BEA AquaLogic Enterprise Security™

Developing Security Providers

Version 3.0
Revised: December 2007

Contents

5. Developing Custom Security Providers

Introduction and Roadmap

The following sections describe the content and organization of this document:

Scope

This book is designed for security and application developers who want to write their own security providers for use with BEA AquaLogic Enterprise Security. It is assumed that those using this document are application developers who have a solid understanding of security concepts, and that no basic security concepts require explanation. It is also assumed that security and application developers are familiar with BEA AquaLogic Enterprise Security and with Java programming.

Prerequisites for This Document

Prior to reading this guide, you should read the *Introduction to BEA AquaLogic Enterprise Security*. This document describes how the product works and provides conceptual information that is helpful to understanding the necessary installation components.

Documentation Audience

This document is intended for the following audiences:

- Application Developers—Developers who are Java programmers who focus on developing Java applications, incorporating security into Java applications and Enterprise JavaBeans (EJBs), and who work with other engineering, quality assurance (QA), and database teams to implement security features. Application Developers have in-depth working knowledge of Java (including J2EE components such as servlets/JSPs and JSEE).
- Security Architects—Individuals who are responsible for designing and implementing the overall security architecture for their organization, evaluating BEA AquaLogic Enterprise Security features, and determining how to best implement policies. Security Architects have in-depth knowledge of Java programming, Java security, and network security, as well as knowledge of security systems and leading-edge security technologies and tools.
- Security Developers—Developers (including third-party developers) who focus on defining the system architecture and infrastructure for security products and who develop custom security providers for use with BEA AquaLogic Enterprise Security services. Security Developers work with Security Architects to ensure that the architecture is implemented according to design specifications and that it does not introduce any security holes. Security Developers also work with administrators to ensure that security is properly configured. Security Developers have a solid understanding of certain concepts, including authentication, authorization, and auditing, and an in-depth knowledge of Java and security provider functionality.

Guide to this Document

This document provides application developers with the information needed to develop custom security providers for use with BEA AquaLogic Enterprise Security™ Security Service Modules. This document is organized as follows:

- Chapter 2, “Introduction to Developing Security Providers,” specifies the audience and prerequisites for this guide, and provides an overview of the development process.
- Chapter 3, “Security Provider Concepts,” explains the concepts that you must understand to be able to develop custom security providers. This topic also includes a discussion about JAAS Login Modules.
- Chapter 4, “Design Considerations,” provides background information about implementing Security Services Provider Interfaces (SSPIs) and generating MBean types.

- Chapter 5, "Developing Custom Security Providers," provides instructions for implementing each type of security provider.
- Chapter 6, "Auditing Events from Custom Security Providers," explains how to add auditing capabilities to custom security providers.
- Chapter 7, "Code Examples for Developing Security Providers," demonstrates how to write the code when developing custom security providers.
- Appendix A, "MBean Definition File Element Syntax," describes the attributes and syntax of the MBean Definition File.

Related Information

Additional documents include:

- *Introduction to AquaLogic Enterprise Security*—This document summarizes ALES features and provides an overview of the architecture.
- *Javadocs for Security Service Provider Interfaces*—This document provides reference documentation for ALES Security Service Provider Interfaces.
- *Policy Managers Guide*—This document defines the ALES policy model.
- *Javadocs for BLM API*—This document provides reference documentation for the Business Logic Manager (BLM) Application Programming Interfaces. This API can be used to write, manage, and distribute access control policy (users, groups, roles, resources, and authorization and role mapping policies).
- *Programming Security for Java Applications*—Describes how to implement security in Java applications. It includes descriptions of the Security Service Application Programming Interfaces and programming instructions.
- *Java API*—Provides Javadoc documentation for the ALES Java Application Programming Interfaces.
- *Programming Security for Web Services*—Describes how to implement security in web servers using the Web Services SSM. It includes descriptions of the Web Services API.
- *Web Services API*—WSDL generated documentation Web Services SSM interface.

CHAPTER 2

Introduction to Developing Security Providers

This section cover the following topics:

- “Overview of the Development Process” on page 2-1
- “Types of Providers” on page 2-5

Overview of the Development Process

To develop a custom security provider, you perform the following tasks:

1. Make security provider design decisions.
2. Write an MBean Definition File for each security provider you want to develop.
3. Run each MBean Definition File file through the WebLogic MBean Maker.
4. Create the runtime classes for each security provider you want to develop.
5. Create the auditing event classes for each security provider from which events are audited (optional).
6. Run the files generated by the MBean Maker **and the runtime class files** through the WebLogic MBean Maker to produce an MBean JAR file (MJF).

   Note: An MBean JAR file can contain multiple security providers. Therefore you only need to run the MBean Maker once to produce the MBean JAR file.

7. Deploy the MJF file to the BEA AquaLogic Enterprise Security systems from which you want to use the providers:

For WebLogic Server version 8.1 providers, this includes copying the JAR file to both the Administration Application and the Security Service Module provider directories.

For providers created with the WebLogic Server 9.x\10.0 WebLogicMBeanMaker, this includes copying the JAR file to the WebLogic Server deployment directory.

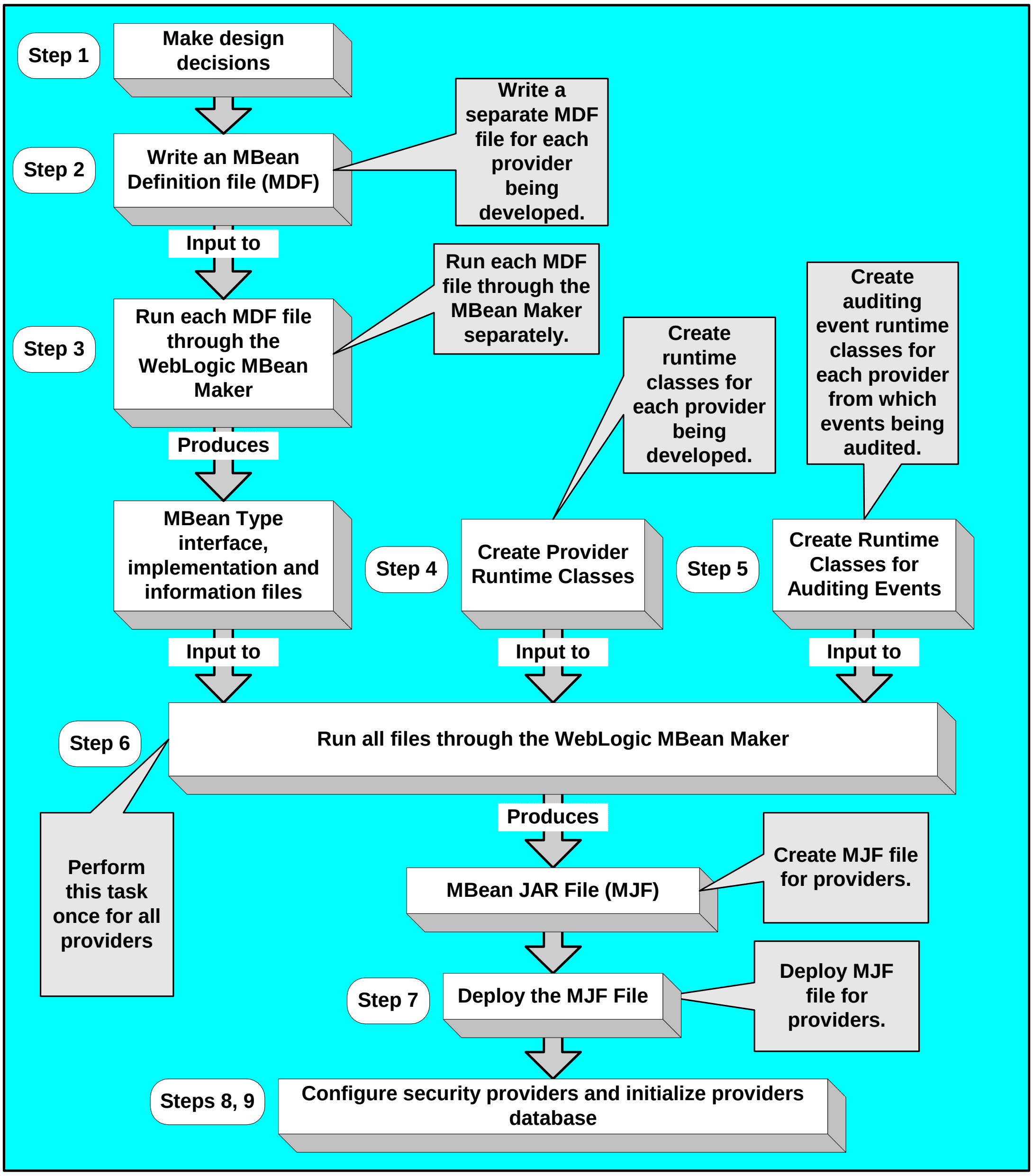
8. For WebLogic Server 8.1 providers, use the BEA AquaLogic Enterprise Security Administration Console to configure the security providers.

   For providers created with the WebLogic Server 9.x\10.0 WebLogicMBeanMaker, use the WebLogic Server Administration Console to configure the security providers, as described in "Security Provider Management Concepts" on page 3-2.

9. Initialize the security provider databases.

Figure 2-1 illustrates the security provider development process. For detailed instructions for each of the development tasks, see Chapter 5, "Developing Custom Security Providers."

Figure 2-1 Developing Custom Security Provider Tasks


Step 1
Make design decisions
Step 2
Write an MBean Definition file (MDF)
Write a separate MDF file for each provider being developed.
Input to
Step 3
Run each MDF file through the WebLogic MBean Maker
Run each MDF file through the MBean Maker separately.
Produces
MBean Type interface, implementation and information files
Step 4
Create Provider Runtime Classes
Create runtime classes for each provider being developed.
Step 5
Create Runtime Classes for Auditing Events
Create auditing event runtime classes for each provider from which events being audited.
Input to
Input to
Input to
Step 6
Run all files through the WebLogic MBean Maker
Perform this task once for all providers
Produces
MBean JAR File (MJF)
Create MJF file for providers.
Step 7
Deploy the MJF File
Deploy MJF file for providers.
Steps 8, 9
Configure security providers and initialize providers database

Types of Providers

You use the SSPI provided with the product to create runtime classes for custom security providers, which are located in the `weblogic.security.spi` package. For more information about this package, see *Javadocs for Security Service Provider Interfaces*.

Note: You can use the WebLogic Server 9.x\10.0 WebLogicMBeanMaker to create any of the security provider types described in Developing WebLogic Security Providers. However, doing so affects how you then manage that provider, as described in "Security Provider Management Concepts" on page 3-2.

Table 2-1 maps the types of security providers and their components with the SSPI and other interfaces you use to develop them. Table 2-1 includes the WebLogic Server 9.x\10.0 providers.

Table 2-1 Security Providers, Components, and Corresponding SSPI

Type/Component	Interface
Identity Assertion provider	`AuthenticationProvider` `AuthenticationProviderV2`
Identity Asserter	`IdentityAsserter` `IdentityAsserterV2`
Principal Validation provider	`PrincipalValidator`
Authorization	`AuthorizationProvider` `DeployableAuthorizationProviderV2`
Access Decision	`AccessDecision`
Adjudication provider	`AdjudicationProvider` `AdjudicationProviderV2`
Adjudicator	`Adjudicator` `AdjudicatorV2`
Role Mapping provider	`RoleProvider` `DeployableRoleProviderV2`
Role Mapper	`RoleMapper`
Auditing provider	`AuditProvider`
Audit Channel	`AuditChannel`

Table 2-1 Security Providers, Components, and Corresponding SSPI

Type/Component	Interface
Credential Mapping provider	`CredentialProvider` `CredentialProviderV2`
Credential Mapper	`CredentialMapper` `CredentialMapperV2`
Cert Path Provider	`CertPathProvider`
Versionable Application Provider	`VersionableApplicationProvider`
Servlet Authentican Filter	`ServletAuthenticationFilter`

Security Provider Concepts

To develop custom security providers, you need to know and understand the security concepts that relate to the type of security providers you are developing. This section describes the concepts of the following types of security providers.

- "Authentication Concepts" on page 3-3
- "Identity Assertion Concepts" on page 3-9
- "Principal Validation Concepts" on page 3-12
- "Authorization Concepts" on page 3-14
- "Role Mapping Concepts" on page 3-14
- "Auditing Concepts" on page 3-15
- "Credential Mapping Concepts"

This section described the most common security provider types. You can create and use any of the security provider types described in the WebLogic Server-specific version of Developing WebLogic Security Providers.

This section begins with a description of the security provider management concepts.

Security Provider Management Concepts

Before you create a custom security provider, you need to understand how that provider will be managed by BEA AquaLogic Enterprise Security. The management model depends on the type of provider that you create and whether you then upgrade that provider.

As described in "Overview of the Development Process" on page 2-1 once your have run your MDF through the WebLogic MBeanMaker to generate your intermediate files, and you have edited the MBean implementation file to supply implementations for the appropriate methods within it, you need to package the MBean files **and the runtime classes for the custom security provider** into an MBean JAR File (MJF).

The version of WebLogic MBeanMaker included with version 9.x\10.0 of WebLogic Server uses a different packaging model than was used in previous versions of WebLogic MBeanMaker, and the resultant MJF files reflect this packaging change. The packaging change is an internal implementation detail and is not generally exposed to your custom security provider. However, this packaging change is detected by BEA AquaLogic Enterprise Security when it attempts to load the MJF files. Therefore, the following conditions apply:

- If you use the version 8.1 WebLogicMBeanMaker to create a custom security provider MJF, the MJF file can be loaded and administratively managed by the BEA AquaLogic Enterprise Security Administrative Console.
- If you use the version 9.x\10.0 WebLogicMBeanMaker to create a custom security provider MJF, that security provider can be administratively managed only by the WebLogic Server Administration Console, WebLogic Scripting Tool (WLST), or some other WebLogic Server mechanism. The BEA AquaLogic Enterprise Security Administrative Console will not be able to load the MJF file.
- If you use the version 9.x\10.0 WebLogic Upgrade Wizard to upgrade a custom security provider for use in WebLogic Server 9.x\10.0, that security provider can be administratively managed only by WebLogic Server Administration Console, WebLogic Scripting Tool (WLST), or some other WebLogic Server mechanism. There is no backward compatibility for the upgrade: the BEA AquaLogic Enterprise Security Administrative Console will not be able to load the MJF file.

How is Your Custom Provider Going to Be Called?

"Security Provider Management Concepts" on page 3-2 describes how a custom security provider will be administratively managed by BEA AquaLogic Enterprise Security. However, as a custom security provider writer, you may find it more convenient to approach the management concept from the perspective of how your custom security provider will be called.

Your custom security provider SSPI interface will be called by the Security Framework that is logically contained by a Security Service Module (SSM). The type of MBean JAR File (MJF) that you create for your custom security provider determines which SSM will be able to call the security provider SSPI. Therefore, if you know the SSM in which you plan to deploy your custom security provider, you can determine which type of MJF file you must create:

- WLS SSM, Web Services SSM, Java SSM. These SSMs require you to use the 9.x\10.x WebLogic MBeanMaker or WebLogic Upgrade Wizard to create the custom security provider.
- WLS8.1 SSM. This SSM requires you to use the 8.1 WebLogic MBeanMaker to create the custom security provider.

Extended WebLogic Security Service Provider Interface (SSPI)

For providers created with the WebLogic Server WebLogicMBeanMaker, you use the WebLogic Server Administration Console to configure the security providers, as described in "Security Provider Management Concepts" on page 3-2.

However, you can use the WebLogic 9.x\10.0 SSM and ALES Java API to interact with these providers. If you do so, the providers must handle the extended WebLogic Security Service Provider Interface (SSPI), as described in "Developing Security Providers using the SSPI" on page 4-6.

Authentication Concepts

Before delving into the specifics of developing custom Authentication providers, it is important to understand the following concepts:

- "Users and Groups, Principals and Subjects" on page 3-3
- "Java Authentication and Authorization Service (JAAS)" on page 3-4

Users and Groups, Principals and Subjects

A **user** typically represents a person. A **group** is a category of users, classified by common traits such as job title. Categorizing users into groups makes it easier to control the access permissions for large numbers of users. Both users and groups can be used as principals. A **principal** is an identity assigned to a user or group as a result of authentication. The Java Authentication and Authorization Service (JAAS) requires that **subjects** be used as containers for authentication information, including principals. Each principal stored in the same subject represents a separate aspect of the same user's identity, much like cards in a person's wallet. (For example, an ATM card identifies someone to their bank, while a membership card identifies them to a professional organization to which they belong.) For more information about JAAS, see "Java Authentication and Authorization Service (JAAS)" on page 3-4. For additional information on the relationship of users and groups and the authorization service, see the Security Services in the *Introduction to AquaLogic Enterprise Security*.

As part of a successful authentication, principals are signed and stored in a subject for future use. A Principal Validation provider signs principals, and an Authentication provider LoginModule actually stores the principals in the subject. Later, when a caller attempts to access a principal stored within a subject, a Principal Validation provider verifies that the principal has not been altered since it was signed, and the principal is returned to the caller (assuming all other security conditions are met).

Note: For more information about Principal Validation providers and LoginModules, see "Principal Validation Concepts" on page 3-12 and "Writing a JAAS LoginModule" on page 3-5.

Any principal that is going to represent a user or group needs to implement the `WLSUser` and `WLSGroup` interfaces, available in the `weblogic.security.spi` package.

Java Authentication and Authorization Service (JAAS)

Whether the client is an application, applet, Enterprise JavaBean (EJB), or servlet that requires authentication, the Java Authentication and Authorization Service (JAAS) classes allow you to reliably and securely authenticate to the client. JAAS implements a Java version of the Pluggable Authentication Module (PAM) framework that permits applications to remain independent from underlying authentication technologies. Therefore, the PAM framework allows the use of new or updated authentication technologies without requiring modifications to your application. Authentication providers use JAAS internally for authentication. Therefore, only developers of custom Authentication providers need to be concerned with JAAS implementation.

This section covers the following topics:

- "Writing a JAAS LoginModule" on page 3-5
- "LoginModule Interface" on page 3-5

- "JAAS Control Flags" on page 3-6
- "CallbackHandlers" on page 3-7

Writing a JAAS LoginModule

Each Authentication Provider requires a LoginModule. LoginModules are responsible for authenticating users within the policy domain and for populating a subject with the necessary principals (users and groups). LoginModules that are *not* used for perimeter authentication also verify the proof material submitted (for example, a password).

If there are multiple Authentication Providers configured within a policy domain, each one requires a LoginModule to store principals within the same subject. Therefore, if a principal that represents a user named *Joe* is added to the subject by one Authentication Provider LoginModule, any other Authentication Provider in the policy domain should be referring to the same person when they encounter *Joe*. In other words, the other Authentication Provider LoginModule should not attempt to add another principal to the subject that represents a user (for example, named *Joseph*) to refer to the same person. However, it is acceptable for another Authentication Provider LoginModule to add a principal of a type other than the name *Joseph*.

LoginModule Interface

You can write LoginModules that handle a variety of authentication mechanisms, including username and password combinations, smart cards, and biometric devices. You develop LoginModules by implementing the `javax.security.auth.spi.LoginModule` interface, that is based on the Java Authentication and Authorization Service (JAAS) and uses a subject as a container for authentication information. The LoginModule interface enables you to plug in different kinds of authentication technologies for use with a single application and the Security Framework supports multiple LoginModule implementations for multi-part authentication.

You can also have dependencies across LoginModule instances or share credentials across those instances. However, the relationship between LoginModules and Authentication providers is one-to-one. In other words, to have a LoginModule that handles a retina scan authentication and a LoginModule that interfaces to a hardware device like a smart card, you must develop and configure two Authentication providers, each of which includes an implementation of the LoginModule interface. For more information, see "Implementing the JAAS LoginModule Interface" on page 5-9.

Note: You can also obtain LoginModules from third-party security vendors instead of developing your own.

