the directory tree. Undefined values are left blank in this listing.

rm: Removing Provider Data

`rm` *`directory_name`*

The `rm` command removes the directory or node in the current directory. For example, if the current directory is the root, then `rm` removes the specified realm. If the current directory is `/realm/users`, it removes the specified user. The effect of `rm` depends on the current directory. An error message is displayed if the specified directory does not exist.

The `rm` command accepts the "`*`" wildcard.

set: Updating Values

`set` *`name=value`*

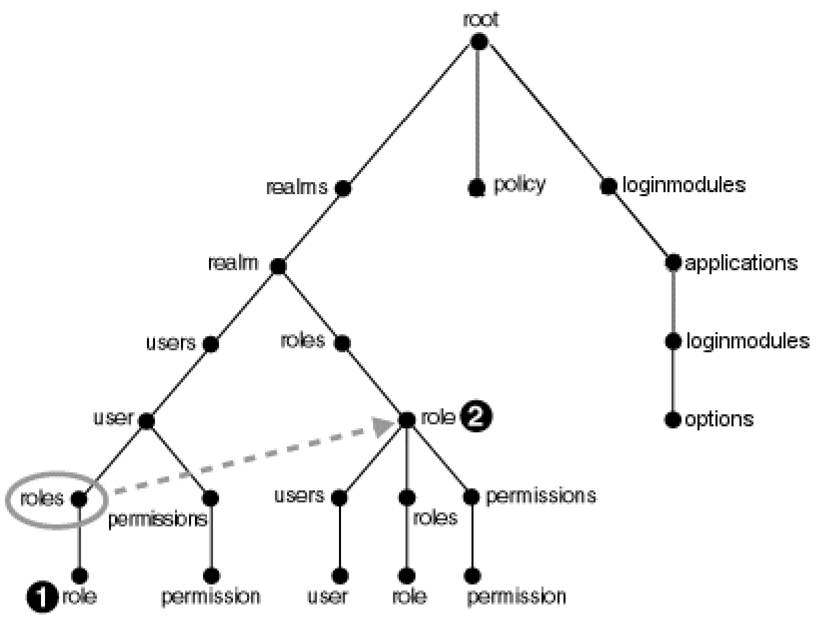
The `set` command updates the value of the specified name. For example, use this command to update the login module class, or a login module control flag, or a login module class option, depending on the working directory.

Admintool Shell Directory Structure

The JAZN Admintool includes a shell called the JAZN shell interface. The JAZN shell is an interactive interface to the JAAS Provider API.

The shell directory structure consists of nodes, where nodes contain subnodes that represent the properties of the parent node. Figure B–1 illustrates the node structure.

Figure B–1 JAZN Shell Directory Structure

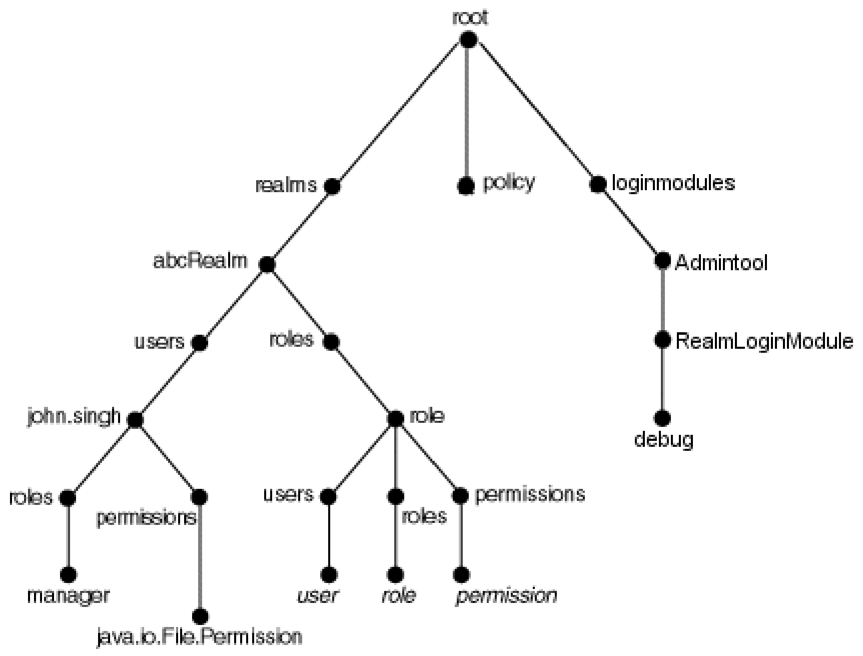


In this structure, the `user` and `role` nodes are linked together. This means that the `roles` link under `user` is the same link as the `roles` link under `realm`. In UNIX terms, the `role` at numeral 1 in the diagram is a symbolic link to `role` at numeral 2 in the diagram.

Note: In this release, the policy directory is always empty.

Figure B–2 shows nodes of the `abcRealm` created by the `jazn-data.xml` file in "Sample: jazn-data.xml Configuration" on page A-1.

Figure B–2 Illustrated Shell Directory Structure



Index