JBoss AS 6.0 Security Guide

Security with JBoss Application Server 6

by Anil Saldhana, Marcus Moyses, and Stefan Guilhen

Part I. Security Overview

Security is a fundamental part of any enterprise application. You need to be able to restrict who is allowed to access your applications and control what operations application users may perform.

The *Java Enterprise Edition* (J2EE) specification defines a simple role-based security model for *Enterprise Java Beans* (EJBs) and web components. The *JBoss Security Extension* (JBossSX) framework handles platform security, and provides support for both the role-based declarative J2EE security model and integration of custom security through a security proxy layer.

The default implementation of the declarative security model is based on *Java Authentication and Authorization Service* (JAAS) login modules and subjects. The security proxy layer allows custom security that cannot be described using the declarative model to be added to an EJB in a way that is independent of the EJB business object.

J2EE Declarative Security Overview

Rather than embedding security into your business component, the J2EE security model is declarative: you describe the security roles and permissions in a standard XML descriptor. This isolates security from business-level code because security tends to be more a function of where the component is deployed than an inherent aspect of the component's business logic.

For example, consider an Automatic Teller Machine (ATM) component used to access a bank account. The security requirements, roles, and permissions of the component will vary independently of how you access the bank account. How you access your account information may also vary based on which bank is managing the account, or where the ATM is located.

Securing a Java EE application is based on the specification of the application security requirements via the standard Java EE deployment descriptors. You secure access to EJBs and web components in an enterprise application by using the `ejb-jar.xml` and `web.xml` deployment descriptors. The following sections look at the purpose and usage of the various security elements.

1.1. Security References

Both EJBs and servlets can declare one or more `security-role-ref` elements as shown in *Figure 1.1, "The security-role-ref element"*. This element declares that a component is using the `role-name` value as an argument to the `isCallerInRole(String)` method. By using the `isCallerInRole` method, a component can verify whether the caller is in a role that has been declared with a `security-role-ref/role-name` element. The `role-name` element value must link to a `security-role` element through the `role-link` element. The typical use of `isCallerInRole` is to perform a security check that cannot be defined by using the role-based `method-permissions` elements.

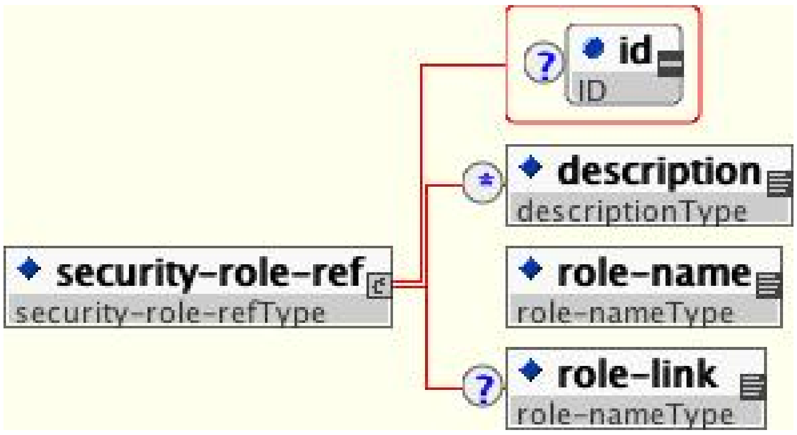

Figure 1.1. The security-role-ref element

Example 1.1, "An ejb-jar.xml descriptor fragment that illustrates the security-role-ref element usage." shows the use of `security-role-ref` in an `ejb-jar.xml`.

Example 1.1. An ejb-jar.xml descriptor fragment that illustrates the security-role-ref element usage.

```
<!-- A sample ejb-jar.xml fragment -->
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>ASessionBean</ejb-name>
      ...
      <security-role-ref>
          <role-name>TheRoleICheck</role-name>
          <role-link>TheApplicationRole</role-link>
      </security-role-ref>
    </session>
  </enterprise-beans>
  ...
</ejb-jar>
```

Example 1.2, “An example web.xml descriptor fragment that illustrates the security-role-ref element usage.” shows the use of `security-role-ref` in a `web.xml`.

Example 1.2. An example web.xml descriptor fragment that illustrates the security-role-ref element usage.

```
<web-app>
    <servlet>
        <servlet-name>AServlet</servlet-name>
        ...
        <security-role-ref>
            <role-name>TheServletRole</role-name>
            <role-link>TheApplicationRole</role-link>
        </security-role-ref>
    </servlet>
    ...
</web-app>
```

1.2. Security Identity

An EJB has the capability to specify what identity an EJB should use when it invokes methods on other components using the `security-identity` element, shown in *Figure 1.2, “The security-identity element”*

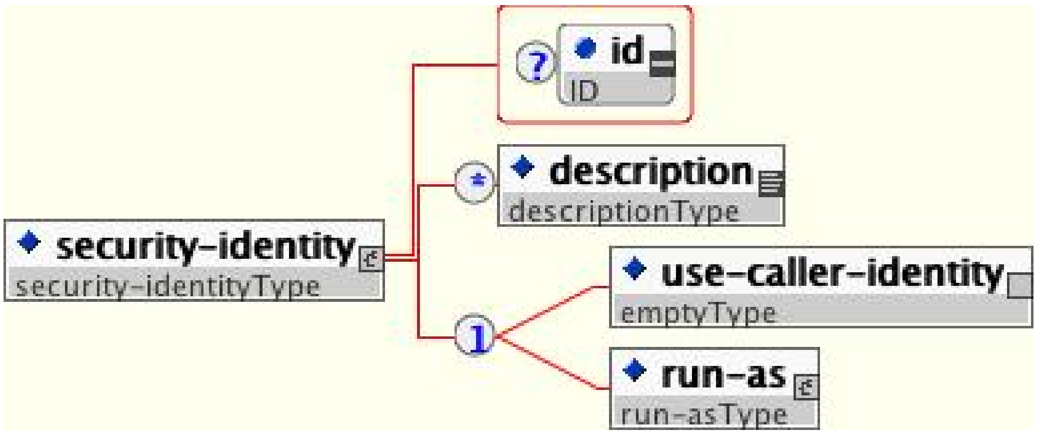

Figure 1.2. The security-identity element

The invocation identity can be that of the current caller, or it can be a specific role. The application assembler uses the `security-identity` element with a `use-caller-identity` child element to indicate that the current caller's identity should be propagated as the security identity for method invocations made by the EJB. Propagation of the caller's identity is the default used in the absence of an explicit `security-identity` element declaration.