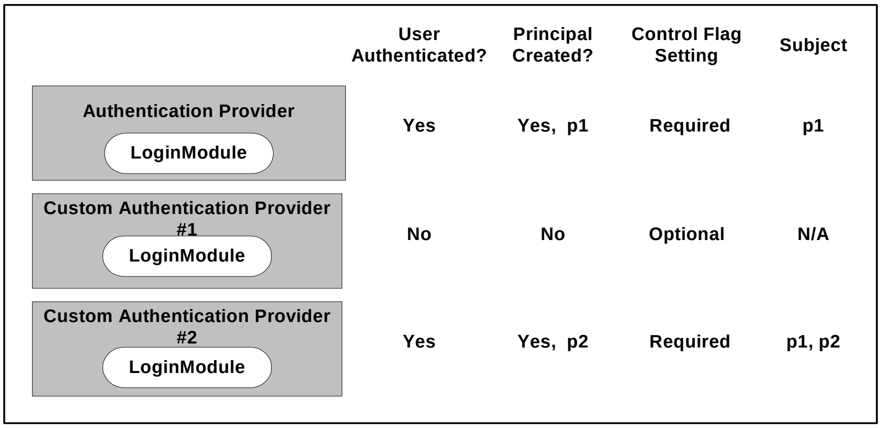
JAAS Control Flags

If a policy domain has multiple Authentication Providers configured, the Control Flag attribute on the Authenticator Provider determines the order of execution. Generally, you configure the control flow of multiple Authentication providers using the Administration Console. For more information on specifying the order or authentication providers, see "JAAS Control Flags" on page 3-6. Setting each LoginModule control flag specifies how to handle a failure during the authentication process. The values for the Control Flag attribute are:

- REQUIRED—The Authentication provider is called and the user must pass its authentication test. However, this test is not run if the SUFFICIENT authentication test is already satisfied. This setting is the default.
- REQUISITE—This LoginModule must succeed. If other Authentication Providers are configured and this LoginModule succeeds, authentication proceeds down the list of LoginModules. Otherwise, return control to the application.
- SUFFICIENT—This LoginModule needs not succeed. If it does succeed, return control to the application. If it fails and other Authentication Providers are configured, authentication proceeds down the LoginModule list.
- OPTIONAL—The user is allowed to pass or fail the authentication test of this Authentication Providers. However, if all Authentication Providers configured in a policy domain have the Control Flag set to OPTIONAL, the user must pass the authentication test of one of the configured providers.

Figure 3-1 illustrates a sample flow involving three different LoginModules that are part of three Authentication providers, and illustrates what happens to the subject for different authentication outcomes.

Figure 3-1 Sample LoginModule Flow



If you set the control flag for Custom Authentication Provider #1 to Required, the authentication failure in the User Authentication step causes the entire authentication process to fail. Also, if the user was not authenticated by the WebLogic Authentication provider (or custom Authentication provider #2), the entire authentication process fails. If the authentication process had failed in any of these ways, all three LoginModules would have been rolled back and the subject would not contain any principals.

Note: For more information about the LoginModule control flag setting and the LoginModule interface, see the *Java Authentication and Authorization Service (JAAS) 1.0 LoginModule Developer's Guide* and the *Java 2 Enterprise Edition, v1.4.2 API Specification Javadoc* for the LoginModule interface, respectively.

CallbackHandlers

A `CallbackHandler` is a highly-flexible JAAS standard that allows a variable number of arguments to be passed as complex objects to a method. An application implements a `CallbackHandler` and passes it to underlying security services so that they may interact with the application to retrieve specific authentication data, such as usernames and passwords, or to display certain information, such as error and warning messages.

`CallbackHandlers` are implemented in an application-dependent fashion and the application developer must implement one for his application. For example, an HTML form (such as, a login page) could prompt the user for information or display an error message. Another implementation might choose to obtain information from an alternate source without asking the user.

Underlying security services make requests for different types of information by passing individual Callbacks to the `CallbackHandler`. The `CallbackHandler` implementation decides how to retrieve and display information depending on the Callbacks passed to it. For example, if the underlying service needs a username and password to authenticate a user, the service uses a `NameCallback` and `PasswordCallback`. The `CallbackHandler` can then request a username and password serially, or request both from a single pop-up window.

How JAAS Works

Authentication using the JAAS classes and the Security Framework is performed in the following manner:

1. The client application creates a callback handler containing a callback that allows a provider to request authentication information from the application.
2. The client application passes the callback handler through the authentication service of the Java API into the Security Framework.
3. The Security Framework presents the callback handler to the LoginModule for the appropriate authentication provider.
4. The LoginModule uses the callback handler to request specific authentication information (e.g., username or password).
5. The client application is responsible for collecting the appropriate information to respond to the authentication callback. For example, this may include prompting for a username or password.
6. After the LoginModule collects all of the information required, it does one of the following:

   Authentication success, returns a valid subject:

   – Principals (users and groups) are signed by a Principal Validation provider to ensure their authenticity between programmatic server invocations. For more information about Principal Validation providers, see "Principal Validation Concepts" on page 3-12.

   – The LoginModule associates the signed principals with a subject representing the user or system process being authenticated. For more information about subjects and principals, see "Users and Groups, Principals and Subjects" on page 3-3.

   Authentication failure, throws a exception (`LoginException`)

   Note: For more information about LoginModules, see "Java Authentication and Authorization Service (JAAS)" on page 3-4.

Identity Assertion Concepts

Before you develop an Identity Assertion provider, you need to understand the following concepts:

- "Identity Assertion Providers and LoginModules" on page 3-9
- "Identity Assertion and Tokens" on page 3-9
- "Passing Tokens for Perimeter Authentication" on page 3-12

Identity Assertion Providers and LoginModules

When used with a LoginModule, Identity Assertion providers support single sign-on. For example, an Identity Assertion provider can generate a token from a digital certificate and that token can be passed around the system so that users are not asked to sign on more than once.

The LoginModule that an Identity Assertion provider uses can be:

- Part of a custom Authentication provider you develop.
- Part of the Authentication provider BEA developed and supplied with AquaLogic Enterprise Security product.
- Part of a third-party Authentication provider.

Unlike in a simple authentication situation, the LoginModules that Identity Assertion providers use *do not* verify proof material such as usernames and passwords; they simply verify that the user exists.

Note: For more information about LoginModules, see "Writing a JAAS LoginModule" on page 3-5.

Identity Assertion and Tokens

You develop Identity Assertion providers to support the specific types of tokens that you want to use to assert the identities of users or system processes. You can develop an Identity Assertion provider to support multiple token types, but you can configure the Identity Assertion provider so that it validates only one "active" token type. While you can have multiple Identity Assertion providers in a security service module with the ability to validate the same token type, only one Identity Assertion provider can actually perform the validation.

Note: Supporting token types means that the Identity Assertion provider runtime class (that is, the `IdentityAsserter` SSPI implementation) can validate the token type with its `assertIdentity` method. For more information, see "Implementing the AuthenticationProvider SSPI" on page 5-8.

The following sections show how to work with new token types:

- "How to Create New Token Types" on page 3-10
- "How to Make New Token Types Available" on page 3-10

How to Create New Token Types

If you develop a custom Identity Assertion provider, you can also create new token types. A **token type** is simply a piece of data represented as a string. The token types you create and use are completely up to you. As examples, the following token types are currently defined for the X.509 Identity Assertion provider: `X.509`, `CSI.PrincipalName`, `CSI.ITTAnonymous`, `CSI.X509CertChain`, and `CSI.DistinguishedName`.

To create new token types, you create a new Java file and declare any new token types as constant variables of type `String`., as shown in Listing 3-1. The `PerimeterIdentityAsserterTokenTypes.java` file defines the names of the token types `Test 1`, `Test 2`, and `Test 3` as strings.

Listing 3-1 PerimeterIdentityAsserterTokenTypes.java

```
package sample.security.providers.authentication.perimeterATN;

public class PerimeterIdentityAsserterTokenTypes
{
   public final static String TEST1_TYPE = “Test 1”;
   public final static String TEST2_TYPE = “Test 2”;
   public final static String TEST3_TYPE = “Test 3”;
}
```

How to Make New Token Types Available

When you configure a custom Identity Assertion provider, the Supported Types field displays a list of the token types that the Identity Assertion provider supports. To configure the provider, use the Administration Console to select the token type that you want to make active.

The content for the Supported Types field is obtained from the `SupportedTypes` attribute of the MBean Definition File (MDF) that you use to generate your custom Identity Assertion provider MBean type. An example from the sample Identity Assertion provider is shown in Listing 3-2.

Listing 3-2 SampleIdentityAsserter MDF: Supported Types Attribute

```
<MBeanType>
...
   <MBeanAttribute
    Name = "SupportedTypes"
    Type = "java.lang.String[]"
    Writeable = "false"
    Default = "new String[] {&quot;SamplePerimeterAtnToken&quot;}"
   />
...
</MBeanType>
```

Similarly, the content for the Active Types field is obtained from the `ActiveTypes` attribute of the MBean Definition File (MDF). You can specify a default `ActiveTypes` attribute in the MDF so that it does not have to be set manually through the Administration Console. An example from the sample Identity Assertion provider is shown in Listing 3-3.

Listing 3-3 Sample Identity Asserter MDF: Active Types Attribute Default Value

```
<MBeanAttribute
 Name= "ActiveTypes"
 Type= "java.lang.String[]"
 Default = "new String[] { &quot;SamplePerimeterAtnToken&quot; }"
/>
```

While setting a default value for the `ActiveTypes` attribute is convenient, only do this if you do not use another Identity Assertion provider to validate that token type. Otherwise, you may configure an invalid Security Service Module (where more than one Identity Assertion provider attempts to validate the same token type). Best practice dictates that all MDFs for Identity Assertion providers turn off the token type by default; then an administrator can manually make the token type active by configuring the Identity Assertion provider that validates it.

Note: If an Identity Assertion provider is not developed and configured to validate and accept a token type, the authentication process fails.

Passing Tokens for Perimeter Authentication

To perform perimeter authentication, clients can pass tokens using HTTP headers, cookies, SSL certificates, or other mechanisms. For example, a string that is base 64-encoded, which enables the sending of binary data, can be sent to a servlet through an HTTP header. The value of this string can be a username or some other string representation of a user's identity. The Identity Assertion provider used for perimeter authentication can then take that string and extract the username.

For example, when using the WebLogic Server 8.1 Security Service Module, if the token is passed through HTTP headers or cookies, the token is equal to the header or cookie value, and the resource container passes the token to the part of the Security Framework that handles authentication. The Security Framework then passes the token to the Identity Assertion provider, unchanged.

Principal Validation Concepts

Before you develop a Principal Validation provider, you need to understand the following concepts:

- "Principal Validation and Principal Types" on page 3-12
- "How Principal Validation Providers Differ From Other Types of Security Providers" on page 3-13
- "Security Exceptions Resulting from Invalid Principals"

Principal Validation and Principal Types

The Principal Validation provider that is associated with the configured Authentication provider signs and verifies all the principals stored in the subject that are the type that the Principal Validation provider is designed to support. A Principal Validation provider is a special type of security provider that acts primarily as a "helper" to an Authentication provider. The main function of a Principal Validation provider is to prevent malicious individuals from tampering with the principals stored in a subject.

Principal Validation providers support specific types of principals. For example, the WebLogic Principal Validation provider signs and verifies the authenticity of AquaLogic Enterprise Security principals.

How Principal Validation Providers Differ From Other Types of Security Providers

The `AuthenticationProvider` SSPI (as described in "Implementing the AuthenticationProvider SSPI" on page 5-12) includes a method called `getPrincipalValidator`. In this method, you return an instance of the Principal Validation provider runtime class to be used with the Authentication provider. The Principal Validation provider runtime class can be the one BEA provides or one you develop. An example of using the Principal Validation provider in an Authentication provider `getPrincipalValidator` method is shown in Listing 7-1, "SampleAuthenticationProviderImpl.java," on page 7-2.

Because you generate MBean types for Authentication providers and configure Authentication providers using the Administration Application, you do not have to perform these steps for a Principal Validation provider.

Security Exceptions Resulting from Invalid Principals

When the Security Framework attempts an authentication (or authorization) operation, it checks the subject principals to see if they are valid. If a principal is not valid, the Security Framework throws a security exception indicating that the subject is invalid. A subject is invalid because:

- A principal in the subject does not have a corresponding Principal Validation provider configured (which means there is no way for the Security Framework to validate the subject).

  Note: Because you can have multiple principals in a subject, each stored by the LoginModule of a different Authentication provider, the principals can have different Principal Validation providers.

- A principal with an invalid signature was created as part of an attempt to compromise security.

- A subject never had its principals signed.

Authorization Concepts

An **Access Decision** is the component of an Authorization provider that actually answers the question, "*Is access allowed?*" Specifically, an Access Decision asks whether a subject has permission to perform a given operation on a resource, with specific parameters in an application. Given this information, the Access Decision responds with a result of `PERMIT`, `DENY`, or `ABSTAIN`. For more information about Access Decisions, see "Implement the AccessDecision SSPI" on page 5-17.

Role Mapping Concepts

Before you develop a Role Mapping provider, you need to understand the following concepts:

- "Security Roles" on page 3-14
- "Dynamic Security Role Computation" on page 3-15

Security Roles

A **security role** is a named collection of users or groups that have similar permissions to access resources. Like groups, security roles allow you to control access to resources for several users at once. However, unlike groups, security roles can be scoped to resources and actions and are defined dynamically.

The `SecurityRole` interface in the `weblogic.security.service` package is used to represent the abstract notion of a security role. (For more information, see the *Javadocs for Security Service Provider Interfaces* for the `SecurityRole` interface.)

Mapping a principal to a security role grants the associated access permissions to that principal, as long as the principal is "in" the security role. For example, an application may define a security role called `AppAdmin`, which provides write access to a small subset of that application's resources. Any principal in the `AppAdmin` security role would then have write access to those resources.

Many principals can be mapped to a single security role. For more information about principals, see "Users and Groups, Principals and Subjects" on page 3-3. Security roles are specified in the Administration Application. For more information, see the Administration Console online help.

Dynamic Security Role Computation

Dynamic security role computation is the term for the late binding of principals to security roles at runtime. The late binding occurs just prior to an authorization decision for a protected resource, regardless of whether the principal-to-security role association is statically defined or dynamically computed. Because of its placement in the invocation sequence, the result of any principal-to-security role computations can be taken as an authentication identity, as part of the authorization decision made for the request.

This dynamic computation of security roles provides a very important benefit: users or groups can be granted a security role based on business rules. For example, a user may be allowed to be in a `Manager` security role only while the actual manager is away on an extended business trip. Dynamically computing this security role means that you do not need to change or redeploy your application to allow for such a temporarily arrangement. Further, you do not need to remember to revoke the special privileges when the actual manager returns, as you would if you temporarily added the user to a `Managers` group.

Note: You typically grant users or groups security roles using the role conditions available in the Administration Console.

The role mapping provider can access information that comprises the context of the request, including the identity of the target (if available) and the parameter values of the request. The context information is typically used as values of parameters in an expression that is evaluated by the Role Mapping provider. You can define role mapping expressions or rules used by the AquaLogic Enterprise Security Role Mapping provider through the Administration Console.

Auditing Concepts

Before you develop an Auditing provider, you need to understand the following concepts:

- "Audit Channels" on page 3-15
- "Auditing Events from Custom Security Providers" on page 3-16

Audit Channels

An **audit channel** is the component of an Auditing provider that determines whether a security event is audited and performs the actual recording of audit information.

Note: For more information about Audit Channels, see "Implement the AuditChannel SSPI" on page 5-20.

Auditing Events from Custom Security Providers

Each type of security provider can call the configured Auditing providers with a request to write out information about security-related events, before or after these events take place. For example, if a user attempts to access a `withdraw` method in a bank account application (to which they do not have access), the Authorization provider can request that this operation be recorded. Security-related events are only recorded when they meet or exceed the severity level specified in the configuration of the Auditing providers.

For information about how to post audit events from a custom security provider, see Chapter 6, "Auditing Events from Custom Security Providers."

Credential Mapping Concepts

A **subject** or source of a resource request has security-related attributes called **credentials**. A credential may contain information used to authenticate the subject to new services. Such credentials include username and password combinations, Kerberos tickets, and public key certificates. Credentials can also contain data that allows a subject to perform certain activities. Cryptographic keys, for example, represent credentials that enable the subject to sign or encrypt data.

A **credential map** is a mapping of credentials used by AquaLogic Enterprise Security to credentials used in a legacy or any remote system that tells the AquaLogic Enterprise Security system how to connect to a given resource in that system. In other words, credential maps allow AquaLogic Enterprise Security to log in to a remote system on behalf of a subject that has already been authenticated. You can map credentials in this way by developing a Credential Mapping provider.

CHAPTER 4

Design Considerations

Careful planning of development activities can greatly reduce the time and effort you spend developing custom security providers. The following sections provide information to help you make design decisions and to understand the process and components of the custom security provider development:

- General Architecture of a Security Provider
- Security Services Provider Interface
- Developing Security Providers using the SSPI
- Security Services Provider Interface MBeans
- Initialization of the Security Provider Database

General Architecture of a Security Provider

Although you can develop different types of security providers, all security providers follow the same general architecture. Figure 4-1 illustrates the general architecture of a security provider.

Figure 4-1 Security Provider Architecture

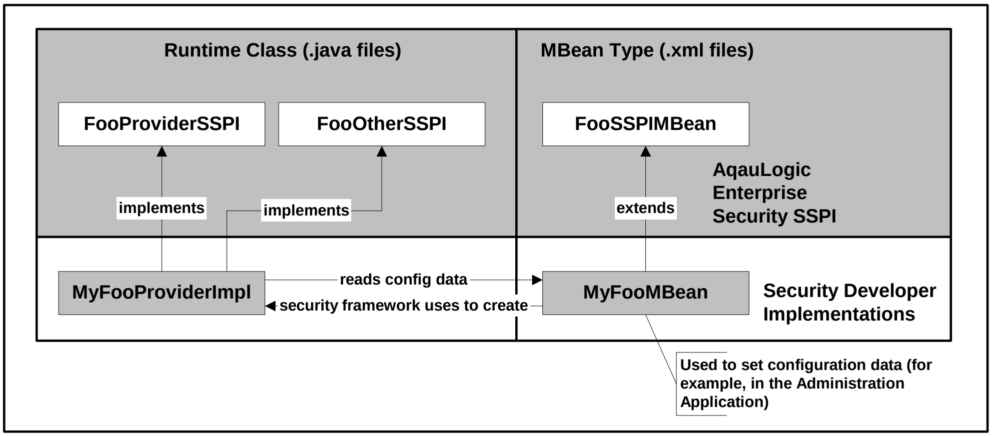

Runtime Class (.java files)
MBean Type (.xml files)
FooProviderSSPI
FooOtherSSPI
FooSSPIMBean
AqauLogic Enterprise Security SSPI
implements
implements
extends
MyFooProviderImpl
reads config data
security framework uses to create
MyFooMBean
Security Developer Implementations
Used to set configuration data (for example, in the Administration Application)

Figure 4-1 shows the relationship between a single runtime class (`MyFooProviderImpl`) and an MBean type (`MyFooMBean`) file. To develop a custom security provider you write an MBean Definition File, the `MyFooMBean` MBean type file. The `MyFooBean` file extends the SSPI MBean (`FooSSPIMBean`). It is written in XML format. When you run the MBean Definition File through the WebLogic MBeanMaker, that utility generates the runtime class for the MBean type, `MyFooMBean.java`.

The process begins when a Security Service Module instance starts and the Security Framework:

1. Locates the MBean type associated with the security provider. MBean types are located in the MJF file for the provider in the /lib/providers directory.
2. Obtains the name of the security provider runtime class from the MBean type (if there are two runtime classes, the one that implements the Provider SSPI), and creates a new instance of the provider runtime class.
3. Passes in the appropriate MBean instance that the security provider uses to initialize (read configuration data) to the initialize() method of the security provider runtime class.

Therefore, both the runtime class or classes and the MBean type form what is called the security provider.

Security Services Provider Interface

You develop a custom security provider by first implementing the security service provider interface to create runtime classes. See "Types of Providers" on page 2-5 for a list of which one to implement for each type of security provider.

Each SSPI that ends in the suffix "`Provider`" (for example, `CredentialProvider`) exposes the services of a security provider to the Security Framework. This allows you to manipulate the security provider (initialize, start, stop, and so on).

Figure 4-2 Provider SSPI

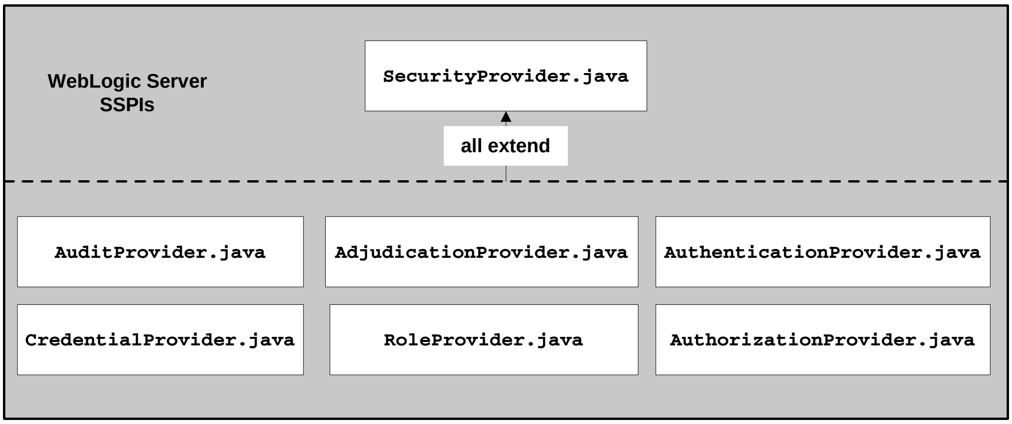


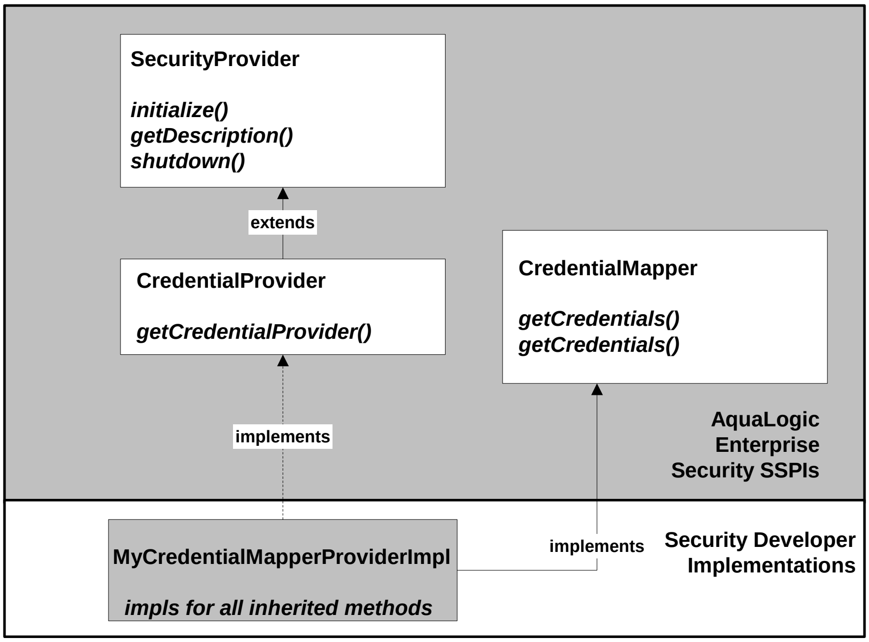
Figure 4-2 shows the SSPI provided with BEA AquaLogic Enterprise Security. This SSPI extends the Security Provider interface (`SecurityProvider.java`) and its methods and exposes the security services to the framework. Because the custom security provider runtime classes implement a Security Provider interface, all such runtime classes must provide implementations for these inherited methods. Table 4-1 describes the security provider interface methods.

Table 4-1 Security Provider Interface Methods

Method	Description
`initialize()`	This method takes two arguments: `providerMBean` and `securityServices`. The `providerMBean` argument can be narrowed to the security provider MBean associated with the security provider. The MBean instance is created from the MBean type you generate and contains configuration data that allows the Administration Application to manage the custom security provider. If this configuration data is available, you can use the `initialize` method to extract it. The `securityServices` argument is an object from which the custom security provider can obtain the Auditor service. For more information about the Auditor Service and auditing, see Creating Auditing Provider Runtime Classes and Auditing Events from Custom Security Providers
`getDescription()`	Returns a brief textual description of the custom security provider.
`shutdown()`	Shuts down the custom security provider.

To develop a custom security provider, you must create runtime classes that implement the security provider SSPI. Using a Credential Mapping provider as an example, Figure 4-3 illustrates the inheritance hierarchy that is common to all SSPIs and shows how a runtime class, can implement the required interfaces. In this example, BEA supplies the `SecurityProvider` interface and the `CredentialProvider` and `CredentialMapper` SSPI. A single runtime class, `MyCredentialMapperProviderImpl`, implements the `CredentialProvider` and `CredentialMapper` SSPI.

Figure 4-3 Credential Mapping SSPI and a Single Runtime Class

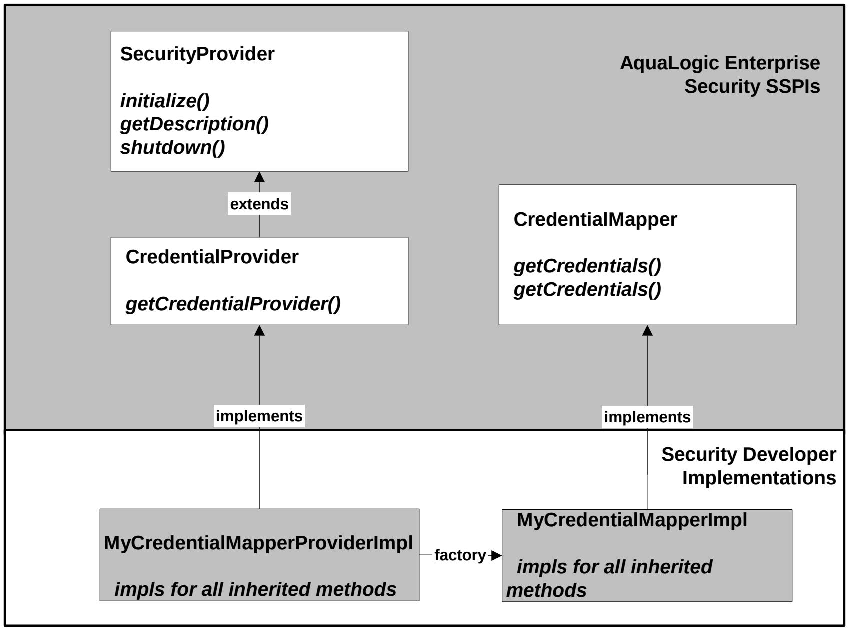


However, Figure 4-3 illustrates only one way you can implement the SSPI, that is, by creating a single runtime class. If you prefer, as illustrated in Figure 4-4, you can create two runtime classes: one for the implementation of the Provider SSPI (for example, the `CredentialProvider`) and one for the implementation of the other SSPI (for example, the `CredentialMapper` SSPI).

When you choose to create two runtime classes, the class that implements the Provider SSPI acts as a factory for generating an instance of the runtime class that implements the other SSPI. For example, in Figure 4-4, `MyCredentialMapperProviderImpl` acts as a factory for generating `MyCredentialMapperImpl`.

Note: If you choose to create two runtime classes, remember to include both of them in the MBean JAR File when you use the WebLogic MBeanMaker to generate the security provider MBean type.

Figure 4-4 Credential Mapping SSPI and Two Runtime Classes



Developing Security Providers using the SSPI

BEA AquaLogic Enterprise Security provides an extended version of the standard WebLogic Security Service Provider Interface (SSPI). Providers that you write to work in both environments must handle both WebLogic resources and extended ones. Listing 4-1 shows how to use the `instanceof` operator in providers to check for extended `resource`.

Listing 4-1 Adding Code to Providers to Check for Extended Resource

```
if ( myresource instanceof com.bea.security.spi.ResourceActionBundle ) {
        // This is a ALES resource that uses the enhanced SSPI.
} else {
        // This is a WLS resource. You must test further for more object
        // types and handle them explicitly.
}
```

Using ResourceActionBundle

The `com.bea.security.spi.ResourceActionBundle` interface is a representation of a resource. As the name implies, this interface is merely a container for two other objects, `ProviderResource` and `ProviderAction`. `ProviderResource` is the resource portion of this object and `ProviderAction` is the action. The security provider understands which object is the operand and which is the verb.

com.bea.security.spi.ProviderResource

A `ProviderResource` object gives the provider a mechanism to parse this resource name without having to understand the intricacies of the specific format. The `ProviderResource.getDeepEnumeration()` method extracts a collection of `NameValueTypes` that can may be mapped into the internal representation of the provider for that resource. The `ProviderResource.getEnumeration()` method is another method that provides a set of ordered `NameValueTypes`, however, this method provides a shallow enumeration, breaking the resource into more coarsely grained pieces. How this resource is parsed is determined by the application developer by means of a naming authority.

A resource type directly relates to its naming authority. A resource that has a naming authority of “`HR_URL`” is considered a different kind of resource than one who has a naming authority of “`INTERNET_URL`,” even if both resources map to the same keys and values. You can get the name of a resource’s naming authority with the `ProviderResource.getAuthorityName()` method.

Additionally, a `ProviderResource` object can also return a reference to its parent resource (`ProviderResource.getResourceParent()` method), if available. Therefore, a provider does not have to know how to produce a parent from the resource; in fact, the resource and the action can have separate parents.

com.bea.security.spi.ProviderAction

A `ProviderAction` object is very similar to a `ProviderResource`. It can be enumerated, parented, and linked to its own naming authority name as well.

Using the ProviderAuditRecord Interface

BEA AquaLogic Enterprise Security provides an extended version of the standard WebLogic Security Service Provider Interface (SSPI). Providers that you write to work in both environments must handle both WebLogic audit records and extended ones. Listing 4-2 shows how to use the `instanceof` operator in providers to check for extended audit records.

Listing 4-2 Adding Code to Providers to Check for an Extended Audit Record

```
if ( myauditrecord instanceof com.bea.security.spi.ProviderAuditRecord) {
        // This is a ALES audit record that uses the enhanced SSPI.
} else {
       // This is a WLS audit record. You must test further for more object
       // types and handle them explicitly.
}
```

A simple audit provider can use the `toString`() method to render the audit record as a string; thus, the provider does not require specific knowledge of the audit record type.

A more complex auditing provider that tracks events by many keys and needs to distinguish messages by various types and attributes, requires a data-driven method of event introspection. The Provider Audit Record Interface, `com.bea.security.spi.ProviderAuditRecord`, satisfies this requirement.

This interface addresses this requirement by employing a similar mechanism as is used to inspect resources. An audit event can be enumerated using the `ProviderAuditRecord.getEnumeration()` and `getDeepEnumeration()` methods.

Additionally, the Provider Audit Record interface can associate an application context with an audit event.

This allows the auditing provider to select some context elements to audit when events occur. For example, when an audit event occurs, you may choose to audit the number of concurrent sessions, the time the user logged on, or some other application specific value propagated by the application context.

Security Services Provider Interface MBeans

The next task in developing a custom security provider is generating an MBean type for the custom security provider.

- Understanding why You Need an MBean Type

- Determining which SSPI MBeans to Extend
- Understanding the Basic Elements of an MBean Definition File
- Understanding the SSPI MBean Hierarchy
- Understanding What the WebLogic MBeanMaker Provides

Understanding why You Need an MBean Type

In addition to creating runtime classes for a custom security provider, you must also generate an MBean type. The term MBean is short for managed bean, a Java object that represents a Java Management eXtensions (JMX) manageable resource. MBeans are used to expose configuration to the provider runtime class.

Note: JMX is a specification created by Sun Microsystems that defines a standard management architecture, APIs, and management services. For more information, see the *Java Management Extensions web page*.

Determining which SSPI MBeans to Extend

You use MBean interfaces called SSPI MBeans to create MBean types. Based on the custom security provider you plan to develop, refer to Table 4-2 and locate the required SSPI MBean.

Table 4-2 Required SSPI MBeans

Type	Package Name	Required SSPI MBean
Authentication provider	`weblogic.management.security.authentication`	`Authenticator`
Identity Assertion provider	`weblogic.management.security.authentication`	`IdentityAsserter`
Authorization provider	`weblogic.management.security.authorization`	`Authorizer`
Adjudication provider	`weblogic.management.security.authorization`	`Adjudicator`
Role Mapping provider	`weblogic.management.security.authorization`	`RoleMapper`

Table 4-2 Required SSPI MBeans

Type	Package Name	Required SSPI MBean
Auditing provider	`weblogic.management.security.audit`	`Auditor`
Credential Mapping provider	`weblogic.management.security.credentials`	`CredentialMapper`

Understanding the Basic Elements of an MBean Definition File

An MBean Definition File is an XML file used by the WebLogic MBeanMaker utility to generate the Java files that comprise an MBean type. All MBean Definition Files *must* extend a required SSPI MBean that is specific to the type of the security provider you have created. Listing 4-3 shows a sample MBean Definition File and an explanation of its content follows.

Note: For a complete reference of MBean Definition File element syntax, see MBean Definition File Element Syntax.

Listing 4-3 SampleCredentialMapper.xml

```
<?xml version="1.0" ?>
<!DOCTYPE MBeanType SYSTEM "commo.dtd">


<!-- MBean Definition File (MDF) for the Sample Credential Mapper.


     Copyright (c) 2003 by BEA Systems, Inc.  All Rights Reserved.
-->


<!-- Declare your mbean.


     Since it is for an credential mapper, it must extend the
     weblogic.management.security.credentials.CredentialMapper or
     weblogic.management.security.credentials.DeployableCredentialMapper
     mbean.

     Since this sample supports WLS RA deployments, it extends the
     weblogic.management.security.credentials.DeployableCredentialMapper
     mbean.

     The Name and DisplayName must be the same.
     They specify the name that will appear on the
     console for this provider.

     Note that since this is an xml document, you can't use double
     quotes directly.  Instead you need to use &quot;

     Note that setting "Writeable" to "false" on an attribute
     makes the attribute read-only.  The default is read-write.
-->

<MBeanType
 Name          = "SampleCredentialMapper"
 DisplayName   = "SampleCredentialMapper"
 Package       = "examples.security.providers.credentials"
 Extends       =
"weblogic.management.security.credentials.DeployableCredentialMapper"
>

 <!-- You must set the value of the ProviderClassName attribute
      (inherited from the weblogic.management.security.Provider mbean)
```

```
          to the name of the java class you wrote that implements the
          weblogic.security.spi.CredentialProvider or
          weblogic.security.spi.DeployableCredentialProvider interface.

          Since this sample supports WLS RA deployments, it implements
          the weblogic.security.spi.DeployableCredentialProvider interface.

          You can think of the provider's mbean as the factory
          for your provider's runtime implementation.
     -->
     <MBeanAttribute
      Name          = "ProviderClassName"
      Type          = "java.lang.String"
      Writeable     = "false"
      Default       =
    "&quot;examples.security.providers.credentials.SampleCredentialMapperProvi
    derImpl&quot;"
     />

     <!-- You must set the value of the Description attribute
          (inherited from the weblogic.management.security.Provider mbean)
          to a brief description of your provider.
          It is displayed in the console.
     -->
     <MBeanAttribute
      Name          = "Description"
      Type          = "java.lang.String"
      Writeable     = "false"
```

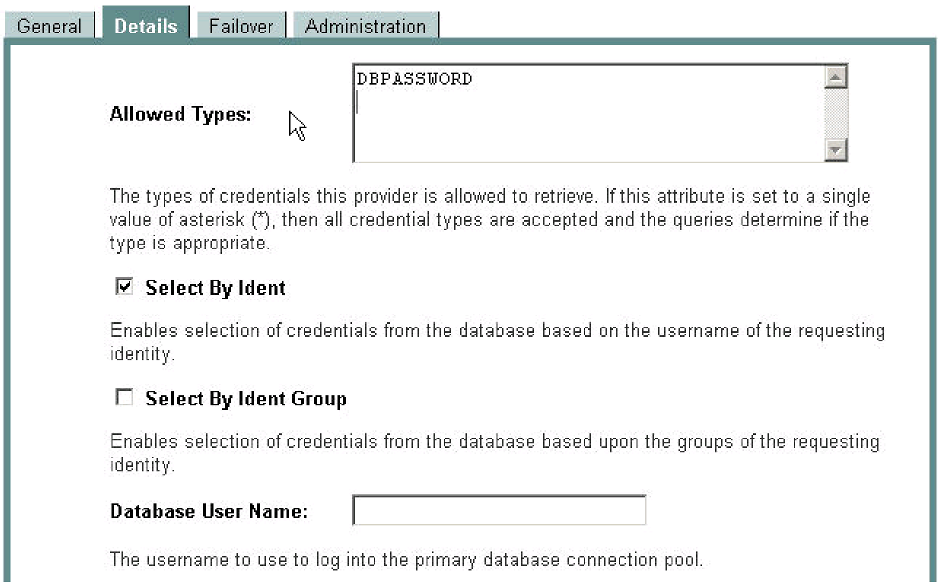
```
  Default        = "&quot;ALES Sample Credential Mapper Provider&quot;"
 />

 <!-- You must set the value of the Version attribute
      (inherited from the weblogic.management.security.Provider mbean)
      to your provider's version.  There is no required format.
 -->
 <MBeanAttribute
  Name           = "Version"
  Type           = "java.lang.String"
  Writeable      = "false"
  Default        = "&quot;1.0&quot;"
 />

 <!-- Add any custom attributes for your provider here.

      The sample credential mapper does not have any custom attributes.
  -->
</MBeanType>
```

The **bold** attributes in the `<MBeanType>` tag show that this MBean Definition File is named `SampleCredentialMapper` and that it extends the required SSPI MBean called `DeployableCredentialMapper`.

The `ProviderClassName`, `Description`, and `Version` attributes defined in the `<MBeanAttribute>` tags are required in any MBean Definition File used to generate MBean types for security providers because they define the basic configuration methods for the provider and are inherited from the base required SSPI MBean called `Provider` (see Figure 4-6). The `ProviderClassName` attribute is especially important. The value for the `ProviderClassName` attribute is the name of the security provider runtime class (that is, the implementation of the appropriate SSPI). The example runtime class shown in Listing 4-3 is `SampleCredentialMapperProviderImpl.java`.

While not shown in Listing 4-3, you can include additional attributes in an MBean Definition File using the `<MBeanAttribute>` tag. Most custom attributes automatically appear in the Details tab for your custom security provider in the Administration Application (as shown in Figure 4-5).

Figure 4-5 Database Credential Mapping Provider Details Tab



Allowed Types: DBPASSWORD

The types of credentials this provider is allowed to retrieve. If this attribute is set to a single value of asterisk (*), then all credential types are accepted and the queries determine if the type is appropriate.

[✓] **Select By Ident**

Enables selection of credentials from the database based on the username of the requesting identity.

[ ] **Select By Ident Group**

Enables selection of credentials from the database based upon the groups of the requesting identity.

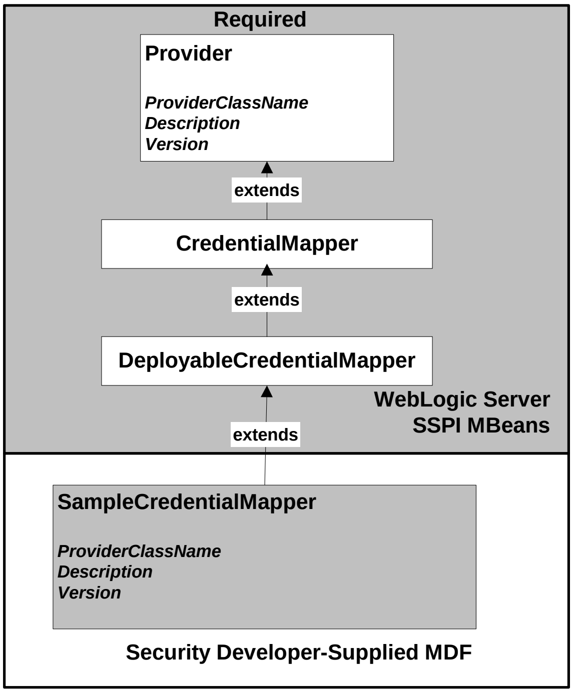
Database User Name:

The username to use to log into the primary database connection pool.