Alternatively, the application assembler can use the `run-as/role-name` child element to specify that a specific security role given by the `role-name` value should be used as the security identity for method invocations made by the EJB. Note that this does not change the caller's identity as seen by the `EJBContext.getCallerPrincipal()` method. Rather, the caller's security roles are set to the single role specified by the `run-as/role-name` element value. One use case for the `run-as` element is to prevent external clients from accessing internal EJBs. You accomplish this by assigning the internal EJB `method-permission` elements that restrict access to a role never assigned to an external client. EJBs that need to use internal EJB are then configured with a `run-as/role-name` equal to the restricted role. The following descriptor fragment that illustrates `security-identity` element usage.

```
<!-- A sample ejb-jar.xml fragment -->
<ejb-jar>
    <enterprise-beans>
        <session>
            <ejb-name>ASessionBean</ejb-name>
            <!-- ... -->
            <security-identity>
                <use-caller-identity/>
```

```
      </security-identity>
    </session>
    <session>
      <ejb-name>RunAsBean</ejb-name>
      <!-- ... -->
      <security-identity>
        <run-as>
          <description>A private internal role</description>
          <role-name>InternalRole</role-name>
        </run-as>
      </security-identity>
    </session>
  </enterprise-beans>
  <!-- ... -->
</ejb-jar>
```

When you use `run-as` to assign a specific role to outgoing calls, JBoss associates a principal named `anonymous`. If you want another principal to be associated with the call, you need to associate a `run-as-principal` with the bean in the `jboss.xml` file. The following fragment associates a principal named `internal` with `RunAsBean` from the prior example.

```
<session>
  <ejb-name>RunAsBean</ejb-name>
  <security-identity>
    <run-as-principal>internal</run-as-principal>
  </security-identity>
</session>
```

The `run-as` element is also available in servlet definitions in a `web.xml` file. The following example shows how to assign the role `InternalRole` to a servlet:

```
<servlet>
  <servlet-name>AServlet</servlet-name>
  <!-- ... -->
  <run-as>
    <role-name>InternalRole</role-name>
  </run-as>
</servlet>
```

Calls from this servlet will be associated with the anonymous `principal`. The `run-as-principal` element is available in the `jboss-web.xml` file to assign a specific principal to go along with the `run-as` role. The following fragment shows how to associate a principal named `internal` to the servlet in the prior example.

```
<servlet>
    <servlet-name>AServlet</servlet-name>
    <run-as-principal>internal</run-as-principal>
</servlet>
```

1.3. Security roles

The security role name referenced by either the `security-role-ref` or `security-identity` element needs to map to one of the application's declared roles. An application assembler defines logical security roles by declaring `security-role` elements. The `role-name` value is a logical application role name like Administrator, Architect, SalesManager, etc.

The J2EE specifications note that it is important to keep in mind that the security roles in the deployment descriptor are used to define the logical security view of an application. Roles defined in the J2EE deployment descriptors should not be confused with the user groups, users, principals, and other concepts that exist in the target enterprise's operational environment. The deployment descriptor roles are application constructs with application domain-specific names. For example, a banking application might use role names such as BankManager, Teller, or Customer.

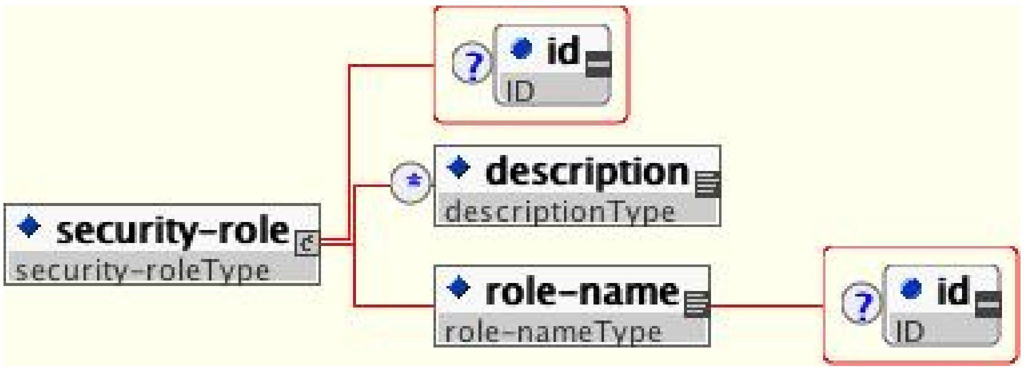

Figure 1.3. The security-role element

In JBoss, a `security-role` element is only used to map `security-role-ref/role-name` values to the logical role that the component role references. The user's assigned roles are a dynamic function of the application's security manager, as you will see when we discuss the JBossSX implementation details. JBoss does not require the definition of `security-role` elements in order to declare method permissions. However, the specification of `security-role` elements is still a recommended practice to ensure portability across application servers and for deployment descriptor maintenance. *Example 1.3, “An ejb-jar.xml descriptor fragment that illustrates the security-role element usage.”* shows the usage of the `security-role` in an `ejb-jar.xml` file.

Example 1.3. An ejb-jar.xml descriptor fragment that illustrates the security-role element usage.

```
<!-- A sample ejb-jar.xml fragment -->
<ejb-jar>
    <!-- ... -->
    <assembly-descriptor>
        <security-role>
            <description>The single application role</description>
            <role-name>TheApplicationRole</role-name>
        </security-role>
    </assembly-descriptor>
</ejb-jar>
```

Example 1.4, “An example web.xml descriptor fragment that illustrates the security-role element usage.” shows the usage of the `security-role` in an `web.xml` file.

Example 1.4. An example web.xml descriptor fragment that illustrates the security-role element usage.

```
<!-- A sample web.xml fragment -->
<web-app>
    <!-- ... -->
    <security-role>
        <description>The single application role</description>
        <role-name>TheApplicationRole</role-name>
    </security-role>
</web-app>
```

1.4. EJB method permissions

An application assembler can set the roles that are allowed to invoke an EJB's home and remote interface methods through method-permission element declarations.