Understanding the SSPI MBean Hierarchy

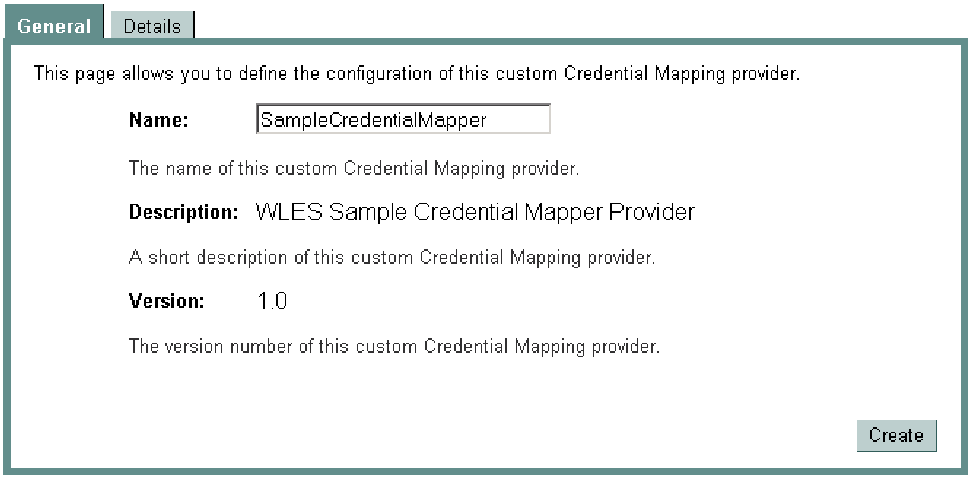
All attributes specified in the required SSPI MBeans that your MBean Definition File extends (all the way up to the `Provider` base SSPI MBean) automatically appear in a Administration Application pages for the associated security provider. You use these attributes to configure your custom security providers. Figure 4-6 illustrates the SSPI MBean hierarchy for security providers using the Sample Credential Mapping MBean Definition File as an example.

Figure 4-6 SSPI MBean Hierarchy for Credential Mapping Providers



Implementing the hierarchy of SSPI MBeans in the Sample Credential Mapper MBean Definition File (shown in Figure 4-6) produces the page in the Administration Application that is shown in Figure 4-7. The full listing of the Sample Credential Mapper MBean Definition File is shown in Listing 4-2.

Figure 4-7 Sample Credential Mapper General Tab



This page allows you to define the configuration of this custom Credential Mapping provider.

Name: SampleCredentialMapper

The name of this custom Credential Mapping provider.

Description: WLES Sample Credential Mapper Provider

A short description of this custom Credential Mapping provider.

Version: 1.0

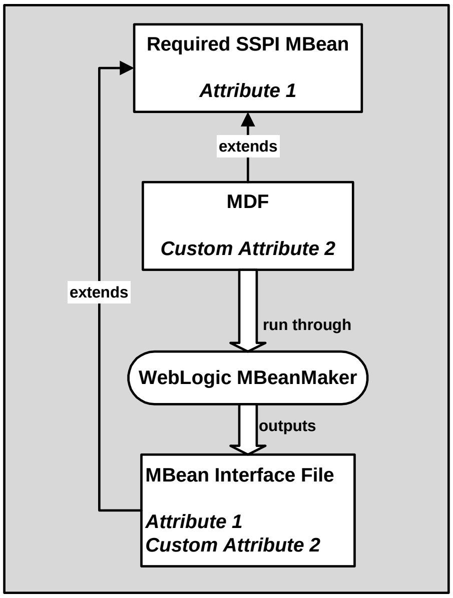
The version number of this custom Credential Mapping provider.

The Name, Description, and Version fields are derived from attributes with the same names, inherited from the base required SSPI MBean called `Provider` and specified in the Sample Credential Mapper MBean Definition File. The `DisplayName` attribute in the Sample Credential Mapper MBean Definition File generates the value for the Name field, and that the `Description` and `Version` attributes generate the values for their respective fields as well.

Understanding What the WebLogic MBeanMaker Provides

The **WebLogic MBeanMaker** is a command-line utility that takes an MBean Definition File as input and outputs an MBean Java interface. This Java interface can then be used in the custom security provider runtime class, through the `weblogic.security.spi.SecurityProvider.initialize()` method, to get configuration attributes. Figure 4-8 shows the operations performed by the WebLogic MBeanMaker utility.

Figure 4-8 What the WebLogic MBeanMaker Provides



Initialization of the Security Provider Database

You must initialize the security provider database with the default users, groups, security policies, security roles, or credentials that your providers need. The provider is not restricted to using a relational database. A provider can store users, groups, etc., in a variety of persistent stores as described in the *Introduction to AquaLogic Enterprise Security*.

- Creating a Simple Database
- Configuring an Existing Database
- Delegating Database Initialization

Creating a Simple Database

The first time you use a custom provider, it attempts to locate a database with the information needed to provide its security service. If the security provider fails to locate the database, it need to create one and populate it with the default users, groups, security policies, security roles, and credentials. This option may be useful for development and testing purposes.

Note: The sample security providers, available under Code Samples:AquaLogic Enterprise Security on the *dev2dev Web site*, simply create and use a properties file as their database. For example, the sample Authentication provider creates a properties file that contains the necessary information about users and groups.

Configuring an Existing Database

If you already have a database (such as an external LDAP server), you can populate that database with the users, groups, security policies, security roles, and credentials that your providers require. Populating an existing database is accomplished using whatever tools you already have in place for performing these tasks.

Once your database contains the necessary information, you must configure the security providers to look in that database. You accomplish this by adding custom attributes in your MBean Definition File. Some examples of custom attributes are the database host, port, password, and so on. You can use the Administration Application to configure these attributes to point to the database.

As an example, Listing 4-4 shows some custom attributes that are part of the LDAP Authentication provider MBean Definition File. These attributes allow an administrator to specify information about the LDAP Authentication provider database (an external LDAP server), so it can locate information about users and groups.

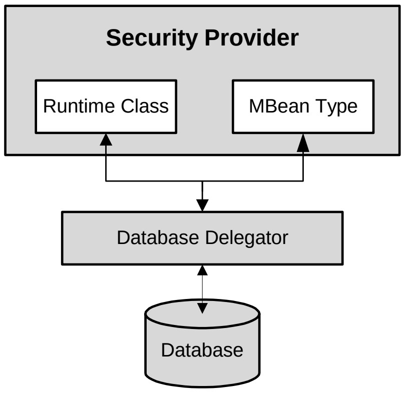
Listing 4-4 LDAPAuthenticator.xml

```
...<MBeanAttribute
 Name = "UserObjectClass"
 Type = "java.lang.String"
 Default = "&quot;person&quot;"
 Description = "The LDAP object class that stores users."
/>

<MBeanAttribute
 Name = "UserNameAttribute"
 Type = "java.lang.String"
 Default = "&quot;uid&quot;"
 Description = "The attribute of an LDAP user object that specifies the name of
   the user."
/>

<MBeanAttribute
 Name = "UserDynamicGroupDNAttribute"
 Type = "java.lang.String"
 Description = "The attribute of an LDAP user object that specifies the
   distinguished names (DNs) of dynamic groups to which this user belongs.
   If such an attribute does not exist, WebLogic Server determines if a
   user is a member of a group by evaluating the URLs on the dynamic group.
   If a group contains other groups, the URLs are evaluated for
   any of the descendents of the group."
/>

<MBeanAttribute
 Name = "UserBaseDN"
 Type = "java.lang.String"
 Default = "&quot;ou=people, o=example.com&quot;"
 Description = "The base distinguished name (DN) of the tree in the LDAP
   directory that contains users."
/>
<MBeanAttribute
 Name = "UserSearchScope"
 Type = "java.lang.String"
 Default = "&quot;subtree&quot;"
 LegalValues = "subtree,onelevel"
 Description = "Specifies how deep in the LDAP directory tree to search
   for Users.
   Valid values are &lt;code&gt;subtree&lt;/code&gt;
   and &lt;code&gt;onelevel&lt;/code&gt;."
/>

...
```

Delegating Database Initialization

If possible, initialization calls between a security provider and the security provider database are done by an intermediary class, referred to as a **database delegator**. Use of a database delegator is convenient because it hides the database and centralizes calls into the database. The database delegator should interact with the runtime class and the MBean type for the security provider, as shown in Figure 4-9.

Figure 4-9 Database Delegator Class Positioning



Developing Custom Security Providers

If the security providers that ship with the AquaLogic Enterprise Security product do not meet your needs, you can develop custom security providers by following the steps outlined in "Overview of the Development Process" on page 2-1.

This section covers the following topics:

Types of Custom Security Providers Supported

This section described the most common security provider types.

You can create and use any of the security provider types described in the WebLogic Server-specific version of Developing WebLogic Security Providers, but you must then manage these providers from the WebLogic Administration Console, as described in "Security Provider Management Concepts" on page 3-2.

The types of custom security providers you can develop include the following:

- **Authentication Provider**

  An Authentication provider is used to prove the identity of users or system processes. Authentication providers also remember, transport, and make that identity information available to various components of a system through subjects when needed. During the authentication process, a Principal Validation provider provides additional security protections for the principals (users and groups) contained within the subject by signing and verifying the authenticity of those principals.

- **Identity Assertion Provider**

  An Identity Assertion provider is a specific form of Authentication provider that allows users or system processes to assert their identity using tokens (in other words, perimeter authentication). You can use an Identity Assertion provider in place of an Authentication provider if you create a LoginModule for the Identity Assertion provider, or in addition to an Authentication provider if you want to use the Authentication provider LoginModule. Identity Assertion providers enable perimeter authentication and support single sign-on.

- **Principal Validation Provider**

  Authentication providers rely on Principal Validation providers to sign and verify the authenticity of principals (users and groups) contained within a subject. Such verification provides an additional level of trust and may reduce the likelihood of malicious principal tampering. The authenticity of the principal is verified when making authorization decisions.

- **Role Mapping Provider**

  Role mapping is the process whereby principals (users or groups) are dynamically mapped to security roles at runtime. A Role Mapping provider determines which security roles apply to the principals stored a subject when the subject is attempting to perform an operation on a resource. Because this operation usually involves gaining access to the resource, Role Mapping providers are typically used with Authorization providers.

- **Authorization Provider**

  Authorization is the process whereby the interactions between users and resources are controlled, based on user identity or other information. In other words, authorization answers the question, *What can you access?* An Authorization provider is used to limit the interactions between users and resources to ensure integrity, confidentiality, and availability.

- **Adjudication Provider**

Adjudication involves resolving any authorization conflicts that may occur when more than one Authorization provider is configured, by weighing the result of each Access Decision. An Adjudication provider tallies the results that multiple Access Decisions return, and determines the final `PERMIT` or `DENY` decision. An Adjudication provider may also specify what should be done when an answer of `ABSTAIN` is returned from a single Authentication provider.

- **Auditing Provider**

  An Auditing Provider processes information about operating requests and the outcome of those requests are collected, stored, and distributed for the purposes of non-repudiation. An Auditing provider provides this electronic trail of computer activity.

- **Credential Mapping Provider**

  A Credential Mapping Provider uses a legacy system database to obtain an appropriate set of credentials to use to authenticate users to a target resource. A Credential Mapping provider employs credential mapping services and bring new types of credentials into the environment.

Writing an MBean Definition File

The MDF for the sample Authentication provider is called `SampleAuthenticator.xml`.

To create an MBean Definition File (MDF), follow these steps:

1. Copy the MDF for the sample Authentication provider to a text file.

Listing 5-1 SampleAuthenticator.xml MDF File

```
<?xml version="1.0" ?>
<!DOCTYPE MBeanType SYSTEM "commo.dtd">


<!-- MBean Definition File (MDF) for the Sample Authenticator.


     Copyright (c) 2003 by BEA Systems, Inc. All Rights Reserved.
-->


<!-- Declare your mbean.
```

```
     Since it is for an authenticator, it must extend the
     weblogic.management.security.authentication.Authenticator mbean.

     The Name and DisplayName cannot be the same.
     They specify the name to appear on the
     console for this provider.

     Because this is an xml document, you can't use double
     quotes directly. Instead you need to use &quot;

     Note that setting "Writeable" to "false" on an attribute
     makes the attribute read-only. The default is read-write.
-->
<MBeanType
 Name          = "SampleAuthenticator"
 DisplayName   = "SampleAuthenticator"
 Package       = "examples.security.providers.authentication"
 Extends       =
"weblogic.management.security.authentication.Authenticator"
>

 <!-- You must set the value of the ProviderClassName attribute
      (inherited from the weblogic.management.security.Provider mbean)
      to the name of the java class you wrote that implements the
      weblogic.security.spi.AuthenticationProvider interface.

      You can think of the provider's mbean as the factory
      for your provider's runtime implementation.
 -->
 <MBeanAttribute
```

```
  Name          = "ProviderClassName"
  Type          = "java.lang.String"
  Writeable     = "false"
  Default       =
"&quot;examples.security.providers.authentication.SampleAuthenticationP
roviderImpl&quot;"
 />

 <!-- You must set the value of the Description attribute
      (inherited from the weblogic.management.security.Provider mbean)
      to a brief description of your provider.
      It is displayed in the console.
 -->
 <MBeanAttribute
  Name          = "Description"
  Type          = "java.lang.String"
  Writeable     = "false"
  Default       = "&quot;ALES Sample Authentication Provider&quot;"
 />

 <!-- You must set the value of the Version attribute
      (inherited from the weblogic.management.security.Provider mbean)
      to your version of the provider. There is no required format.
 -->
 <MBeanAttribute
  Name          = "Version"
  Type          = "java.lang.String"
  Writeable     = "false"
  Default       = "&quot;1.0&quot;"
 />
 <!-- Add any custom attributes for your provider here.

      The sample authenticator does not have any custom attributes.

-->

</MBeanType>
```

2. Modify the content of the `<MBeanType>` and `<MBeanAttribute>` elements in your MDF so that they are appropriate for the type of custom security provider you are developing.
3. Add any custom attributes. (that is, additional `<MBeanAttribute>` elements) to your MDF.
4. Save the file.

Note: A complete reference of MDF element syntax is available in "MBean Definition File Element Syntax" on page A-1.

Using the WebLogic MBeanMaker to Generate the MBean Type

After you create your MDF, you are ready to run it through the WebLogic MBeanMaker. The WebLogic MBeanMaker is a command-line utility that takes an MDF and outputs some intermediate Java files, including an MBean interface, an MBean implementation, and an associated MBean information file. Together, these intermediate files form the **MBean type** for your custom security provider.

To generate the MBean type, follow these steps:

1. Create a new DOS shell.
2. Set the AquaLogic Enterprise Security environment variable by calling `ALES_HOME/bin/set-env.bat.`
3. Type the following command:

   `java -DMDF=`*xmlfile* `-Dfiles=`*filesdir* `-DcreateStubs=true weblogic.management.commo.WebLogicMBeanMaker`

   where:

   `-DMDF` is a flag that instructs the WebLogic MBeanMaker to translate the MDF into code.

`xmlFile` is the MDF (the XML MBean Description File).

`filesdir` is the location where the WebLogic MBeanMaker places the intermediate files for the MBean type.

Whenever *`xmlfile`* is provided, a new set of output files is generated. If files already exist in the location specified by *`filesdir`*, the existing files are overwritten.

Each time you use the `-DcreateStubs=true` flag, the MBeanMaker overwrites any existing MBean implementation file.

Note: As of version 9.0 of WebLogic Server, you can also provide a directory that contains multiple MDF's by using the -DMDFDIR <MDF directory name> option. In prior versions of WebLogic Server, the WebLogic MBeanMaker processed only one MDF at a time. Therefore, you had to repeat this process if you had multiple MDFs (in other words, multiple providers).

4. Proceed to "Creating an MBean JAR File" on page 5-22.

About the Generated MBean Interface File

The MBean interface file is the client-side API to the MBean that your runtime class uses to obtain configuration data. The initialize method uses the MBean interface file. Because the WebLogic MBeanMaker generates MBean types from the MDF you created, the generated MBean interface file has the same name as the MDF, appended with MBean. For example, the result of running the `SampleAuthenticator` MDF through the WebLogic MBeanMaker yields an MBean interface file called `SampleAuthenticatorMBean.java`.

Creating Security Provider Runtime Classes

This section describes how to create runtime classes for each type of provider. For more information about the SSPI and the methods described, see the *Javadocs for Security Service Provider Interfaces*.

Creating Authentication Provider Runtime Classes

To create the runtime classes for your custom Authentication provider, perform the following tasks:

- Implementing the AuthenticationProvider SSPI
- Implementing the JAAS LoginModule Interface

- Implementing Custom Exceptions for LoginModules

For an example of how to create a runtime class for a custom Authentication provider, see "Example: Creating the Runtime Classes for the Sample Authentication Provider" on page 7-1.

Implementing the AuthenticationProvider SSPI

To implement the AuthenticationProvider SSPI, provide implementations for the methods described in Table 4-1 and the `weblogic.security.spi.AuthenticationProvider` interface methods, described in Table 5-1.

Table 5-1 AuthenticationProvider Interface Methods

Method	Description
`getLoginModuleConfiguration()`	The `getLoginModuleConfiguration` method obtains information about the LoginModule associated with the Authentication provider, which is returned as an `AppConfigurationEntry`. The `AppConfigurationEntry` is a Java Authentication and Authorization Service (JAAS) class that contains the classname of the LoginModule; the LoginModule control flag (passed in through the MBean associated with the Authentication provider); and a configuration options map for the LoginModule (which allows other configuration information to be passed into the LoginModule). For more information about the `AppConfigurationEntry` class (located in the `javax.security.auth.login` package) and the control flag options for LoginModules, see the Java API Specification Javadoc AppConfigurationEntry class and the Configuration class. For more information about LoginModules, see "Writing a JAAS LoginModule" on page 3-5.
`getAssertionModuleConfiguration()`	The `getAssertionModuleConfiguration` method obtains information about the LoginModule associated with the Identity Assertion provider, which is returned as an `AppConfigurationEntry`. The `AppConfigurationEntry` is a JAAS class that contains the classname of the LoginModule; the LoginModule control flag (passed in through the MBean associated with the Authentication provider); and a configuration options map for the LoginModule (which allows other configuration information to be passed into the LoginModule). The implementation of the `getAssertionModuleConfiguration` method can be to return `null`, if you want the Identity Assertion provider to use the same LoginModule as the Authentication provider.

Table 5-1 AuthenticationProvider Interface Methods (Continued)

Method	Description
`getPrincipalValidator()`	The `getPrincipalValidator` method obtains a reference to the Principal Validation provider runtime class (that is, the Principal Validator SSPI implementation). In most cases, the Principal Validation provider supplied with the product can be used (see "SampleAuthenticationProviderImpl.java" on page 7-2 for an example of how to return the Principal Validation provider). For more information about Principal Validation providers, see "Creating Identity Assertion Runtime Classes" on page 5-12.
`getIdentityAsserter()`	The `getIdentityAsserter` method obtains a reference to the Identity Assertion provider runtime class (that is, the Identity Asserter SSPI implementation). In most cases, the return value for this method is `null` (see Listing 7-1 for an example). For more information about Identity Assertion providers, see "Creating Identity Assertion Runtime Classes" on page 5-12.

Implementing the JAAS LoginModule Interface

To implement the JAAS `javax.security.auth.spi.LoginModule` interface, provide implementations for the method described in Table 5-2.