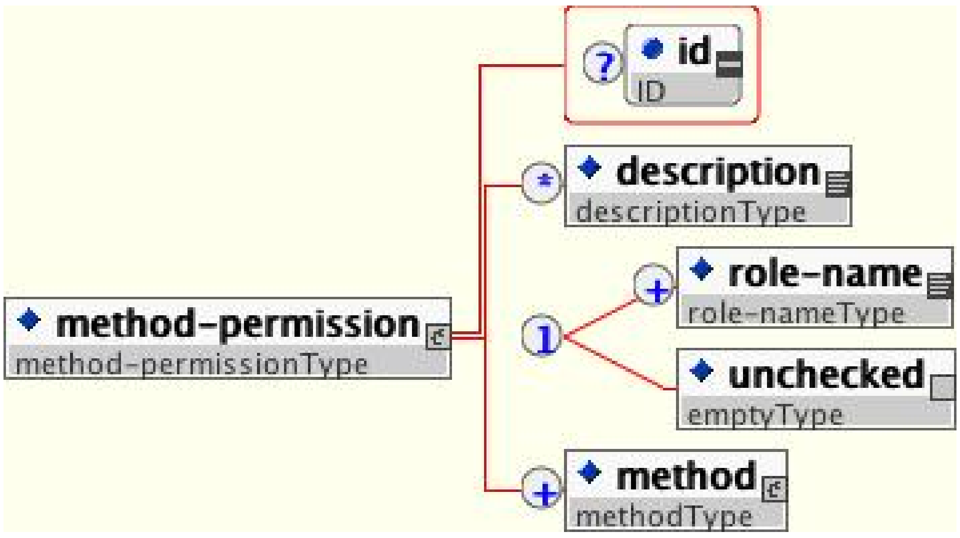

Figure 1.4. The method-permissions element

Each `method-permission` element contains one or more role-name child elements that define the logical roles that are allowed to access the EJB methods as identified by method child elements. You can also specify an `unchecked` element instead of the `role-name` element to declare that any authenticated user can access the methods identified by method child elements. In addition, you can declare that no one should have access to a method that has the `exclude-list` element. If an EJB has methods that have not been declared as accessible by a role using a `method-permission` element, the EJB methods default to being excluded from use. This is equivalent to defaulting the methods into the `exclude-list`.

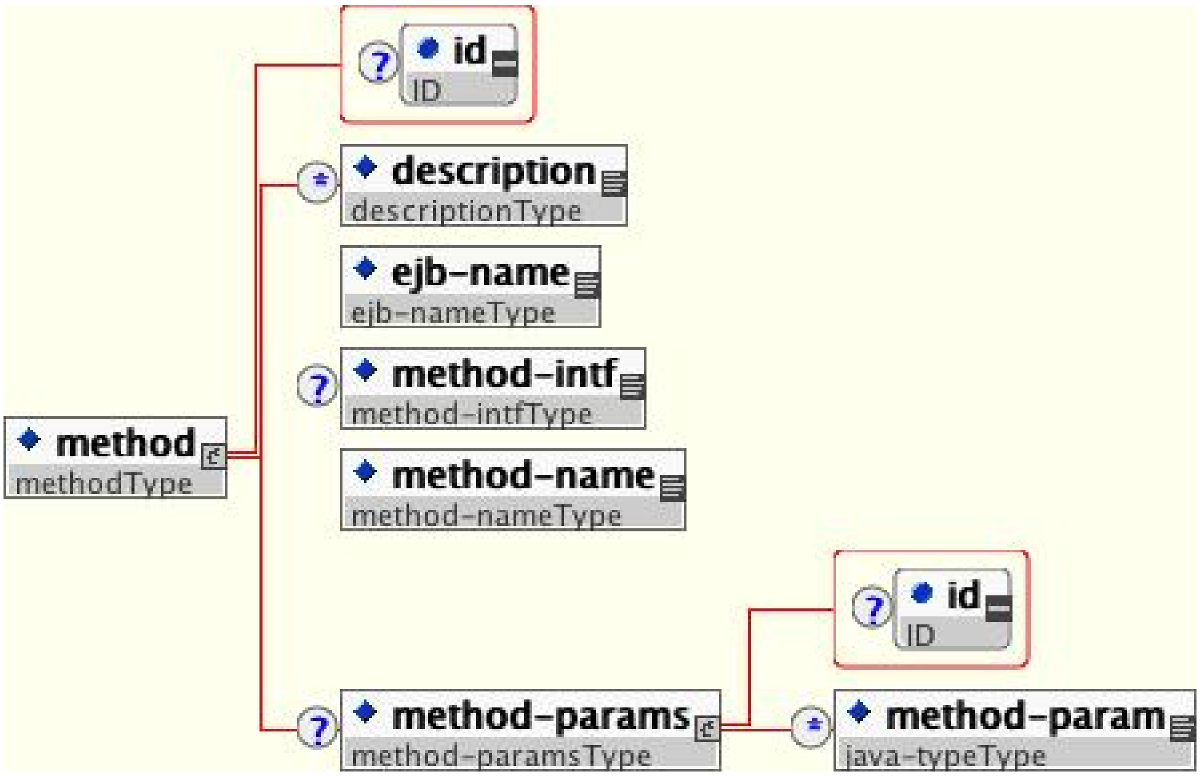

Figure 1.5. The method element

There are three supported styles of method element declarations.

The first is used for referring to all the home and component interface methods of the named enterprise bean:

```
<method>
    <ejb-name>EJBNAME</ejb-name>
    <method-name>*</method-name>
</method>
```

The second style is used for referring to a specified method of the home or component interface of the named enterprise bean:

```
<method>
    <ejb-name>EJBNAME</ejb-name>
    <method-name>METHOD</method-name>
                </method>
```

If there are multiple methods with the same overloaded name, this style refers to all of the overloaded methods.

The third style is used to refer to a specified method within a set of methods with an overloaded name:

```
<method>
  <ejb-name>EJBNAME</ejb-name>
  <method-name>METHOD</method-name>
  <method-params>
    <method-param>PARAMETER_1</method-param>
    <!-- ... -->
    <method-param>PARAMETER_N</method-param>
  </method-params>
</method>
```

The method must be defined in the specified enterprise bean's home or remote interface. The method-param element values are the fully qualified name of the corresponding method parameter type. If there are multiple methods with the same overloaded signature, the permission applies to all of the matching overloaded methods.

The optional `method-intf` element can be used to differentiate methods with the same name and signature that are defined in both the home and remote interfaces of an enterprise bean.

Example 1.5, “An ejb-jar.xml descriptor fragment that illustrates the method-permission element usage.” provides complete examples of the `method-permission` element usage.

Example 1.5. An ejb-jar.xml descriptor fragment that illustrates the method-permission element usage.

```
<ejb-jar>
  <assembly-descriptor>
    <method-permission>
      <description>The employee and temp-employee roles may access any
        method of the EmployeeService bean </description>
      <role-name>employee</role-name>
      <role-name>temp-employee</role-name>
      <method>
        <ejb-name>EmployeeService</ejb-name>
        <method-name>*</method-name>
      </method>
    </method-permission>
    <method-permission>
```

```
        <description>The employee role may access the findByPrimaryKey,
            getEmployeeInfo, and the updateEmployeeInfo(String) method of
            the AardvarkPayroll bean </description>
        <role-name>employee</role-name>
        <method>
            <ejb-name>AardvarkPayroll</ejb-name>
            <method-name>findByPrimaryKey</method-name>
        </method>
        <method>
            <ejb-name>AardvarkPayroll</ejb-name>
            <method-name>getEmployeeInfo</method-name>
        </method>
        <method>
            <ejb-name>AardvarkPayroll</ejb-name>
            <method-name>updateEmployeeInfo</method-name>
            <method-params>
                <method-param>java.lang.String</method-param>
            </method-params>
        </method>
    </method-permission>
    <method-permission>
        <description>The admin role may access any method of the
            EmployeeServiceAdmin bean </description>
        <role-name>admin</role-name>
        <method>
            <ejb-name>EmployeeServiceAdmin</ejb-name>
            <method-name>*</method-name>
        </method>
    </method-permission>
    <method-permission>
        <description>Any authenticated user may access any method of the
            EmployeeServiceHelp bean</description>
        <unchecked/>
        <method>
            <ejb-name>EmployeeServiceHelp</ejb-name>
            <method-name>*</method-name>
        </method>
    </method-permission>
    <exclude-list>
        <description>No fireTheCTO methods of the EmployeeFiring bean may be
            used in this deployment</description>
        <method>
            <ejb-name>EmployeeFiring</ejb-name>
            <method-name>fireTheCTO</method-name>
            </method>
        </exclude-list>
    </assembly-descriptor>
</ejb-jar>
```

1.5. Web Content Security Constraints

In a web application, security is defined by the roles that are allowed access to content by a URL pattern that identifies the protected content. This set of information is declared by using the `web.xmlsecurity-constraint` element.