Table 5-2 LoginInterface Methods

Method	Description
`initialize()`	The `initialize` method initializes the LoginModule. It takes as arguments a subject in which to store the resulting principals, a CallbackHandler that the Authentication provider uses to call back to the container for authentication information, a map of any shared state information, and a map of configuration options (that is, any additional information you want to pass to the LoginModule). A CallbackHandler is a highly-flexible JAAS standard that allows a variable number of arguments to be passed as complex objects to a method. For more information about CallbackHandlers, see the *Java 2 Enterprise Edition, v1.4.2 API Specification Javadoc for the CallbackHandler interface*.
`login()`	The `login` method attempts to authenticate the user and create principals for the user by calling back to the container for authentication information. If multiple LoginModules are configured (as part of multiple Authentication providers), this method is called for each LoginModule in the order that they are configured. Information about whether the login was successful (that is, whether principals were created) is stored for each LoginModule.
`commit()`	The `commit` method attempts to add the principals created in the `login` method to the subject. This method is also called for each configured LoginModule (as part of the configured Authentication providers), and executed in order. Information about whether the commit was successful is stored for each LoginModule.
`abort()`	The `abort` method is called for each configured LoginModule (as part of the configured Authentication providers), if any commits for the LoginModules failed; in other words, the relevant `REQUIRED`, `REQUISITE`, `SUFFICIENT` and `OPTIONAL` LoginModules did not succeed. The `abort` method removes those principals from the subject, effectively rolling back the actions performed.
`logout()`	The `logout` method attempts to log the user out of the system. It also resets the subject so that its associated principals are no longer stored.

For more information about the JAAS `LoginModule` interface and the methods described above, see the *Java Authentication and Authorization Service (JAAS) 1.0 Developer's Guide*, and the *Java 2 Enterprise Edition, v1.4.2 API Specification Javadoc* for the LoginModule interface.

Implementing Custom Exceptions for LoginModules

Optionally, you may want LoginModule that you write to throw a custom exception. The custom exception can be caught by your application and the appropriate action taken. For example, if the LoginModule throws a `PasswordChangeRequiredException`, you can catch that exception within your application, and use it to forward users to a page that allows them to change their password.

You must make your custom exception available to both the Authentication provider (at build, compile, and runtime) and to your application at compile time. You can do this using either of the following two methods.

Method 1: Make Custom Exceptions Available through the System Classpath

1. Write an exception class that extends `LoginException`.
2. Use the custom exception class in your classes that implement the LoginModule and Authentication Provider interfaces.
3. Put the custom exception class in the system classpath and the compiler path when compiling the security provider runtime class.
4. See "Using the WebLogic MBeanMaker to Generate the MBean Type" on page 5-6 for additional instructions.

Method 2: Make Custom Exceptions Available through the System Classpath and the Authentication Provider

1. Write an exception class that extends `LoginException`.
2. Use the custom exception class in your classes that implement the LoginModule and Authentication Provider interfaces.
3. Put the custom exception class in the compiler path when compiling the security provider runtime class.
4. See "Using the WebLogic MBeanMaker to Generate the MBean Type" on page 5-6 for additional instructions.
5. Add the custom exception class to the MJF (MBean JAR File) generated by the WebLogic MBeanMaker.
6. Include the MJF in the compiler and system classpath when compiling and running your application.

Creating Identity Assertion Runtime Classes

After creating any custom exceptions, you must create the runtime classes for your custom Identity Assertion provider. If you want to create a separate LoginModule for your custom Identity Assertion provider (that is, not use the LoginModule from your Authentication provider), you need to implement the JAAS LoginModule interface, as described in "Implementing the JAAS LoginModule Interface" on page 5-9.

For an example of how to create a runtime classes for a custom Identity Assertion provider, see "Example: Creating the Runtime Class for the Sample Identity Assertion Provider" on page 7-9.

Implementing the AuthenticationProvider SSPI

To implement the AuthenticationProvider SSPI, provide implementations for the Security Provider interface methods described in Table 4-1 and the `weblogic.security.spi.AuthenticationProvider` interface methods described in Table 5-1.

Note: When the LoginModule used for the Identity Assertion provider is the same as that used for an existing Authentication provider, implementations for the methods in the Authentication Provider SSPI (excluding the `getIdentityAsserter` method) for Identity Assertion providers can just return `null`. An example of this is shown in Listing 7-3.

Implementing the IdentityAsserter SSPI

To implement the IdentityAsserter SSPI, provide an implementation of the `weblogic.security.spi.IdentityAsserter.assertIdentity()` method, described in Table 5-3.

Note: The `ActiveTypes` MBeanAttribute contains the subset of your MBean's `SupportedTypes` that are active in the security realm. If you define the `ActiveTypes` MBeanAttribute, you must also define `SupportedTypes`.

Table 5-3 IdentityAsserter SSPI Method

Method	Description
`assertIdentity()`	The `assertIdentity` method asserts an identity based on the token identity information that is supplied. In other words, the purpose of this method is to validate any tokens that are not currently trusted against trusted client principals. The `type` parameter represents the token type to be used for the identity assertion. Note that identity assertion types are case insensitive. The `token` parameter contains the actual identity information. The `CallbackHandler` returned from the `assertIdentity` method is passed to all configured Authentication provider LoginModules to perform principal mapping, and should contain the asserted username. If the `CallbackHandler` is `null`, this signifies that the anonymous user should be used. A CallbackHandler is a highly-flexible JAAS standard that allows a variable number of arguments to be passed as complex objects to a method. For more information about CallbackHandlers, see the *Java 2 Enterprise Edition, v1.4.2 API Specification Javadoc* for the CallbackHandler interface.

Creating Principal Validation Provider Runtime Classes

To develop a custom Principal Validation provider:

- Write your own `UserImpl` and `GroupImpl` classes by:
  - Implementing the `weblogic.security.spi.WLSUser` and `weblogic.security.spi.WLSGroup` interfaces.
  - Implementing the `java.io.Serializable` interfaces.
- Write your own `PrincipalValidationImpl` class by implementing the `weblogic.security.spi.PrincipalValidator` SSPI. For instructions, see "Implementing the PrincipalValidator SSPI".

Implementing the PrincipalValidator SSPI

To implement the PrincipalValidator SSPI, provide implementations of the Principal Validator methods described in Table 5-4.

Table 5-4 PrincipalValidator SSPI Methods

Method	Description
`validate`	The `validate` method takes a principal as an argument and attempts to validate it. In other words, this method verifies that the principal was not altered since it was signed.
`sign`	The `sign` method takes a principal as an argument and signs it to assure trust. This allows the principal to later be verified using the `validate` method. Your implementation of the `sign` method should be a secret algorithm that malicious individuals cannot easily recreate. You can include that algorithm within the `sign` method itself, have the `sign` method call out to a server for a token it should use to sign the principal, or implement some other way of signing the principal.
`getPrincipalBaseClass`	The `getPrincipalBaseClass` method returns the base class of principals that this Principal Validation provider knows how to validate and sign.

Creating Role Mapping Provider Runtime Classes

To create the runtime classes for your custom Role Mapping provider, perform the following tasks:

- "Implement the RoleProvider SSPI" on page 5-14
- "Implement the RoleMapper SSPI" on page 5-15
- "Implement the SecurityRole Interface" on page 5-15

For an example of how to create a runtime class for a custom Role Mapping provider, see "Example: Creating the Runtime Class for the Sample Role Mapping Provider" on page 7-15.

Implement the RoleProvider SSPI

To implement the RoleProvider SSPI, provide implementations for the methods described in Table 4-1 and the `weblogic.security.spi.RoleProvider.getRoleMapper` method described in Table 5-5.

Table 5-5 RoleProvider SSPI Method

Method	Description
`getRoleMapper`	The `getRoleMapper` method obtains the implementation of the Role Mapper SSPI. For a single runtime class called `MyRoleProviderImpl.java`, the implementation of the `getRoleMapper` method is: `return this;` If there are two runtime classes, then the implementation of the `getRoleMapper` method is: `return new MyRoleMapperImpl();` This is because the runtime class that implements the Role Provider SSPI is used as a factory to obtain classes that implement the Role Mapper SSPI.

Implement the RoleMapper SSPI

To implement the RoleMapper SSPI, provide implementations for the `weblogic.security.spi.RoleMapper.getRoles` method described in Table 5-6.

Table 5-6 RoleMapper SSPI Method

Method	Description
`getRoles`	The `getRoles` method returns the security roles associated with a given subject for a specified resource, possibly using the optional information specified in the `ContextHandler`. For more information about Context Handlers, see "ContextHandler Object" on page 6-8.

Implement the SecurityRole Interface

The methods on the SecurityRole interface allow you to obtain basic information about a security role or to compare it to another security role. These methods are designed for the convenience of security providers.

Note: Security Role implementations are returned as a `Map` by the `getRoles()` method, keyed by role name.

To implement the Security Role interface, provide implementations for the `weblogic.security.service.SecurityRole` interface methods described in Table 5-7.

Table 5-7 SecurityRole Interface Methods

Method	Description
`equals`	The `equals` method returns `TRUE` if the security role passed in matches the security role represented by the implementation of this interface; otherwise it returns `FALSE`.
`toString`	The `toString` method returns the security role, represented as a String.
`hashCode`	The `hashCode` method returns a hashcode for the security role, represented as an integer.
`getName`	The `getName` method returns the name of the security role, represented as a String.
`getDescription`	The `getDescription` method returns a description of the security role, represented as a String. The description should describe the purpose of this security role.

Creating AuthorizationProvider Runtime Classes

To create the runtime classes for your custom Authorization provider, perform the following tasks:

- "Implement the AuthorizationProvider SSPI" on page 5-16
- "Implement the AccessDecision SSPI" on page 5-17

For an example of how to create a runtime class for a custom Authorization provider, see "Example: Creating the Runtime Class for the Sample Authorization Provider" on page 7-12.

Implement the AuthorizationProvider SSPI

To implement the AuthorizationProvider SSPI, provide implementations for the methods described in Table 4-1 and the method described in Table 5-8.

Table 5-8 AuthorizationProvider SSPI Method

Method	Description

`getAccessDecision`	The `getAccessDecision` method obtains the implementation of the Access Decision SSPI. For a single runtime class called `MyAuthorizationProviderImpl.java`, the implementation of the `getAccessDecision` method is:
	`return this;`
	If there are two runtime classes, then the implementation of the `getAccessDecision` method is
	`return new MyAccessDecisionImpl();`
	This is because the runtime class that implements the Authorization Provider SSPI is used as a factory to obtain classes that implement the Access Decision SSPI.

Implement the AccessDecision SSPI

When you implement the AccessDecision SSPI, you must provide implementations for the methods described in Table 5-9.

Table 5-9 AccessDecision SSPI Methods

Method	Description
`isAccessAllowed`	The `isAccessAllowed` method uses information contained within the subject to determine if the requestor is allowed to access a protected resource. The `isAccessAllowed` method can be called prior to or after a request, and returns values of `PERMIT`, `DENY`, or `ABSTAIN`. If multiple Access Decisions are configured and return conflicting values, an Adjudication provider is needed to determine a final result. For more information, see "Creating AdjudicationProvider Runtime Classes" on page 5-18.
`isProtectedResource`	The `isProtectedResource` method determines if the specified resource is protected, without incurring the cost of an actual access check. It is only a lightweight mechanism because it does not compute a set of security roles that can be granted to the subject.

Creating AdjudicationProvider Runtime Classes

To create the runtime classes for your custom Adjudication provider, perform the following tasks:

- Implement the AdjudicationProvider SSPI
- Implement the Adjudicator SSPI

Implement the AdjudicationProvider SSPI

To implement the AdjudicationProvider SSPI, provide implementations for the methods described in Table 4-1 and the `weblogic.security.spi.AdjudicationProvider.getAdjudicator` method described in Table 5-10.

Table 5-10 AdjudicationProvider SSPI Method

Method	Description

`getAdjudicator`	The `getAdjudicator` method obtains the implementation of the Adjudicator SSPI. For a single runtime class called `MyAdjudicationProviderImpl.java`, the implementation of the `getAdjudicator` method is:
	`return this;`
	If there are two runtime classes, then the implementation of the `getAdjudicator` method is:
	`return new MyAdjudicatorImpl();`
	This is because the runtime class that implements the Adjudication Provider SSPI is used as a factory to obtain classes that implement the `Adjudicator` SSPI.

Implement the Adjudicator SSPI

To implement the Adjudicator SSPI, provide implementations for the methods described in Table 5-11.

Table 5-11 Adjudicator SSPI Method

Method	Description
`initialize`	The `initialize` method initializes the names of all the configured Authorization provider Access Decisions called to supply a result for the "Is access allowed?" The `accessDecisionClassNames` parameter can be used by an Adjudication provider in its `adjudicate` method to favor a result from a particular Access Decision.
`adjudicate`	The `adjudicate` method determines the answer to the "Is access allowed?" given all the results from the configured Authorization provider Access Decisions.

Creating Auditing Provider Runtime Classes

To create the runtime classes for your custom Auditing provider, perform the following tasks:

- Implement the AuditProvider SSPI
- Implement the AuditChannel SSPI

For an example of how to create a runtime class for a custom Auditing provider, see "Example: Creating the Runtime Class for the Sample Auditing Provider" on page 7-19.

Implement the AuditProvider SSPI

To implement the AuditProvider SSPI, provide implementations for the methods described in Table 4-1 and the `weblogic.security.spi.AuditProvider.getAuditChannel` method described in Table 5-12.

Table 5-12 AuditProvider SSPI Method

Method	Description
`getAuditChannel`	The `getAuditChannel` method obtains the implementation of the Audit Channel SSPI. For a single runtime class called `MyAuditProviderImpl.java`, the implementation of the `getAuditChannel` method is: `return this;` If there are two runtime classes, then the implementation of the `getAuditChannel` method is `return new MyAuditChannelImpl();` This is because the runtime class that implements the Audit Provider SSPI is used as a factory to obtain classes that implement the Audit Channel SSPI.

Implement the AuditChannel SSPI

To implement the AuditChannel SSPI, provide an implementation for the `weblogic.security.spi.AuditChannel.writeEvent` method described in Table 5-13.

Table 5-13 AuditChannel SSPI Method

Method	Description
`writeEvent`	The `writeEvent` method writes an audit record based on the information specified in the `AuditEvent` object that is passed in. For more information about `AuditEvent` objects, see "Creating an Audit Event" on page 6-3.

Creating Credential Mapping Provider Runtime Classes

To create the runtime classes for your custom Credential Mapping provider, perform the following tasks:

- Implement the CredentialProvider SSPI
- Implement the Credential Mapper SSPI

Implement the CredentialProvider SSPI

To implement the CredentialProvider SSPI, provide implementations for the methods described in Table 4-1 and the `weblogic.security.spi.CredentialProvider.getCredentialProvider` method described in Table 5-14.

Table 5-14 CredentialProvider SSPI Method

Method	Description

`getCredentialProvider`	The `getCredentialProvider` method obtains the implementation of the Credential Mapper SSPI. For a single runtime class called `MyCredentialMapperProviderImpl.java` (as in Figure 4-3), the implementation of the `getCredentialProvider` method is:
	`return this;`
	If there are two runtime classes, then the implementation of the `getCredentialProvider` method is:
	`return new MyCredentialMapperImpl();`
	This is because the runtime class that implements the Credential Provider SSPI is used as a factory to obtain classes that implement the Credential Mapper SSPI.

Implement the Credential Mapper SSPI

To implement the Credential Mapper SSPI, you must provide implementations for the `weblogic.security.spi.CredentialMapper` methods described in Table 5-15.

Table 5-15 Credential Mapper SSPI Methods

Method	Description
`getCredentials()`	**Format:** `public java.util.Vector getCredentials(Subject requestor, Subject initiator, Resource resource, String[] credentialTypes);` The `getCredentials` method obtains the appropriate set of credentials for the target resource, based on the identity of the initiator. This version of the method returns a list of matching credentials for all of the principals within the subject (as a vector) by consulting the database for the remote system.
`getCredentials()`	**Format:** `public java.lang.Object getCredentials(Subject requestor, String initiator, Resource resource, String[] credentialTypes);` The `getCredentials` method obtains the appropriate set of credentials for the target resource, based on the identity of the initiator. This version of the method returns one credential for the specified subject (as an object) by consulting the database for the remote system.

Creating an MBean JAR File

After you run your MDF through the WebLogic MBeanMaker to generate your custom MBean files, you need to package the MBean files and the runtime classes for the custom security provider into an MBean JAR file. The WebLogic MBeanMaker automates this process.

To create an MJF for your custom security provider, follow these steps:

1. Create a new DOS shell.
2. Set the AquaLogic Enterprise Security environment variable by calling `ALES_HOME/bin/set-env.bat.`
3. Type the following command:

   `java -DMJF=`*jarfile* `-DFiles=`*filesdir*
   `weblogic.management.commo.WebLogicMBeanMaker`

   where:

   `-DMJF` is a flag instructing the WebLogic MBeanMaker to build an MBean JAR file containing the new provider.

jarfile is the name for the MBean JAR file.

filesdir is the location where the WebLogic MBeanMaker looks for the files to JAR into the MBean JAR file.

Compilation occurs at this point, so errors are possible. If *jarfile* is provided and no errors occur, an MBean JAR file is created with the specified name.

Notes: If you want to update an existing MBean JAR file, simply delete the MBean JAR file and regenerate it. The WebLogic MBeanMaker also has a `-DIncludeSource` option that controls whether to include source files in the resulting MBean JAR file. Source files include both the generated source and the MBean definition file itself. The default is `false`. This option is ignored when `-DMJF` is not used.

The resulting MBean JAR file can be deployed into your AquaLogic Enterprise Security environment or distributed for installation into other AquaLogic Enterprise Security environments.

Deploying a Security Provider MJF File

As described in "Security Provider Management Concepts" on page 3-2, before you create a custom security provider, you need to understand how that provider will be managed by BEA AquaLogic Enterprise Security.

If you used the version 8.1 WebLogic MBeanMaker to create the custom security provider, copy the MJF file into the following directory:

```
PRODUCT_HOME\lib\providers
```

where:

PRODUCT_HOME is the top-level installation directory for BEA AquaLogic Enterprise Security product.

Note: You must copy the file to both the machine on which the Security Service Module is installed and to the Administration Server. You must copy the file to any and all instances of Security Service Modules that use the new provider.

This deploys your custom security provider—that is, you can configure the custom security provider from the Administration Application and us it with your Security Service Module instance.

If you used the version 8.1 WebLogic MBeanMaker to create the custom security provider, copy the MJF into the *WL_HOME*`\server\lib\mbeantypes` directory, where *WL_HOME* is the top-level installation directory for WebLogic Server. This makes the custom provider manageable from the WebLogic Server Administration Console and WLST.

`*WL_HOME*\server\lib\mbeantypes` is the default directory for installing MBean types. Beginning with 9.0, security providers can be loaded from `...\domaindir\lib\mbeantypes` as well. If you want WebLogic Server to look for MBean types in additional directories, use the `-Dweblogic.alternateTypesDirectory=<dir>` command-line flag when starting your server, where `<dir>` is a comma-separated list of directory names.

Auditing Events from Custom Security Providers

The sections covers the following topics:

- How Events are Audited
- Security Services and the Auditor Service
- Adding Auditing to a Custom Security Provider

How Events are Audited

Auditing is the process whereby information about operating requests and the outcome of those requests are collected, stored, and distributed for the purposes of non-repudiation**.** Auditing providers capture this electronic trail of computer activity.

Each type of security provider can call the configured Auditing providers with a request to write out information about security-related events, before or after these events take place. For example, if a user attempts to access a `withdraw` method in a bank account application (to which they should not have access), the Authorization provider can request that this operation be recorded. Security-related events are only recorded when they meet or exceed the severity level specified in the configuration of the Auditing providers.

The following sections provide the background information you need to understand before adding auditing capability to your custom security providers, and provide step-by-step instructions for adding auditing capability to a custom security provider:

- Security Services and the Auditor Service
- Adding Auditing to a Custom Security Provider

Security Services and the Auditor Service

The `SecurityServices` object passed to a provider during initialization allows providers to retrieve services from the Security Framework. One of these services is the auditor service which you can use in a custom provider to audit events.

The `SecurityServices` object implements the `weblogic.security.spi.SecurityServices` interface, which contains the `getAuditorService` method used to retrieve the auditor service.

The auditor service provides security providers with auditing capabilities, through the `providerAuditWriteEvent` method. The Security Framework forwards this event to the standard `writeEvent` of the audit channel. For more information about the `writeEvent` method, see "Implement the AuditChannel SSPI" on page 5-20. For more information about `AuditEvent` objects, see "Creating an Audit Event" on page 6-3.) The Auditor Service interface includes the `providerAuditWriteEvent` method, described in Table 6-1.

Table 6-1 providerAuditWriteEvent Method

Method	Description
providerAuditWriteEvent	The `providerAuditWriteEvent` method gives security providers write access to the object in the Security Framework that calls the configured Auditing providers. The `event` parameter is an `AuditEvent` object that contains the audit criteria, including the type of event to audit and the audit severity level. For more information about Audit Events and audit severity levels, see "Creating an Audit Event" on page 6-3 and "Audit Severity and the AuditSeverity Class" on page 6-7, respectively.

Security providers designed with auditing capabilities need to obtain the Auditor Service as described in "Obtain and Use the Auditor Service to Write Audit Events" on page 6-8.

Adding Auditing to a Custom Security Provider

To add auditing capability to your custom security provider, perform the following tasks:

- Creating an Audit Event
- Obtain and Use the Auditor Service to Write Audit Events

Examples for each of these tasks are provided in "Example: Implementation of the AuditRoleEvent Interface" on page 7-21 and "Example: Obtaining and Using the Auditor Service to Write Role Audit Events" on page 7-23, respectively.

Note: If your custom security provider is to record audit events, be sure to include any classes created as a result of these steps in the MBean JAR File for the custom security provider (that is, in addition to the other files that are required).

Creating an Audit Event

Security providers must provide information about the events you want audited, such as the type of event and the audit severity. **Audit Events** contain this information and other contextual data that is understandable to a configured Auditing provider. For information on how to create an Audit Event, see the following sections:

- Implementing the AuditEvent SSPI
- Implementing an AuditEvent Interface
- Implement the Provider Audit Record (described in "Using the ProviderAuditRecord Interface" on page 4-7)

In addition to the procedures for creating an audit event, the section covers the following topics:

- Audit Severity and the AuditSeverity Class
- AuditContext Interface

Implementing the AuditEvent SSPI

To implement the AuditEvent SSPI, provide implementations for the methods described in Table 6-2.

Table 6-2 AuditEvent SSPI Methods

Method	Description
`getEventType`	The `getEventType` method returns a string representation of the event type being audited, as used by the Audit Channel (that is, the runtime class that implements the Audit Channel SSPI). For example, the event type for the BEA-provided implementation is "`Authentication Audit Event`". For more information, see "Audit Channels" on page 3-15 and "Implement the AuditChannel SSPI" on page 5-20.
`getFailureException`	The `getFailureException` method returns an `Exception` object used by the Audit Channel to obtain audit information, in addition to the information provided by "`toString`" on page 6-5.
`getSeverity`	The `getSeverity` method returns the severity level value associated with the event type being audited, as used by the Audit Channel. This allows the Audit Channel to make the decision about whether or not to audit. For more information, see "Audit Severity and the AuditSeverity Class" on page 6-7.
`toString`	The `toString` method returns pre-formatted audit information to the Audit Channel.

Implementing an AuditEvent Interface

There are several sub-interfaces of the AuditEvent SSPI that are provided for your convenience, and that can assist you in structuring and creating Audit Events. An Audit Channel can use each of these interfaces (that is, a runtime class that implements the Audit Channel SSPI), to more effectively determine the instance types of extended event type objects for a certain type of security provider. For example, the an Audit Channel can use the `AuditAtnEvent` interface to determine the instance types of extended authentication event type objects. For more information, see "Audit Channels" on page 3-15 and "Implement the AuditChannel SSPI" on page 5-20. It is recommended, but not required, that you implement one of the Audit Event interfaces.The following sections provide information on how to implement the Audit Event interfaces:

- AuditAtnEvent Interface
- AuditAtzEvent and AuditPolicyEvent Interfaces
- AuditMgmtEvent Interface

AuditAtnEvent Interface

The `AuditAtnEvent` interface helps Audit Channels determine instance types of extended authentication event type objects. To implement the `AuditAtnEvent` interface, provide implementations for the methods described in Table 6-2 and the `AuditAtnEvent` interface methods, described in Table 6-3.

Table 6-3 AuditAtnEvent Interface Methods

Method	Description
`getUsername`	The `getUsername` method returns the username associated with the authentication event.
`AtnEventType`	The `AtnEventType` method returns an event type that represents the authentication event. The specific authentication event types are: `AUTHENTICATE`—simple authentication using a username and password occurred. `ASSERTIDENTITY`—perimeter authentication based on tokens occurred. `IMPERSONATEIDENTITY`—client identity was established using the supplied client username. `VALIDATEIDENTITY`—authenticity (trust) of the principals within the supplied subject was validated. `USERLOCKED`—a user account was locked because of invalid login attempts. `USERUNLOCKED`—a lock on a user account was cleared. `USERLOCKOUTEXPIRED`—a lock on a user account expired.
`toString`	The `toString` method returns the specific authentication information to audit, represented as a string.

Note: The `AuditAtnEvent` convenience interface extends both the `AuditEvent` and `AuditContext` interfaces. For more information about the `AuditContext` interface, see AuditContext Interface.

AuditAtzEvent and AuditPolicyEvent Interfaces

The `AuditAtzEvent` and `AuditPolicyEvent` interfaces help Audit Channels determine instance types of extended authorization event type objects.

Note: The difference between the `AuditAtzEvent` interface and the `AuditPolicyEvent` interface is that the latter only extends the `AuditEvent` interface. It does not extend the `AuditContext` interface. For more information about the `AuditContext` interface, see AuditContext Interface.

To implement the `AuditAtzEvent` or `AuditPolicyEvent` interface, provide implementations for the methods described in Table 6-2 and the `AuditPolicyEvent` interface methods, described in Table 6-4. To implement the AuditAtzEvent interface, you must also provide implementation for the methods defined by the AuditContext interface (see "AuditContext Interface" on page 6-8).

Table 6-4 AuditPolicyEvent interface Methods

Method	Description
`getSubject`	The `getSubject` method returns the subject associated with the authorization event (that is, the subject attempting to access the resource).
`getResource`	The `getResource` method returns the resource associated with the authorization event the subject is attempting to access.

AuditMgmtEvent Interface

The `AuditMgmtEvent` interface helps Audit Channels determine instance types of extended security management event type objects. You must implement the methods described in Table 6-2.

AuditRoleEvent Interface

The `AuditRoleEvent` interface helps Audit Channels determine instance types of extended role mapping event type objects. They contain no methods that you must implement, but maintain the best practice structure for an Audit Event implementation. You must implement the methods described in Table 6-2 and Table 6-6 as described in "AuditContext Interface" on page 6-8).

AuditCredentialMappingEvent

The `AuditCredentialMappingEvent` interface helps Audit Channels determine instance types of credential mapping event type objects. You must implement the methods described in Table 6-5.

Table 6-5 AuditCredentialMappingEvent Interface Methods

Method	Description
`getCredentialTypes`	Gets the string array of credential types requested in the getCredential operation associated with this AuditCredentialMappingEvent.
`getInitiatorString`	Gets the initiator of the getCredential operation associated with this AuditCredentialMappingEvent.
`getInitiatorSubject`	Gets the subject of the initiator of the getCredential operation associated with this AuditCredentialMappingEvent.
`getRequestorSubject`	Gets the requestor subject of the getCredential operation associated with this AuditCredentialMappingEvent.
`getResource`	Gets the resource of the getCredential operation associated with this AuditCredentialMappingEvent

AuditRoleDeploymentEvent

The `AuditRoleDeploymentEvent` provides a convenience interface for Auditing providers to determine the instance types of extended `AuditEvent` type objects. You must implement the methods described in Table 6-2.

Audit Severity and the AuditSeverity Class

The **audit severity** is the level at which a security provider wants to record audit events. When a configured Auditing provider receive a request to audit, it examines the severity level of events taking place. If the severity level of an event is greater than or equal to the level an Auditing provider was configured with, that Auditing provider records the audit data.

The `AuditSeverity` class, which is part of the `weblogic.security.spi` package, provides audit severity levels as both numeric and text values to the Audit Channel (that is, the Audit Channel SSPI implementation) through the `AuditEvent` object. The numeric severity value is used in logic, and the text severity value is used in the composition of the audit record output. For more i

nformation about the Audit Channel SSPI and the `AuditEvent` object, see Implement the AuditChannel SSPI and “Creating an Audit Event” on page 6-3, respectively.

AuditContext Interface

Some of the Audit Event interfaces extend the AuditContext interface to indicate that an implementation also contains contextual information. This contextual information can then be used by Audit Channels. For more information, see "Audit Channels" on page 3-15 and "Implement the AuditChannel SSPI" on page 5-20. The Audit Context interface includes the `getContext` method, described in Table 6-6.

Table 6-6 AuditContext Interface Method

Method	Description
`getContext`	The `getContext` method returns a `ContextHandler` object used by the runtime class (that is, the Audit Channel SSPI implementation) to obtain additional audit information. For more information about ContextHandlers, see "ContextHandler Object" on page 6-8.

Obtain and Use the Auditor Service to Write Audit Events

To obtain and use the Auditor Service to write audit events from a custom security provider, follow these steps:

1. Use the `getAuditorService` method on the `SecurityServices` object to return the Audit Service.

   Note: Recall that a `SecurityServices` object is passed into a security provider implementation of a "Provider" SSPI as part of the `initialize` method. (For more information, see Table 4-1.) An `AuditorService` object is only returned if an Auditing provider is configured.

2. Instantiate the Audit Event you created in "Implementing the AuditEvent SSPI" on page 6-3 and send it to the Auditor Service through the `AuditService.providerAuditWriteEvent` method.

ContextHandler Object

A `ContexHandler` is a class interface that obtains additional context and container-specific information from a resource container, and provides that information to security providers making access or role mapping decisions. The `ContextHandler` interface provides a way for an internal container to pass additional information to a Security Framework call, so that a security provider can obtain contextual information beyond what is provided by the arguments to a particular method. A `ContextHandler` is essentially a name/value list and as such, it requires that a security provider know what names to look for. In other words, use of a `ContextHandler` requires close cooperation between the resource container and the security provider.) Each name/value pair in a `ContextHandler` is known as a context element, and is represented by a `ContextElement` object.

A context handler is included with some event types to allow an audit provider to extract other information about the state of the application server at the time of the audit event. The auditing provider may log this other contextual information as a way to elaborate on the event and provide other useful information about the causes of the event.

Best Practice: Posting Audit Events from a Provider's MBean

Provider’s management operations that do writes (for example, create user, delete user, remove data) should post audit events, regardless of whether or not the operation succeeds.

If your provider audits MBean operations, you should keep the following Best Practice guidelines in mind.

- If the write operation succeeds, post an INFORMATION audit event.
- If the write operation fails because of a bad parameter (for example, because the user already exists, or due to a bad import format name, a non-existent file name, or the wrong file format), do not post an audit event.
- If the write operation fails because of an error (for example, LDAPException, RuntimeException), post a FAILURE audit event.
- Import operations can partially succeed. For example, some of the users are imported, but others are skipped because there are already users with that name in the provider.
- If you can easily detect that the data you are skipping is identical to the data already in the provider (for example, the username, description, and password are the same) then consider posting a WARNING event.
- If you are skipping data because there is a partial collision (for example, the username is the same but the password is different), you should post a FAILURE event.
- If it is too difficult to distinguish the import data from the data already stored in theprovider, post a FAILURE event.

Code Examples for Developing Security Providers

This section includes the following security provider code examples:

- Example: Creating the Runtime Classes for the Sample Authentication Provider
- Example: Creating the Runtime Class for the Sample Identity Assertion Provider
- Example: Creating the Runtime Class for the Sample Authorization Provider
- Example: Creating the Runtime Class for the Sample Role Mapping Provider
- Example: Creating the Runtime Class for the Sample Auditing Provider
- Example: Implementation of the AuditRoleEvent Interface
- Example: Obtaining and Using the Auditor Service to Write Role Audit Events

You can create any of the WebLogic Server 9.x\10.0 security provider types, as described in Developing WebLogic Security Providers. However, doing so affects how you then manage that provider. See "Security Provider Management Concepts" on page 3-2 for management considerations.

Example: Creating the Runtime Classes for the Sample Authentication Provider

Listing 7-1 shows the `SampleAuthenticationProviderImpl.java` class, which is one of two runtime classes for the sample Authentication provider. This runtime class includes implementations for:

- The three methods inherited from the `SecurityProvider` interface: `initialize`, `getDescription` and `shutdown`
- The four methods in the `AuthenticationProvider` SSPI: the `getLoginModuleConfiguration`, `getAssertionModuleConfiguration`, `getPrincipalValidator`, and `getIdentityAsserter` methods.

Note: The bold face code in Listing 7-1 highlights the class declaration and the method signatures.

Listing 7-1 SampleAuthenticationProviderImpl.java

```
package examples.security.providers.authentication;

import java.util.HashMap;
import javax.security.auth.login.AppConfigurationEntry;
import javax.security.auth.login.AppConfigurationEntry.LoginModuleControlFlag;
import weblogic.management.security.ProviderMBean;
import weblogic.security.provider.PrincipalValidatorImpl;
import weblogic.security.spi.AuthenticationProvider;
import weblogic.security.spi.IdentityAsserter;
import weblogic.security.spi.PrincipalValidator;
import weblogic.security.spi.SecurityServices;

public final class SampleAuthenticationProviderImpl implements
AuthenticationProvider
{
   private String description;
   private SampleAuthenticatorDatabase database;
   private LoginModuleControlFlag controlFlag;

   public void initialize(ProviderMBean mbean, SecurityServices services)
   {
      System.out.println("SampleAuthenticationProviderImpl.initialize");
      SampleAuthenticatorMBean myMBean = (SampleAuthenticatorMBean)mbean;
      description = myMBean.getDescription() + "\n" + myMBean.getVersion();
      database = new SampleAuthenticatorDatabase(myMBean);

      String flag = myMBean.getControlFlag();
      if (flag.equalsIgnoreCase("REQUIRED")) {
        controlFlag = LoginModuleControlFlag.REQUIRED;
      } else if (flag.equalsIgnoreCase("OPTIONAL")) {
        controlFlag = LoginModuleControlFlag.OPTIONAL;
      } else if (flag.equalsIgnoreCase("REQUISITE")) {
        controlFlag = LoginModuleControlFlag.REQUISITE;
      } else if (flag.equalsIgnoreCase("SUFFICIENT")) {
        controlFlag = LoginModuleControlFlag.SUFFICIENT;
      } else {
        throw new IllegalArgumentException("invalid flag value" + flag);
      }
   }

   public String getDescription()
   {
      return description;
   }

   public void shutdown()
   {
      System.out.println("SampleAuthenticationProviderImpl.shutdown");
   }

   private AppConfigurationEntry getConfiguration(HashMap options)
   {
      options.put("database", database);
      return new
        AppConfigurationEntry(
          "examples.security.providers.authentication.SampleLoginModuleImpl",
          controlFlag,
          options
        );
   }

   public AppConfigurationEntry getLoginModuleConfiguration()
   {
      HashMap options = new HashMap();
      return getConfiguration(options);
   }

   public AppConfigurationEntry getAssertionModuleConfiguration()
   {
      HashMap options = new HashMap();
      options.put("IdentityAssertion","true");
      return getConfiguration(options);
   }

   public PrincipalValidator getPrincipalValidator()
   {
      return new PrincipalValidatorImpl();
   }

   public IdentityAsserter getIdentityAsserter()
   {
      return null;
   }

}
```

Listing 7-2 shows the `SampleLoginModuleImpl.java` class, which is one of two runtime classes for the sample Authentication provider. This runtime class implements the JAAS LoginModule interface (as described in “Implementing the JAAS LoginModule Interface” on page 5-9), and therefore includes implementations for its `initialize`, `login`, `commit`, `abort`, and `logout` methods.

Note: The bold face code in Listing 7-2 highlights the class declaration and the method signatures.

Listing 7-2 SampleLoginModuleImpl.java

```
package examples.security.providers.authentication;

import java.io.IOException;
import java.util.Enumeration;
import java.util.Map;
import java.util.Vector;
import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.UnsupportedCallbackException;
import javax.security.auth.login.LoginException;
import javax.security.auth.login.FailedLoginException;
import javax.security.auth.spi.LoginModule;
import weblogic.management.utils.NotFoundException;
import weblogic.security.spi.WLSGroup;
import weblogic.security.spi.WLSUser;
import weblogic.security.principal.WLSGroupImpl;
import weblogic.security.principal.WLSUserImpl;

final public class SampleLoginModuleImpl implements LoginModule
{
   private Subject subject;
   private CallbackHandler callbackHandler;
   private SampleAuthenticatorDatabase database;

   // Determine whether this is a login or assert identity
   private boolean isIdentityAssertion;

   // Authentication status
   private boolean loginSucceeded;
   private boolean principalsInSubject;
   private Vector principalsForSubject = new Vector();

   public void initialize(Subject subject, CallbackHandler callbackHandler, Map
   sharedState, Map options)
   {
      // only called (once!) after the constructor and before login

      System.out.println("SampleLoginModuleImpl.initialize");
      this.subject = subject;
      this.callbackHandler = callbackHandler;

      // Check for Identity Assertion option
      isIdentityAssertion =
         "true".equalsIgnoreCase((String)options.get("IdentityAssertion"));

      database = (SampleAuthenticatorDatabase)options.get("database");
   }

   public boolean login() throws LoginException
   {
      // only called (once!) after initialize

      System.out.println("SampleLoginModuleImpl.login");

      // loginSucceeded        should be false
      // principalsInSubject   should be false
      // user                  should be null
      // group                 should be null

      Callback[] callbacks = getCallbacks();

      String userName = getUserName(callbacks);

      if (userName.length() > 0) {
         if (!database.userExists(userName)) {
            throwFailedLoginException("Authentication Failed: User " + userName
            + " doesn't exist.");
         }
        if (!isIdentityAssertion) {
         String passwordWant = null;
         try {
            passwordWant = database.getUserPassword(userName);
         } catch (NotFoundException shouldNotHappen) {}
            String passwordHave = getPasswordHave(userName, callbacks);
            if (passwordWant == null || !passwordWant.equals(passwordHave)) {
               throwFailedLoginException(
                 "Authentication Failed: User " + userName + " bad password. " +
                 "Have " + passwordHave + ".  Want " + passwordWant + "."
               );
            }
         }
         } else {
          // anonymous login - let it through?
         System.out.println("\tempty userName");
         }

         loginSucceeded = true;
         principalsForSubject.add(new WLSUserImpl(userName));
         addGroupsForSubject(userName);

         return loginSucceeded;
   }

   public boolean commit() throws LoginException
   {
      // only called (once!) after login

      // loginSucceeded      should be true or false
      // principalsInSubject should be false
      // user      should be null if !loginSucceeded, null or not-null otherwise
      // group     should be null if user == null, null or not-null otherwise

      System.out.println("SampleLoginModule.commit");
      if (loginSucceeded) {
         subject.getPrincipals().addAll(principalsForSubject);
         principalsInSubject = true;
         return true;
      } else {
         return false;
      }
   }

   public boolean abort() throws LoginException
   {
      // The abort method is called to abort the authentication process. This is
      // phase 2 of authentication when phase 1 fails. It is called if the
      // LoginContext's overall authentication failed.

      // loginSucceeded      should be true or false
      // user      should be null if !loginSucceeded, otherwise null or not-null
      // group     should be null if user == null, otherwise null or not-null
      // principalsInSubject      should be false if user is null, otherwise
true
      //                          or false

      System.out.println("SampleLoginModule.abort");
      if (principalsInSubject) {
         subject.getPrincipals().removeAll(principalsForSubject);
         principalsInSubject = false;
      }
      return true;
   }

   public boolean logout() throws LoginException
   {
      // should never be called
      System.out.println("SampleLoginModule.logout");
      return true;
   }

   private void throwLoginException(String msg) throws LoginException
   {
      System.out.println("Throwing LoginException(" + msg + ")");
      throw new LoginException(msg);
   }

   private void throwFailedLoginException(String msg) throws
FailedLoginException
   {
      System.out.println("Throwing FailedLoginException(" + msg + ")");
      throw new FailedLoginException(msg);
   }

   private Callback[] getCallbacks() throws LoginException
   {
      if (callbackHandler == null) {
         throwLoginException("No CallbackHandler Specified");
      }

      if (database == null) {
         throwLoginException("database not specified");
      }

      Callback[] callbacks;
      if (isIdentityAssertion) {
         callbacks = new Callback[1];
      } else {
         callbacks = new Callback[2];
         callbacks[1] = new PasswordCallback("password: ",false);
      }
      callbacks[0] = new NameCallback("username: ");

      try {
          callbackHandler.handle(callbacks);
      } catch (IOException e) {
         throw new LoginException(e.toString());
      } catch (UnsupportedCallbackException e) {
         throwLoginException(e.toString() + " " + e.getCallback().toString());
      }
      return callbacks;
   }

   private String getUserName(Callback[] callbacks) throws LoginException
   {
      String userName = ((NameCallback)callbacks[0]).getName();
      if (userName == null) {
         throwLoginException("Username not supplied.");
      }
      System.out.println("\tuserName\t= " + userName);
      return userName;
   }

   private void addGroupsForSubject(String userName)
   {
      for (Enumeration e = database.getUserGroups(userName);
         e.hasMoreElements();) {
            String groupName = (String)e.nextElement();
            System.out.println("\tgroupName\t= " + groupName);
            principalsForSubject.add(new WLSGroupImpl(groupName));
      }
   }

   private String getPasswordHave(String userName, Callback[] callbacks) throws
   LoginException
   {
      PasswordCallback passwordCallback = (PasswordCallback)callbacks[1];
      char[] password = passwordCallback.getPassword();
      passwordCallback.clearPassword();
      if (password == null || password.length < 1) {
         throwLoginException("Authentication Failed: User " + userName + ".
            Password not supplied");
      }
      String passwd = new String(password);
      System.out.println("\tpasswordHave\t= " + passwd);
      return passwd;
   }

}
```

Example: Creating the Runtime Class for the Sample Identity Assertion Provider

Listing 7-3 shows the `SampleIdentityAsserterProviderImpl.java` class, which is the runtime class for the sample Identity Assertion provider. This runtime class includes implementations for:

- The three methods inherited from the `SecurityProvider` interface: `initialize`, `getDescription`, and `shutdown`
- The four methods in the `AuthenticationProvider` SSPI: the `getLoginModuleConfiguration`, `getAssertionModuleConfiguration`, `getPrincipalValidator`, and `getIdentityAsserter` methods
- The method in the `IdentityAsserter` SSPI: the `assertIdentity` method.