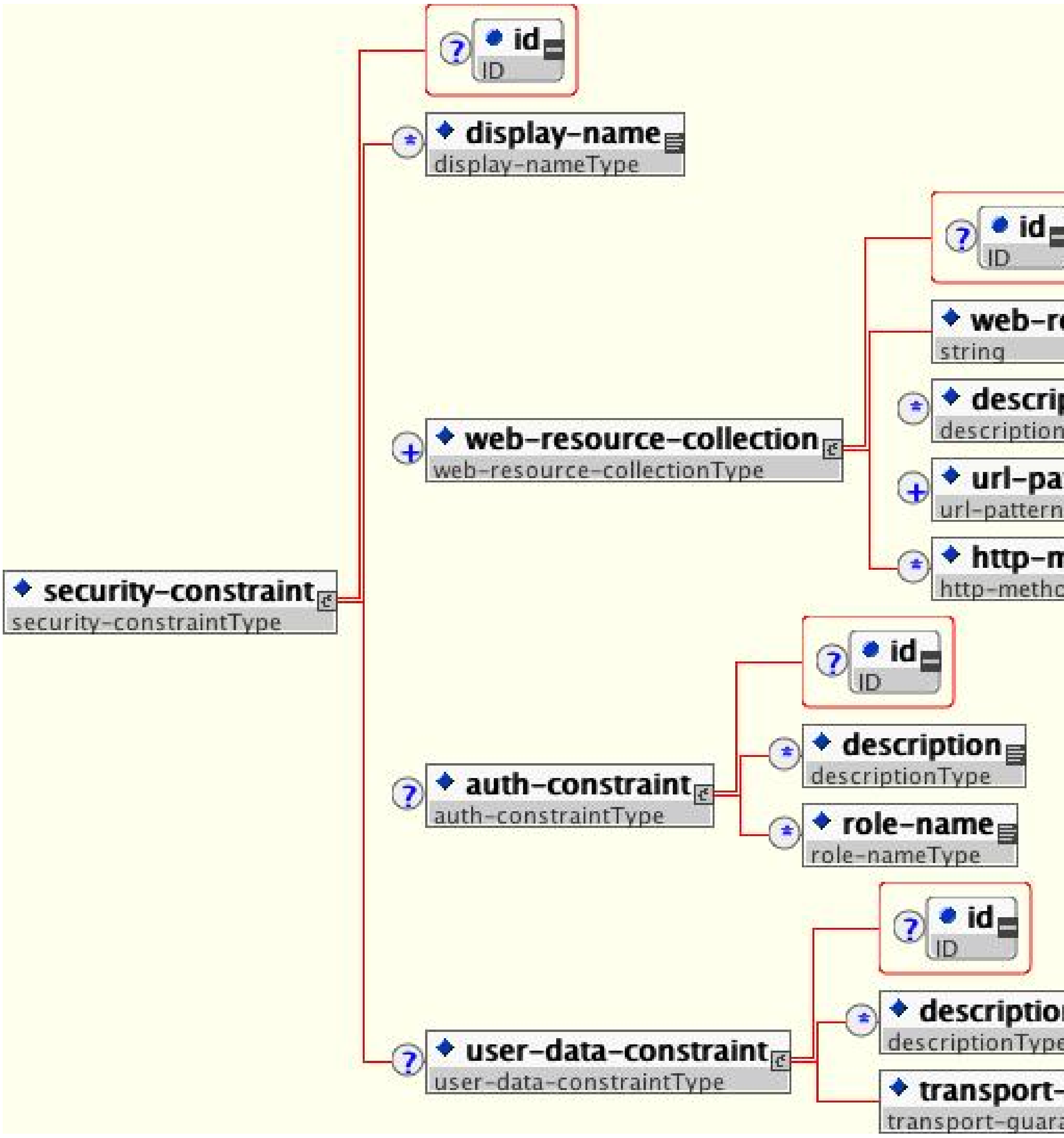

Figure 1.6. The security-constraint element

The content to be secured is declared using one or more `web-resource-collection` elements. Each `web-resource-collection` element contains an optional series of `url-pattern` elements followed by an optional series of `http-method` elements. The `url-pattern` element value specifies a URL pattern against which a request URL must match for the request to correspond to an attempt to access secured content. The `http-method` element value specifies a type of HTTP request to allow.

The optional `user-data-constraint` element specifies the requirements for the transport layer of the client to server connection. The requirement may be for content integrity (preventing data tampering in the communication process) or for confidentiality (preventing reading while in transit). The transport-guarantee element value specifies the degree to which communication between the client and server should be protected. Its values are `NONE`, `INTEGRAL`, and `CONFIDENTIAL`. A value of `NONE` means that the application does not require any transport guarantees. A value of `INTEGRAL` means that the application requires the data sent between the client and server to be sent in such a way that it can't be changed in transit. A value of `CONFIDENTIAL` means that the application requires the data to be transmitted in a fashion that prevents other entities from observing the contents of the transmission. In most cases, the presence of the `INTEGRAL` or `CONFIDENTIAL` flag indicates that the use of SSL is required.

The optional `login-config` element is used to configure the authentication method that should be used, the realm name that should be used for rhw application, and the attributes that are needed by the form login mechanism.

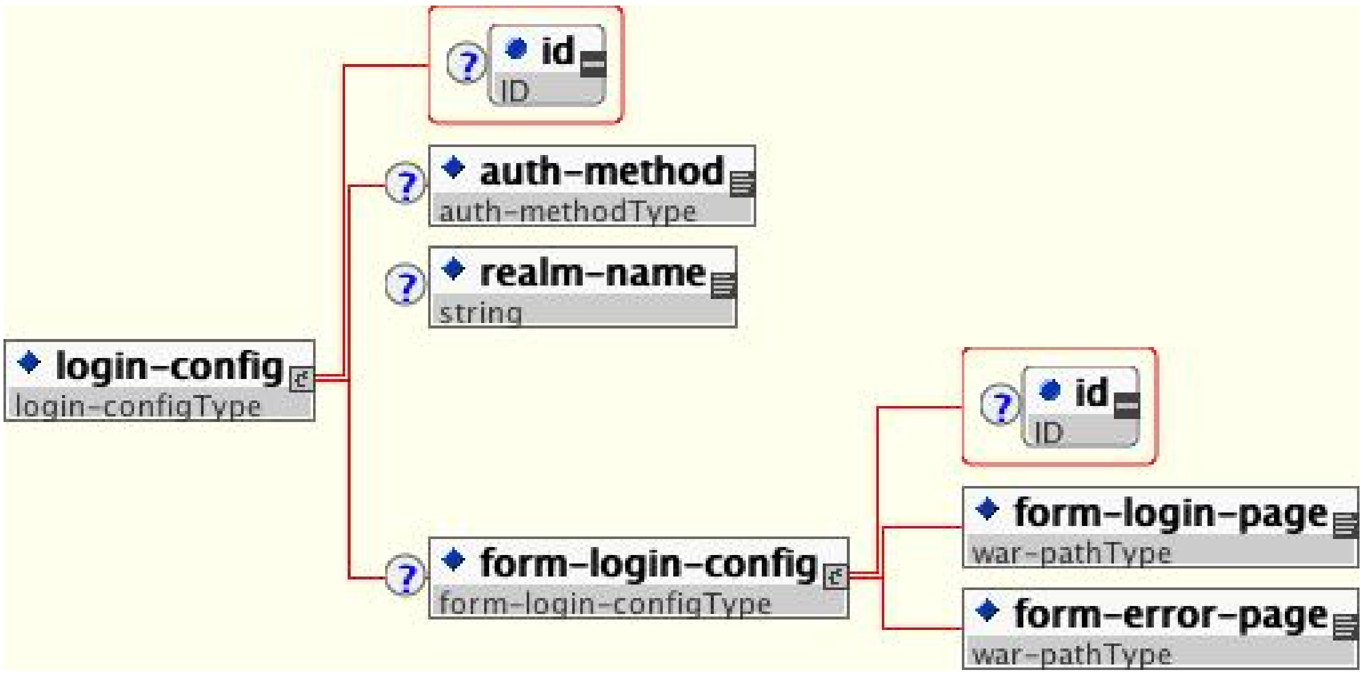

Figure 1.7. The login-config element

The `auth-method` child element specifies the authentication mechanism for the web application. As a prerequisite to gaining access to any web resources that are protected by an authorization constraint, a user must have authenticated using the configured mechanism. Legal `auth-method` values are `BASIC`, `DIGEST`, `FORM`, and `CLIENT-CERT`. The `realm-name` child element specifies the realm name to use in HTTP basic and digest authorization. The `form-login-config` child element specifies the log in as well as error pages that should be used in form-based login. If the `auth-method` value is not `FORM`, then `form-login-config` and its child elements are ignored.

As an example, the `web.xml` descriptor fragment given in *Example 1.6, “ A web.xml descriptor fragment which illustrates the use of the security-constraint and related elements.”* indicates that any URL lying under the web application's `/restricted` path requires an `AuthorizedUser` role. There is no required transport guarantee and the authentication method used for obtaining the user identity is BASIC HTTP authentication.

Example 1.6. A web.xml descriptor fragment which illustrates the use of the security-constraint and related elements.

```
<web-app>
    <!-- ... -->
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>Secure Content</web-resource-name>
            <url-pattern>/restricted/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>AuthorizedUser</role-name>
        </auth-constraint>
        <user-data-constraint>
            <transport-guarantee>NONE</transport-guarantee>
        </user-data-constraint>
    </security-constraint>
    <!-- ... -->
    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>The Restricted Zone</realm-name>
    </login-config>
    <!-- ... -->
    <security-role>
        <description>The role required to access restricted content </description>
        <role-name>AuthorizedUser</role-name>
    </security-role>
</web-app>
```

1.6. Enabling Declarative Security in JBoss

The J2EE security elements that have been covered so far describe the security requirements only from the application's perspective. Because J2EE security elements declare logical roles, the application deployer maps the roles from the application domain onto the deployment environment. The J2EE specifications omit these application server-specific details. In JBoss, mapping the application roles onto the deployment environment entails specifying a security manager that implements the J2EE security model using JBoss server specific deployment descriptors. The details behind the security configuration are discussed in *Chapter 3, JBoss Security Model*.

Introduction to JAAS

The JBossSX framework is based on the JAAS API. It is important that you understand the basic elements of the JAAS API to understand the implementation details of JBossSX. The following sections provide an introduction to JAAS to prepare you for the JBossSX architecture discussion later in this chapter.

The JAAS 1.0 API consists of a set of Java packages designed for user authentication and authorization. It implements a Java version of the standard *Pluggable Authentication Module* (PAM) framework and compatibly extends the Java 2 Platform's access control architecture to support user-based authorization. JAAS was first released as an extension package for JDK 1.3 and is bundled with JDK 1.4+. Because the JBossSX framework uses only the authentication capabilities of JAAS to implement the declarative role-based J2EE security model, this introduction focuses on only that topic.

JAAS authentication is performed in a pluggable fashion. This permits Java applications to remain independent from underlying authentication technologies and allows the JBossSX security manager to work in different security infrastructures. Integration with a security infrastructure can be achieved without changing the JBossSX security manager implementation. All that needs to change is the configuration of the authentication stack that JAAS uses.

2.1. The JAAS Core Classes

The JAAS core classes can be broken down into three categories: common, authentication, and authorization. The following list presents only the common and authentication classes because these are the specific classes used to implement the functionality of JBossSX covered in this chapter.

The are the common classes:

- `Subject` (`javax.security.auth.Subject`)
- `Principal` (`java.security.Principal`)

These are the authentication classes:

- `Callback` (`javax.security.auth.callback.Callback`)
- `CallbackHandler` (`javax.security.auth.callback.CallbackHandler`)
- `Configuration` (`javax.security.auth.login.Configuration`)
- `LoginContext` (`javax.security.auth.login.LoginContext`)
- `LoginModule` (`javax.security.auth.spi.LoginModule`)

2.1.1. The Subject and Principal Classes

To authorize access to resources, applications first need to authenticate the request's source. The JAAS framework defines the term subject to represent a request's source. The `Subject` class is the central class in JAAS. A `Subject` represents information for a single entity, such as a person or service. It encompasses the entity's principals, public credentials, and private credentials. The JAAS APIs use the existing Java 2 `java.security.Principal` interface to represent a principal, which is essentially just a typed name.

During the authentication process, a subject is populated with associated identities, or principals. A subject may have many principals. For example, a person may have a name principal (John Doe), a social security number principal (123-45-6789), and a username principal (johnd), all of which help distinguish the subject from other subjects. To retrieve the principals associated with a subject, two methods are available:

```
public Set getPrincipals() {...}
public Set getPrincipals(Class c) {...}
```

The first method returns all principals contained in the subject. The second method returns only those principals that are instances of class `c` or one of its subclasses. An empty set is returned if the subject has no matching principals. Note that the `java.security.acl.Group` interface is a subinterface of `java.security.Principal`, so an instance in the principals set may represent a logical grouping of other principals or groups of principals.

2.1.2. Authentication of a Subject

Authentication of a subject requires a JAAS login. The login procedure consists of the following steps:

1. An application instantiates a `LoginContext` and passes in the name of the login configuration and a `CallbackHandler` to populate the `Callback` objects, as required by the configuration `LoginModules`.

2. The `LoginContext` consults a `Configuration` to load all the `LoginModules` included in the named login configuration. If no such named configuration exists the `other` configuration is used as a default.

3. The application invokes the `LoginContext.login` method.

4. The login method invokes all the loaded `LoginModules`. As each `LoginModule` attempts to authenticate the subject, it invokes the handle method on the associated `CallbackHandler` to obtain the information required for the authentication process. The required information is passed to the handle method in the form of an array of `Callback` objects. Upon success, the `LoginModules` associate relevant principals and credentials with the subject.

5. The `LoginContext` returns the authentication status to the application. Success is represented by a return from the login method. Failure is represented through a LoginException being thrown by the login method.

6. If authentication succeeds, the application retrieves the authenticated subject using the `LoginContext.getSubject` method.
7. After the scope of the subject authentication is complete, all principals and related information associated with the subject by the login method can be removed by invoking the `LoginContext.logout` method.

The `LoginContext` class provides the basic methods for authenticating subjects and offers a way to develop an application that is independent of the underlying authentication technology. The `LoginContext` consults a `Configuration` to determine the authentication services configured for a particular application. `LoginModule` classes represent the authentication services. Therefore, you can plug different login modules into an application without changing the application itself. The following code shows the steps required by an application to authenticate a subject.

```
CallbackHandler handler = new MyHandler();
LoginContext lc = new LoginContext("some-config", handler);

try {
    lc.login();
    Subject subject = lc.getSubject();
} catch(LoginException e) {
    System.out.println("authentication failed");
    e.printStackTrace();
}

// Perform work as authenticated Subject
// ...

// Scope of work complete, logout to remove authentication info
try {
    lc.logout();
} catch(LoginException e) {
    System.out.println("logout failed");
    e.printStackTrace();
}

// A sample MyHandler class
class MyHandler implements CallbackHandler
{
    public void handle(Callback[] callbacks) throws
        IOException, UnsupportedCallbackException
    {
        for (int i = 0; i < callbacks.length; i++) {
            if (callbacks[i] instanceof NameCallback) {
                NameCallback nc = (NameCallback)callbacks[i];
                nc.setName(username);
            } else if (callbacks[i] instanceof PasswordCallback) {
                PasswordCallback pc = (PasswordCallback)callbacks[i];
                pc.setPassword(password);
            } else {
                throw new UnsupportedCallbackException(callbacks[i],
                                                       "Unrecognized Callback");
            }
        }
    }
}
```

Developers integrate with an authentication technology by creating an implementation of the `LoginModule` interface. This allows an administrator to plug different authentication technologies into an application. You can chain together multiple `LoginModule`s to allow for more than one authentication technology to participate in the authentication process. For example, one `LoginModule` may perform username/password-based authentication, while another may interface to hardware devices such as smart card readers or biometric authenticators.