Note: The bold face code in Listing 7-3 highlights the class declaration and the method signatures.

Listing 7-3 SampleIdentityAsserterProviderImpl.java

```
package examples.security.providers.identityassertion;

import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.login.AppConfigurationEntry;
import weblogic.management.security.ProviderMBean;
import weblogic.security.spi.AuthenticationProvider;
import weblogic.security.spi.IdentityAsserter;
import weblogic.security.spi.IdentityAssertionException;
import weblogic.security.spi.PrincipalValidator;
import weblogic.security.spi.SecurityServices;

public final class SampleIdentityAsserterProviderImpl implements
AuthenticationProvider, IdentityAsserter
{
   final static private String TOKEN_TYPE   = "SamplePerimeterAtnToken";
   final static private String TOKEN_PREFIX = "username=";

   private String description;

   public void initialize(ProviderMBean mbean, SecurityServices services)
   {
      System.out.println("SampleIdentityAsserterProviderImpl.initialize");
      SampleIdentityAsserterMBean myMBean = (SampleIdentityAsserterMBean)mbean;
      description = myMBean.getDescription() + "\n" + myMBean.getVersion();
   }

   public String getDescription()
   {
      return description;
   }

   public void shutdown()
   {
      System.out.println("SampleIdentityAsserterProviderImpl.shutdown");
   }

   public AppConfigurationEntry getLoginModuleConfiguration()
   {
      return null;
   }

   public AppConfigurationEntry getAssertionModuleConfiguration()
   {
      return null;
   }

   public PrincipalValidator getPrincipalValidator()
   {
      return null;
   }

   public IdentityAsserter getIdentityAsserter()
   {
      return this;
   }

   public CallbackHandler assertIdentity(String type, Object token) throws
   IdentityAssertionException
   {
      System.out.println("SampleIdentityAsserterProviderImpl.assertIdentity");
      System.out.println("\tType\t\t= "  + type);
      System.out.println("\tToken\t\t= " + token);

      if (!(TOKEN_TYPE.equals(type))) {
         String error = "SampleIdentityAsserter received unknown token type \""
            + type + "\"." + " Expected " + TOKEN_TYPE;
         System.out.println("\tError: " + error);
         throw new IdentityAssertionException(error);
      }

      if (!(token instanceof byte[])) {
         String error = "SampleIdentityAsserter received unknown token class \""
            + token.getClass() + "\"." + " Expected a byte[].";
         System.out.println("\tError: " + error);
         throw new IdentityAssertionException(error);
      }

      byte[] tokenBytes = (byte[])token;
      if (tokenBytes == null || tokenBytes.length < 1) {
         String error = "SampleIdentityAsserter received empty token byte
array";
         System.out.println("\tError: " + error);
         throw new IdentityAssertionException(error);
      }

      String tokenStr = new String(tokenBytes);

      if (!(tokenStr.startsWith(TOKEN_PREFIX))) {
         String error = "SampleIdentityAsserter received unknown token string
\""
            + type + "\"." + " Expected " + TOKEN_PREFIX + "username";
         System.out.println("\tError: " + error);
         throw new IdentityAssertionException(error);
      }

      String userName = tokenStr.substring(TOKEN_PREFIX.length());
      System.out.println("\tuserName\t= " + userName);
      return new SampleCallbackHandlerImpl(userName);
   }

}
```

Listing 7-4 shows the sample `CallbackHandler` implementation that is used along with the `SampleIdentityAsserterProviderImpl.java` runtime class. This `CallbackHandler` implementation is used to send the username back to an Authentication provider's LoginModule.

Listing 7-4 SampleCallbackHandlerImpl.java

```
package examples.security.providers.identityassertion;

import javax.security.auth.callback.Callback;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;

/*package*/ class SampleCallbackHandler implements CallbackHandler
{
   private String userName;
   /*package*/ SampleCallbackHandlerImpl(String user)
   {
      userName = user;
   }

   public void handle(Callback[] callbacks) throws UnsupportedCallbackException
   {
      for (int i = 0; i < callbacks.length; i++) {

            Callback callback = callbacks[i];

            if (!(callback instanceof NameCallback)) {
               throw new UnsupportedCallbackException(callback, "Unrecognized
                  Callback");
            }

            NameCallback nameCallback = (NameCallback)callback;
            nameCallback.setName(userName);
      }
   }
}
```

Example: Creating the Runtime Class for the Sample Authorization Provider

Listing 7-5 shows the `SampleAuthorizationProviderImpl.java` class, which is the runtime class for the sample Authorization provider. This runtime class includes implementations for:

- The three methods inherited from the `SecurityProvider` interface: `initialize`, `getDescription` and `shutdown`
- The method inherited from the `AuthorizationProvider` SSPI: the `getAccessDecision` method.
- The two methods in the `AccessDecision` SSPI: the `isAccessAllowed` and `isProtectedResource` methods.

Note: The bold face code in Listing 7-5 highlights the class declaration and the method signatures.

Listing 7-5 SampleAuthorizationProviderImpl.java

```
package examples.security.providers.authorization;
```

```
import java.security.Principal;
import java.util.Enumeration;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
import javax.security.auth.Subject;
import weblogic.management.security.ProviderMBean;
import weblogic.security.WLSPrincipals;
import weblogic.security.service.ContextHandler;
import weblogic.security.spi.AccessDecision;
import weblogic.security.spi.DeployableAuthorizationProvider;
import weblogic.security.spi.Direction;
import weblogic.security.spi.InvalidPrincipalException;
import weblogic.security.spi.Resource;
import weblogic.security.spi.ResourceCreationException;
import weblogic.security.spi.ResourceRemovalException;
import weblogic.security.spi.Result;
import weblogic.security.spi.SecurityServices;

public final class SampleAuthorizationProviderImpl implements
AuthorizationProvider, AccessDecision
{
   private String description;
   private SampleAuthorizerDatabase database;

   public void initialize(ProviderMBean mbean, SecurityServices services)
   {
      System.out.println("SampleAuthorizationProviderImpl.initialize");
      SampleAuthorizerMBean myMBean = (SampleAuthorizerMBean)mbean;
      description = myMBean.getDescription() + "\n" + myMBean.getVersion();
      database = new SampleAuthorizerDatabase(myMBean);
   }

   public String getDescription()
   {
      return description;
   }

   public void shutdown()
   {
      System.out.println("SampleAuthorizationProviderImpl.shutdown");
   }

   public AccessDecision getAccessDecision()
   {
      return this;
   }

   public Result isAccessAllowed(Subject subject, Map roles, Resource resource,
   ContextHandler handler, Direction direction) throws
```

```
InvalidPrincipalException
   {
      System.out.println("SampleAuthorizationProviderImpl.isAccessAllowed");
      System.out.println("\tsubject\t= " + subject);
      System.out.println("\troles\t= " + roles);
      System.out.println("\tresource\t= " + resource);
      System.out.println("\tdirection\t= " + direction);

      Set principals = subject.getPrincipals();

      for (Resource res = resource; res != null; res = res.getParentResource()) {
         if (database.policyExists(res)) {
            return isAccessAllowed(res, principals, roles);
         }
      }
      return Result.ABSTAIN;
   }

   public boolean isProtectedResource(Subject subject, Resource resource) throws
   InvalidPrincipalException
   {
      System.out.println("SampleAuthorizationProviderImpl.
        isProtectedResource");
      System.out.println("\tsubject\t= " + subject);
      System.out.println("\tresource\t= " + resource);

      for (Resource res = resource; res != null; res = res.getParentResource()) {
         if (database.policyExists(res)) {
            return true;
         }
      }
      return false;
   }

   private Result isAccessAllowed(Resource resource, Set principals, Map roles)
   {
      for (Enumeration e = database.getPolicy(resource); e.hasMoreElements();)
      {
       String principalOrRoleNameAllowed = (String)e.nextElement();
       if (WLSPrincipals.getEveryoneGroupname().
         equals(principalOrRoleNameAllowed) ||
         (WLSPrincipals.getUsersGroupname().equals(principalOrRoleNameAllowed)
         && !principals.isEmpty()) || principalsOrRolesContain(principals,
         roles, principalOrRoleNameAllowed))
         {
            return Result.PERMIT;
         }
      }
      return Result.DENY;
```

```
    }
}
```

Example: Creating the Runtime Class for the Sample Role Mapping Provider

Listing 7-6 shows the `SampleRoleMapperProviderImpl.java` class, which is the runtime class for the sample Role Mapping provider. This runtime class includes implementations for:

- The three methods inherited from the `SecurityProvider` interface: `initialize`, `getDescription` and `shutdown`.
- The method inherited from the `RoleProvider` SSPI: the `getRoleMapper` method.
- The method in the `RoleMapper` SSPI: the `getRoles` method.

Note: The bold face code in Listing 7-6 highlights the class declaration and the method signatures.

Listing 7-6 SampleRoleMapperProviderImpl.java

```
package examples.security.providers.roles;

import java.security.Principal;
import java.util.Collections;
import java.util.Enumeration;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Properties;
import java.util.Set;
import javax.security.auth.Subject;
import weblogic.management.security.ProviderMBean;
import weblogic.security.WLSPrincipals;
import weblogic.security.service.ContextHandler;
import weblogic.security.spi.DeployableRoleProvider;
import weblogic.security.spi.Resource;
import weblogic.security.spi.RoleCreationException;
import weblogic.security.spi.RoleMapper;
import weblogic.security.spi.RoleRemovalException;
import weblogic.security.spi.SecurityServices;
```

```
public final class SampleRoleMapperProviderImpl implements RoleProvider,
RoleMapper
{
   private String description;
   private SampleRoleMapperDatabase database;
   private static final Map NO_ROLES = Collections.unmodifiableMap(new
     HashMap(1));

   public void initialize(ProviderMBean mbean, SecurityServices services)
   {
      System.out.println("SampleRoleMapperProviderImpl.initialize");
      SampleRoleMapperMBean myMBean = (SampleRoleMapperMBean)mbean;
      description = myMBean.getDescription() + "\n" + myMBean.getVersion();
      database = new SampleRoleMapperDatabase(myMBean);
   }

   public String getDescription()
   {
      return description;
   }

   public void shutdown()
   {
      System.out.println("SampleRoleMapperProviderImpl.shutdown");
   }

   public RoleMapper getRoleMapper()
   {
      return this;
   }

   public Map getRoles(Subject subject, Resource resource, ContextHandler
   handler)
   {
      System.out.println("SampleRoleMapperProviderImpl.getRoles");
      System.out.println("\tsubject\t= " + subject);
      System.out.println("\tresource\t= " + resource);

      Map roles = new HashMap();
      Set principals = subject.getPrincipals();

      for (Resource res = resource; res != null; res = res.getParentResource())
      {
         getRoles(res, principals, roles);
      }

      getRoles(null, principals, roles);

      if (roles.isEmpty()) {
         return NO_ROLES;
      }
      return roles;
   }

   private void getRoles(Resource resource, Set principals, Map roles)
   {
      for (Enumeration e = database.getRoles(resource); e.hasMoreElements();)
      {
         String role = (String)e.nextElement();
         if (roleMatches(resource, role, principals))
         {
            roles.put(role, new SampleSecurityRoleImpl(role, "no
description"));
         }
      }
   }

   private boolean roleMatches(Resource resource, String role, Set
   principalsHave)
   {
      for (Enumeration e = database.getPrincipalsForRole(resource, role);
        e.hasMoreElements();)
      {
         String principalWant = (String)e.nextElement();
         if (principalMatches(principalWant, principalsHave))
         {
            return true;
         }
      }
      return false;
   }

   private boolean principalMatches(String principalWant, Set principalsHave)
   {
      if (WLSPrincipals.getEveryoneGroupname().equals(principalWant) ||
        (WLSPrincipals.getUsersGroupname().equals(principalWant) &&
        !principalsHave.isEmpty()) || (WLSPrincipals.getAnonymousUsername().
        equals(principalWant) && principalsHave.isEmpty()) ||
        principalsContain(principalsHave, principalWant))
        {
           return true;
        }
      return false;
   }

   private boolean principalsContain(Set principalsHave, String
   principalNameWant)
   {
      for (Iterator i = principalsHave.iterator(); i.hasNext();)
      {
         Principal principal = (Principal)i.next();
         String principalNameHave = principal.getName();
         if (principalNameWant.equals(principalNameHave))
         {
            return true;
         }
      }
      return false;
   }
}
```

Listing 7-7 shows the sample `SecurityRole` implementation that is used along with the `SampleRoleMapperProviderImpl.java` runtime class.

Listing 7-7 SampleSecurityRoleImpl.java

```
package examples.security.providers.roles;

import weblogic.security.service.SecurityRole;

public class SampleSecurityRoleImpl implements SecurityRole
{
   private String _roleName;
   private String _description;
   private int _hashCode;

   public SampleSecurityRoleImpl(String roleName, String description)
   {
      _roleName = roleName;
      _description = description;
      _hashCode = roleName.hashCode() + 17;
   }

   public boolean equals(Object secRole)
   {
      if (secRole == null)
      {
         return false;
      }

      if (this == secRole)
      {
         return true;
      }
      if (!(secRole instanceof SampleSecurityRoleImpl))
      {
         return false;
      }

      SampleSecurityRoleImpl anotherSecRole = (SampleSecurityRoleImpl)secRole;

      if (!_roleName.equals(anotherSecRole.getName()))
      {
         return false;
      }

      return true;
   }

   public String toString () { return _roleName; }
   public int hashCode () { return _hashCode; }
   public String getName () { return _roleName; }
   public String getDescription () { return _description; }
}
```

Example: Creating the Runtime Class for the Sample Auditing Provider

Listing 7-8 shows the `SampleAuditProviderImpl.java` class, which is the runtime class for the sample Auditing provider. This runtime class includes implementations for:

- The three methods inherited from the `SecurityProvider` interface: `initialize`, `getDescription` and `shutdown`.
- The method inherited from the `AuditProvider` SSPI: the `getAuditChannel` method.
- The method in the Audit Channel SSPI: the `writeEvent` method.

Note: The bold face code in Listing 7-8 highlights the class declaration and the method signatures.

Listing 7-8 SampleAuditProviderImpl.java

```
package examples.security.providers.audit;

import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.PrintStream;
import weblogic.management.security.ProviderMBean;
import weblogic.security.spi.AuditChannel;
import weblogic.security.spi.AuditEvent;
import weblogic.security.spi.AuditProvider;
import weblogic.security.spi.SecurityServices;

public final class SampleAuditProviderImpl implements AuditChannel,
AuditProvider
{
   private String description;
   private PrintStream log;

   public void initialize(ProviderMBean mbean, SecurityServices services)
   {
      System.out.println("SampleAuditProviderImpl.initialize");

      description = mbean.getDescription() + "\n" + mbean.getVersion();

      SampleAuditorMBean myMBean = (SampleAuditorMBean)mbean;
      File file = new File(myMBean.getLogFileName());
      System.out.println("\tlogging to " + file.getAbsolutePath());

      try {
         log = new PrintStream(new FileOutputStream(file), true);
      } catch (IOException e) {
         throw new RuntimeException(e.toString());
      }
   }

   public String getDescription()
   {
      return description;
   }

   public void shutdown()
   {
      System.out.println("SampleAuditProviderImpl.shutdown");
      log.close();
   }

   public AuditChannel getAuditChannel()
   {
      return this;
   }

   public void writeEvent(AuditEvent event)
   {
      // Write the event out to the sample Auditing provider’s log file using
      // the event's "toString" method.
```

```
        log.println(event);
    }
}
```

Example: Implementation of the AuditRoleEvent Interface

Listing 7-9 shows the `MyAuditRoleEventImpl.java` class, which is a sample implementation of an Audit Event convenience interface (in this case, the `AuditRoleEvent` convenience interface). This class includes implementations for:

- The four methods inherited from the `AuditEvent` SSPI: `getEventType`, `getFailureException`, `getSeverity` and `toString`
- One additional method: `getContext`, which returns additional contextual information via the ContextHandler.

Note: The bold face code in Listing 7-9 highlights the class declaration and the method signatures.

Listing 7-9 MyAuditRoleEventImpl.java

```
package mypackage;

import javax.security.auth.Subject;
import weblogic.security.SubjectUtils;
import weblogic.security.service.ContextHandler;
import weblogic.security.spi.AuditRoleEvent;
import weblogic.security.spi.AuditSeverity;
import weblogic.security.spi.Resource;

/*package*/ class MyAuditRoleEventImpl implements AuditRoleEvent
{
   private Subject subject;
   private Resource resource;
   private ContextHandler context;
   private String details;
   private Exception failureException;
```

```
/*package*/ MyAuditRoleEventImpl(Subject subject, Resource resource,
   ContextHandler context, String details, Exception
   failureException) {
      this.subject = subject;
      this.resource = resource;
      this.context = context;
      this.details = details;
      this.failureException = failureException;
}
public Exception getFailureException()
{
   return failureException;
}
public AuditSeverity getSeverity()
{
   return (failureException == null) ? AuditSeverity.SUCCESS :
      AuditSeverity.FAILURE;
}
public String getEventType()
{
   return "MyAuditRoleEventType";
}
public ContextHandler getContext()
{
   return context;
}
public String toString()
{
   StringBuffer buf = new StringBuffer();
   buf.append("EventType:" + getEventType() + "\n");
   buf.append("\tSeverity: " +
      getSeverity().getSeverityString());
   buf.append("\tSubject: " +
      SubjectUtils.displaySubject(getSubject());
      buf.append("\tResource: " + resource.toString());
      buf.append("\tDetails: " + details);
      if (getFailureException() != null) {
         buf.append("\n\tFailureException:" +
            getFailureException());
      }
      return buf.toString();
   }
}
```

Example: Obtaining and Using the Auditor Service to Write Role Audit Events

Listing 7-10 illustrates how a custom Role Mapping provider's runtime class (called `MyRoleMapperProviderImpl.java`) would obtain the Auditor Service and use it to write out audit events.