The life cycle of a `LoginModule` is driven by the `LoginContext` object against which the client creates and issues the login method. The process consists of two phases. The steps of the process are as follows:

- The `LoginContext` creates each configured `LoginModule` using its public no-arg constructor.
- Each `LoginModule` is initialized with a call to its initialize method. The `Subject` argument is guaranteed to be non-null. The signature of the initialize method is: `public void initialize(Subject subject, CallbackHandler callbackHandler, Map sharedState, Map options)`.
- The `login` method is called to start the authentication process. For example, a method implementation might prompt the user for a username and password and then verify the information against data stored in a naming service such as NIS or LDAP. Alternative implementations might interface to smart cards and biometric devices, or simply extract user information from the underlying operating system. The validation of user identity by each `LoginModule` is considered phase 1 of JAAS authentication. The signature of the `login` method is `boolean login() throws LoginException`. A `LoginException` indicates failure. A return value of true indicates that the method succeeded, whereas a return valueof false indicates that the login module should be ignored.
- If the `LoginContext`'s overall authentication succeeds, `commit` is invoked on each `LoginModule`. If phase 1 succeeds for a `LoginModule`, then the commit method continues with phase 2 and associates the relevant principals, public credentials, and/or private credentials with the subject. If phase 1 fails for a `LoginModule`, then `commit` removes any previously stored

authentication state, such as usernames or passwords. The signature of the `commit` method is: `boolean commit() throws LoginException`. Failure to complete the commit phase is indicated by throwing a `LoginException`. A return of true indicates that the method succeeded, whereas a return of false indicates that the login module should be ignored.

- If the `LoginContext`'s overall authentication fails, then the `abort` method is invoked on each `LoginModule`. The `abort` method removes or destroys any authentication state created by the login or initialize methods. The signature of the `abort` method is `boolean abort() throws LoginException`. Failure to complete the `abort` phase is indicated by throwing a `LoginException`. A return of true indicates that the method succeeded, whereas a return of false indicates that the login module should be ignored.
- To remove the authentication state after a successful login, the application invokes `logout` on the `LoginContext`. This in turn results in a `logout` method invocation on each `LoginModule`. The `logout` method removes the principals and credentials originally associated with the subject during the `commit` operation. Credentials should be destroyed upon removal. The signature of the `logout` method is: `boolean logout() throws LoginException`. Failure to complete the logout process is indicated by throwing a `LoginException`. A return of true indicates that the method succeeded, whereas a return of false indicates that the login module should be ignored.

When a `LoginModule` must communicate with the user to obtain authentication information, it uses a `CallbackHandler` object. Applications implement the `CallbackHandler` interface and pass it to the LoginContext, which forwards it directly to the underlying login modules. Login modules use the `CallbackHandler` both to gather input from users, such as a password or smart card PIN, and to supply information to users, such as status information. By allowing the application to specify the `CallbackHandler`, underlying `LoginModules` remain independent from the different ways applications interact with users. For example, a `CallbackHandler`'s implementation for a GUI application might display a window to solicit user input. On the other hand, a `callbackhandler`'s implementation for a non-GUI environment, such as an application server, might simply obtain credential information by using an application server API. The `callbackhandler` interface has one method to implement:

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

The `Callback` interface is the last authentication class we will look at. This is a tagging interface for which several default implementations are provided, including the `NameCallback` and `PasswordCallback` used in an earlier example. A `LoginModule` uses a `Callback` to request information required by the authentication mechanism. `LoginModules` pass an array of `Callbacks` directly to the `CallbackHandler.handle` method during the authentication's login phase. If a `callbackhandler` does not understand how to use a `Callback` object passed into the handle method, it throws an `UnsupportedCallbackException` to abort the login call.

JBoss Security Model

Similar to the rest of the JBoss architecture, security at the lowest level is defined as a set of interfaces for which alternate implementations may be provided. The following interfaces define the JBoss server security layer:

- **org.jboss.security.AuthenticationManager**
- **org.jboss.security.RealmMapping**
- **org.jboss.security.SecurityProxy**
- **org.jboss.security.AuthorizationManager**
- **org.jboss.security.AuditManager**
- **org.jboss.security.MappingManager**

Figure 3.1, “The key security model interfaces and their relationship to the JBoss server EJB container elements.” shows a class diagram of the security interfaces and their relationship to the EJB container architecture.