Note: The `MyRoleMapperProviderImpl.java` class relies on the `MyAuditRoleEventImpl.java` class from Listing 7-10.

Listing 7-10 MyRoleMapperProviderImpl.java

```
package mypackage;
import javax.security.auth.Subject;
import weblogic.management.security.ProviderMBean;
import weblogic.security.SubjectUtils;
import weblogic.security.service.ContextHandler;
import weblogic.security.spi.AuditorService;
import weblogic.security.spi.RoleMapper;
import weblogic.security.spi.RoleProvider;
import weblogic.security.spi.Resource;
import weblogic.security.spi.SecurityServices;
```

```
public final class MyRoleMapperProviderImpl implements RoleProvider,
RoleMapper
{
   private AuditorService auditor;

   public void initialize(ProviderMBean mbean, SecurityServices
      services)
   {
      auditor = services.getAuditorService();
      ...
   }

   public Map getRoles(Subject subject, Resource resource,
      ContextHandler handler)
   {
      ...
      if (auditor != null)
      {
         auditor.providerAuditWriteEvent(
            new MyRoleEventImpl(subject, resource, context,
            "why logging this event",
            null);                // no exception occurred
      }
      ...
   }

}
```

MBean Definition File Element Syntax

An **MBean Definition File (MDF)** is an input file to the WebLogic MBeanMaker utility, which uses the file to create an MBean type for managing a custom security provider. An MDF must be formatted as a well-formed and valid XML file that describes a single MBean type. The following sections describe all the elements and attributes that are available for use in a valid MDF:

- "The MBeanType (Root) Element" on page A-1
- "The MBeanAttribute Subelement" on page A-4
- "The MBeanConstructor Subelement" on page A-10
- "The MBeanOperation Subelement" on page A-10
- "Examples: Well-Formed and Valid MBean Definition Files (MDFs)" on page A-16

The MBeanType (Root) Element

All MDFs must contain exactly one root element called `MBeanType`, which has the following syntax:

`<MBeanType Name=` *string   optional_attributes*`>`
        subelements
`</MBeanType>`

The `MBeanType` element must include a `Name` attribute, which specifies the internal, programmatic name of the MBean type. (To specify a name that is visible in a user interface, use the `DisplayName` attribute.) Other attributes are optional.

The following is a simplified example of an `MBeanType` (root) element:

```
<MBeanType Name=“MyMBean” Package=”com.mycompany”>
   <MBeanAttribute Name=“MyAttr” Type=”java.lang.String” Default=“Hello
World”/>
</MBeanType>
```

Attributes specified in the `MBeanType` (root) element apply to the entire set of MBeans instantiated from that MBean type. To override attributes for specific MBean instances, you need to specify attributes in the `MBeanAttribute` subelement. For more information, see “The MBeanAttribute Subelement” on page A-4.

Table A-1 describes the attributes available to the `MBeanType` (root) element. The JMX Specification/BEA Extension column indicates whether the attribute is a BEA extension to the JMX specification or a standard JMX attribute. Note that BEA extensions might not function on other J2EE Web servers.

Table A-1 MBeanType (Root) Element Attributes

Attribute	JMX Specification /BEA Extension	Allowed Values	Description
`Abstract`	BEA Extension	`true`/`false`	A `true` value specifies that the MBean type cannot be instantiated (like any abstract Java class), though other MBean types can inherit its attributes and operations. If you specify `true`, you must create other non-abstract MBean types for carrying out management tasks. If you do not specify a value for this attribute, the assumed value is `false`.
`Deprecated`	BEA Extension	`true`/`false`	Indicates that the MBean type is deprecated. This information appears in the generated Java source, and is also placed in the `ModelMBeanInfo` object for possible use by a management application. If you do not specify this attribute, the assumed value is `false`.

Table A-1 MBeanType (Root) Element Attributes

Attribute	JMX Specification /BEA Extension	Allowed Values	Description
`Description`	JMX Specification	*String*	An arbitrary string associated with the MBean type that appears in various locations, such as the Javadoc for generated classes. There is no default or assumed value. **Note:** To specify a description that is visible in a user interface, use the `DisplayName` attribute.
`DisplayName`	JMX Specification	*String*	The name that a user interface displays to identify instances of MBean types. For an instance of type X, the default `DisplayName` is "instance of type X." This value is typically overridden when instances are created.
`Extends`	BEA Extension	*Pathname*	A fully qualified MBean type name that this MBean type extends.
`Implements`	BEA Extension	*Comma-separated list*	A comma-separated list of fully qualified MBean type names that this MBean type implements. See also `Extends`.
`Name`	JMX Specification	*String*	Mandatory attribute that specifies the internal, programmatic name of the MBean type.

Table A-1 MBeanType (Root) Element Attributes

Attribute	JMX Specification /BEA Extension	Allowed Values	Description
`Package`	BEA Extension	*String*	Specifies the package name of the MBean type and determines the location of the class files that the WebLogic MBeanMaker creates. If you do not specify this attribute, the MBean type is placed in the Java default package. **Note:** MBean type names can be the same as long as the package name varies.
`PersistPolicy`	JMX Specification	`/OnUpdate`	Specifies how persistence will occur: `OnUpdate`. The attribute is stored every time the attribute is updated. Note: When specified in the MBeanType element, this value overrides any setting within an individual MBeanAttribute subelement.

The MBeanAttribute Subelement

You must supply one instance of an `MBeanAttribute` subelement for each attribute in your MBean type. The `MBeanAttribute` subelement must be formatted as follows:

`<MBeanAttribute Name=`*string optional_attributes* `/>`

The `MBeanAttribute` subelement must include a `Name` attribute, which specifies the internal, programmatic name of the Java attribute in the MBean type. (To specify a name that is visible in a user interface, use the `DisplayName` attribute.) Other attributes are optional.

The following is a simplified example of an `MBeanAttribute` subelement within an `MBeanType` element:

```
<MBeanType Name="MyMBean" Package="com.mycompany">
    <MBeanAttribute Name= "WhenToCache"
     Type="java.lang.String"
     LegalValues="'cache-on-reference','cache-at-initialization','cache-never'
"
     Default= "cache-on-reference"
    />
</MBeanType>
```

Attributes specified in an `MBeanAttribute` subelement apply to a specific MBean instance. To set attributes for the entire set of MBeans instantiated from an MBean type, you need to specify attributes in the `MBeanType` (root) element. For more information, see "The MBeanType (Root) Element" on page A-1.

Table A-2 describes the attributes available to the `MBeanAttribute` subelement. The JMX Specification/BEA Extension column indicates whether the attribute is a BEA extension to the JMX specification. Note that BEA extensions might not function on other J2EE Web servers.

Table A-2 MBeanAttribute Subelement Attributes

Attribute	JMX Specification /BEA Extension	Allowed Values	Description
`Default`	JMX Specification	*String*	The value to be returned if the `MBeanAttribute` subelement does not provide a getter method or a cached value. The string represents a Java expression that must evaluate to an object of a type that is compatible with the provided data type for this attribute. If you do not specify this attribute, the assumed value is `null`. If you use this assumed value, and if you set the `LegalNull` attribute to `false`, then an exception is thrown by WebLogic MBeanMaker and WebLogic Server.
`Deprecated`	BEA Extension	`true/false`	Indicates that the MBean attribute is deprecated. This information appears in the generated Java source, and is also placed in the `ModelMBeanInfo` object for possible use by a management application. If you do not specify this attribute, the assumed value is `false`.

Table A-2 MBeanAttribute Subelement Attributes

Attribute	JMX Specification /BEA Extension	Allowed Values	Description
`Description`	JMX Specification	*String*	An arbitrary string associated with the MBean attribute that appears in various locations, such as the Javadoc for generated classes. There is no default or assumed value. **Note:** To specify a description that is visible in a user interface, use the `DisplayName` attribute.
`Dynamic`	BEA Extension	true/false	Changes made to dynamic MBeans take effect without rebooting the server. By default, all custom security provider MBean attributes are non-dynamic. Note that in 8.1 and 7.0, all custom security provider MBean attributes were dynamic.
`Encrypted`	BEA Extension	`true/false`	A `true` value indicates that this MBean attribute will be encrypted when it is set. If you do not specify this attribute, the assumed value is `false`.

Table A-2 MBeanAttribute Subelement Attributes

Attribute	JMX Specification /BEA Extension	Allowed Values	Description
`InterfaceType`	BEA Extension	`String`	Classname of an interface to be used instead of the MBean interface generated by the WebLogic MBeanMaker. `InterfaceType` can be • int • long • float • double • char • byte Do not specify if "Type" is `java.lang.String`, `java.lang.String[]`, or `java.lang.Properties.`
`IsIs`	JMX Specification	`true`/`false`	Specifies whether a generated Java interface uses the JMX `is<AttributeName>` method to access the boolean value of the MBean attribute (as opposed to the `get<AttributeName>` method). If you do not specify this attribute, the assumed value is `false`.
`LegalNull`	BEA Extension	`true`/`false`	Specifies whether `null` is an allowable value for the current `MBeanAttribute` subelement. If you do not specify this attribute, the assumed value is `true`.

Table A-2 MBeanAttribute Subelement Attributes

Attribute	JMX Specification /BEA Extension	Allowed Values	Description
`LegalValues`	BEA Extension	*Comma-separated list*	Specifies a fixed set of allowable values for the current `MBeanAttribute` subelement. If you do not specify this attribute, the MBean attribute allows any value of the type that is specified by the `Type` attribute. **Note:** The items in the list must be convertible to the data type that is specified by the subelement's `Type` attribute.
`Max`	BEA Extension	*Integer*	For numeric MBean attribute types only, provides a numeric value that represents the inclusive maximum value for the attribute. If you do not specify this attribute, the value can be as large as the data type allows.
`Min`	BEA Extension	*Integer*	For numeric MBean attribute types only, provides a numeric value which represents the inclusive minimum value for the attribute. If you do not specify this attribute, the value can be as small as the data type allows.
`Name`	JMX Specification	*String*	Mandatory attribute that specifies the internal, programmatic name of the MBean attribute.

Table A-2 MBeanAttribute Subelement Attributes

Attribute	JMX Specification /BEA Extension	Allowed Values	Description
`Type`	JMX Specification	`Java class name`	The fully qualified classname of the data type of this attribute. This corresponding class must be available on the classpath. If you do not specify this attribute, the assumed value is `java.lang.String`. Type can be • `java.lang.Integer` • `java.lang.Integer[]` • `java.lang.Long` • `java.lang.Long[]` • `java.lang.Float` • `java.lang.Float[]` • `java.lang.Double` • `java.lang.Double[]` • `java.lang.Char` • `java.lang.Char[]` • `java.lang.Byte` • `java.lang.Byte[]` • `java.lang.String` • `java.lang.String[]` • `java.util.Properties`

Table A-2 MBeanAttribute Subelement Attributes

Attribute	JMX Specification /BEA Extension	Allowed Values	Description
`Writeable`	JMX Specification	`true`/`false`	A `true` value allows the MBean API to set an `MBeanAttribute`'s value. If you do not specify this attribute in `MBeanType` or `MBeanAttribute`, the assumed value is `true`. When specified in the `MBeanType` element, this value is considered the default for individual `MBeanAttribute` subelements.

The MBeanConstructor Subelement

`MBeanConstructor` subelements are not currently used by the WebLogic MBeanMaker, but are supported for compliance with the *Java Management eXtensions 1.0 specification* and upward compatibility. Therefore, attribute details for the `MBeanConstructor` subelement (and its associated `MBeanConstructorArg` subelement) are omitted from this documentation.

The MBeanOperation Subelement

You must supply one instance of an `MBeanOperation` subelement for each operation (method) that your MBean type supports. The `MBeanOperation` must be formatted as follows:

```
<MBeanOperation Name=string   optional_attributes >
     <MBeanOperationArg Name=string  optional_attributes />
</MBeanOperation>
```

The `MBeanOperation` subelement must include a `Name` attribute, which specifies the internal, programmatic name of the operation. (To specify a name that is visible in a user interface, use the `DisplayName` attribute.) Other attributes are optional.

Within the `MBeanOperation` element, you must supply one instance of an `MBeanOperationArg` subelement for each argument that your operation (method) uses. The `MBeanOperationArg` must be formatted as follows:

```
<MBeanOperationArg Name=string   optional_attributes  />
```

The `Name` attribute must specify the name of the operation. The only optional attribute for `MBeanOperationArg` is `Type`, which provides the Java class name that specifies behavior for a specific type of Java attribute. If you do not specify this attribute, the assumed value is `java.lang.String`.

The following is a simplified example of an `MBeanOperation` and `MBeanOperationArg` subelement within an `MBeanType` element:

```
<MBeanType Name=“MyMBean” Package=”com.mycompany”>

   <MBeanOperation
    Name= “findParserSelectMBeanByKey”
    ReturnType=”XMLParserSelectRegistryEntryMBean”
    Description=”Given a public ID, system ID, or root element tag, returns the
object name of the corresponding XMLParserSelectRegistryEntryMBean.”
   >
      <MBeanOperationArg Name=”publicID” Type=”java.lang.String”/>
      <MBeanOperationArg Name=”systemID” Type=”java.lang.String”/>
      <MBeanOperationArg Name=”rootTag” Type=”java.lang.String”/>
   </MBeanOperation>

</MBeanType>
```

Table A-3 describes the attributes available to the `MBeanOperation` subelement. The JMX Specification/BEA Extension column indicates whether the attribute is a BEA extension to the JMX specification. Note that BEA extensions might not function on other J2EE Web servers.

Table A-3 MBeanOperation Subelement Attributes

Attribute	JMX Specification /BEA Extension	Allowed Values	Description
`Deprecated`	BEA Extension	`true/false`	Indicates that the MBean operation is deprecated. This information appears in the generated Java source, and is also placed in the `ModelMBeanInfo` object for possible use by a management application. If you do not specify this attribute, the assumed value is `false`.
`Description`	JMX Specification	*String*	An arbitrary string associated with the MBean operation that appears in various locations, such as the Javadoc for generated classes. There is no default or assumed value. **Note:** To specify a description that is visible in a user interface, use the `DisplayName` attribute.

Table A-3 MBeanOperation Subelement Attributes

Attribute	JMX Specification /BEA Extension	Allowed Values	Description
`Name`	JMX Specification	*String*	Mandatory attribute that specifies the internal, programmatic name of the MBean operation.
`ReturnType`	JMX Specification	*`String`*	A string containing the fully qualified classname of the Java object returned by the operation being described. `ReturnType` can be void or the following: • `int` • `int[]` • `long` • `long[]` • `float` • `float[]` • `double` • `double[]` • `char` • `char[]` • `byte` • `byte[]` • `java.lang.String` • `java.lang.String[]` • `java.util.Properties`

Table A-4 describes the attributes available to the `MBeanOperationArg` subelement. The JMX Specification/BEA Extension column indicates whether the attribute is a BEA extension to the JMX specification. Note that BEA extensions might not function on other J2EE Web servers.

Table A-4 MBeanOperationArg Subelement Attributes

Attribute	JMX Specification /BEA Extension	Allowed Values	Description
`Description`	JMX Specification	*String*	An arbitrary string associated with the MBean operation argument that appears in various locations, such as the Javadoc for generated classes. There is no default or assumed value.

Table A-4 MBeanOperationArg Subelement Attributes

Attribute	JMX Specification /BEA Extension	Allowed Values	Description
`Name`	JMX Specification	*String*	Mandatory attribute that specifies the name of the argument.
`Type`	JMX Specification	*String*	The type of the MBean operation argument. If you do not specify this attribute, the assumed value is `java.lang.String`. `Type` can be • `int` • `int[]` • `long` • `long[]` • `float` • `float[]` • `double` • `double[]` • `char` • `char[]` • `byte` • `byte[]` • `java.lang.String` • `java.lang.String[]` • `java.util.Properties`

MBean Operation Exceptions

Your MBean Definition Files (MDFs) must use only JDK exception types or `weblogic.management.utils` exception types. The following is a code fragment from Listing A-1 that shows the use of an `MBeanException` within an `MBeanOperation` subelement:

```
<MBeanOperation
Name = "registerPredicate"
ReturnType = "void"
Description = "Registers a new predicate with the specified class name."
>

<MBeanOperationArg
Name = "predicateClassName"
Type = "java.lang.String"
Description = "The name of the Java class that implements the predicate."
/>

<MBeanException>weblogic.management.utils.InvalidPredicateException</MBean
Exception>

<MBeanException>weblogic.management.utils.AlreadyExistsException</MBeanExc
eption>

</MBeanOperation>
```

Examples: Well-Formed and Valid MBean Definition Files (MDFs)

Listing A-1 and Listing A-2 provide examples of MBean Definition Files (MDFs) that use many of the attributes described in this Appendix. Listing A-1 shows the MDF used to generate an MBean type that manages predicates and reads data about predicates and their arguments.Listing A-2 shows the MDF used to generate the MBean type for the WebLogic (default) Authorization provider.

Listing A-1 PredicateEditor.xml

```
<?xml version="1.0" ?>
<!DOCTYPE MBeanType SYSTEM "commo.dtd">

<MBeanType
Name = "PredicateEditor"
Package = "weblogic.security.providers.authorization"
Implements = "weblogic.security.providers.authorization.PredicateReader"
PersistPolicy = "OnUpdate"
Abstract = "false"
```

```
Description = "This MBean manages predicates and reads data about predicates
and their arguments.&lt;p&gt;"
>

<MBeanOperation
Name = "registerPredicate"
ReturnType = "void"
Description = "Registers a new predicate with the specified class name."
>

<MBeanOperationArg
Name = "predicateClassName"
Type = "java.lang.String"
Description = "The name of the Java class that implements the predicate."
/>


<MBeanException>weblogic.management.utils.InvalidPredicateException</MBean
Exception>

<MBeanException>weblogic.management.utils.AlreadyExistsException</MBeanExc
eption>

</MBeanOperation>


<MBeanOperation
Name = "unregisterPredicate"
ReturnType = "void"
Description = "Unregisters the currently registered predicate."   >


<MBeanOperationArg
Name = "predicateClassName"
Type = "java.lang.String"
Description = "The name of the Java class that implements predicate to be
unregistered."
/>

<MBeanException>weblogic.management.utils.NotFoundException</MBeanExceptio
n>
```

```
</MBeanOperation>
</MBeanType>
```

Listing A-2 DefaultAuthorizer.xml

```
<?xml version="1.0" ?>
<!DOCTYPE MBeanType SYSTEM "commo.dtd">

<MBeanType
Name = "DefaultAuthorizer"
DisplayName = "DefaultAuthorizer"
Package = "weblogic.security.providers.authorization"
Extends ="weblogic.management.security.authorization.DeployableAuthorizer"
Implements = "weblogic.management.security.authorization.PolicyEditor,
weblogic.security.providers.authorization.PredicateEditor"
PersistPolicy = "OnUpdate"
Description = "This MBean represents configuration attributes for the
WebLogic Authorization provider. &lt;p&gt;"
>

<MBeanAttribute
Name = "ProviderClassName"
Type = "java.lang.String"
Writeable = "false"
Default"&quot;weblogic.security.providers.authorization.DefaultAuthorizati
onProviderImpl&quot;"
Description = "The name of the Java class used to load the WebLogic
Authorization provider."
/>

<MBeanAttribute
Name = "Description"
Type = "java.lang.String"
Writeable = "false"
Default = "&quot;Weblogic Default Authorization Provider&quot;"
Description = "A short description of the WebLogic Authorization provider."
/>
```

```
<MBeanAttribute
Name = "Version"
Type = "java.lang.String"
Writeable = "false"
Default = "&quot;1.0&quot;"
Description = "The version of the WebLogic Authorization provider."
/>

</MBeanType>
```