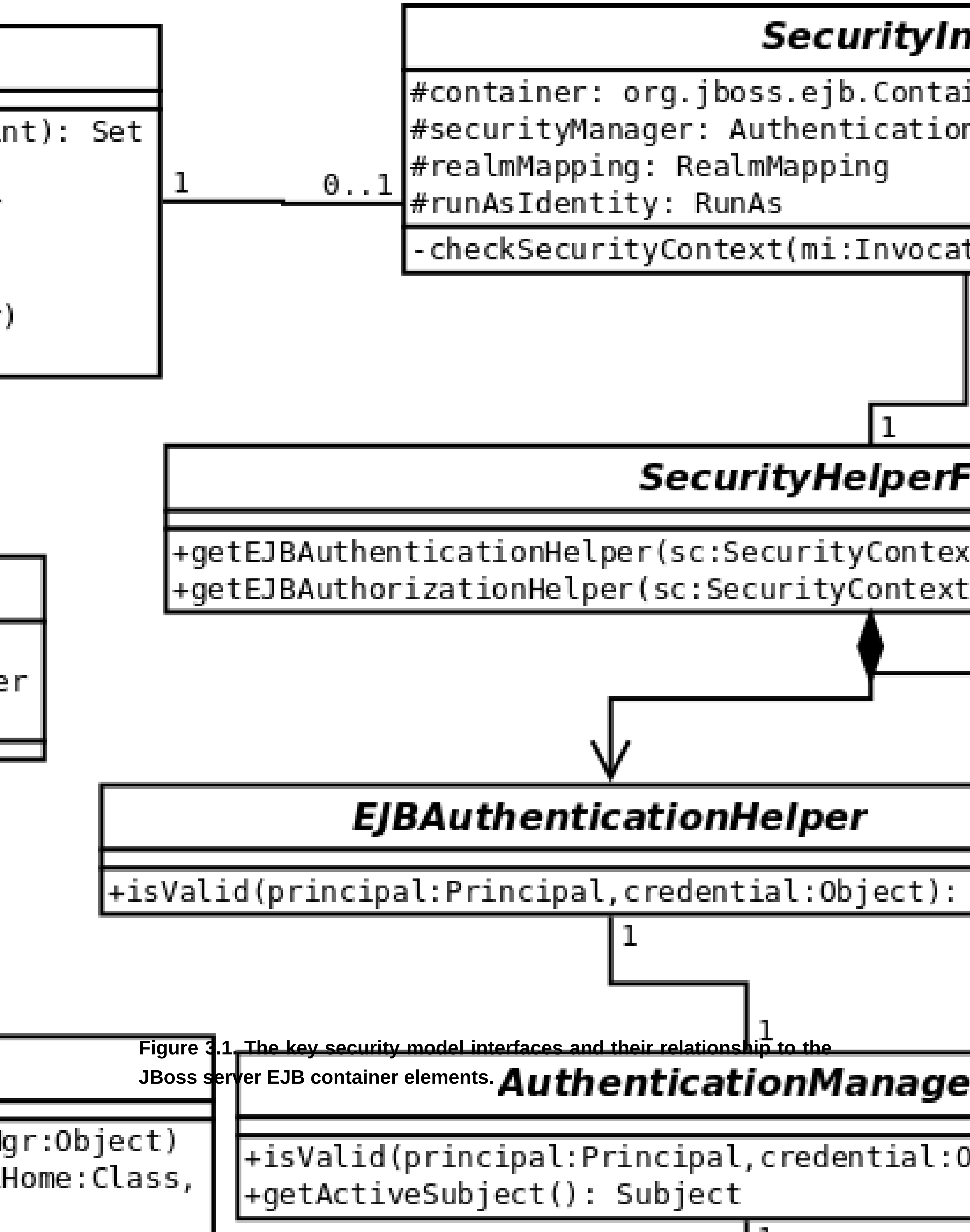
SecurityIn
#container: org.jboss.ejb.Contai
#securityManager: Authenticatio
#realmMapping: RealmMapping
#runAsIdentity: RunAs
-checkSecurityContext(mi:Invocat
nt): Set
1
0..1
1
SecurityHelperF
+getEJBAuthenticationHelper(sc:SecurityContex
+getEJBAuthorizationHelper(sc:SecurityContext
er
EJBAuthenticationHelper
+isValid(principal:Principal,credential:Object):
1
1
AuthenticationManage
+isValid(principal:Principal,credential:O
+getActiveSubject(): Subject
gr:Object)
Home:Class,

Figure 3.1. The key security model interfaces and their relationship to the JBoss server EJB container elements.

The EJB Container layer is represented by the classes `org.jboss.ejb.Container`, `org.jboss.SecurityInterceptor` and `org.jboss.SecurityProxyInterceptor`. The other classes are interfaces and classes provided by the JBoss security subsystem.

The two interfaces required for the J2EE security model implementation are:

- **org.jboss.security.AuthenticationManager**
- **org.jboss.security.AuthorizationManager**

The roles of the security interfaces presented in *Figure 3.1, “The key security model interfaces and their relationship to the JBoss server EJB container elements.”* are summarized below.

- **AuthenticationManager**: This interface is responsible for validating credentials associated with *Principals*. Principals are identities, such as usernames, employee numbers, and social security numbers. *Credentials* are proof of the identity, such as passwords, session keys, and digital signatures. The `isValid` method is invoked to determine whether a user identity and associated credentials as known in the operational environment are valid proof of the user's identity.
- **AuthorizationManager**: This interface is responsible for the access control mandated by the J2EE specifications. The implementation of this interface provides the ability to stack a set of Policy Providers useful for pluggable authorization.
- **SecurityProxy**: This interface describes the requirements for a custom `SecurityProxyInterceptor` plugin. A `SecurityProxy` allows for the externalization of custom security checks on a per-method basis for both the EJB home and remote interface methods.
- **AuditManager**: This interface is responsible for providing an audit trail of security events.
- **MappingManager**: This interface is responsible for providing mapping of Principal, Role, and Attributes. The implementation of AuthorizationManager may internally call the mapping manager to map roles before performing access control.
- **RealMapping**: This interface is responsible for principal mapping and role mapping. The `getPrincipal` method takes an user identity as known in the operational environment and returns the application domain identity. The `doesUserHaveRole` method validates that the user identity in the operation environment has been assigned the indicated role from the application domain.

Note that the `AuthenticationManager`, `RealmMapping` and `SecurityProxy` interfaces have no association to JAAS related classes. Although the JBossSX framework is heavily dependent on JAAS, the basic security interfaces required for implementation of the J2EE security model are not. The JBossSX framework is simply an implementation of the basic security plugin interfaces that are based on JAAS.

The component diagram in *Figure 3.2, “The relationship between the JBossSX framework implementation classes and the JBoss server EJB container layer.”* illustrates this fact. The implication of this plug-in architecture is that you are free to replace the JAAS-based JBossSX implementation classes with your own non-JAAS custom security manager implementation. You'll see how to do this when you look at the JBossSX MBeans available for JBossSX configuration in *Figure 3.2, “The relationship between the JBossSX framework implementation classes and the JBoss server EJB container layer.”*.

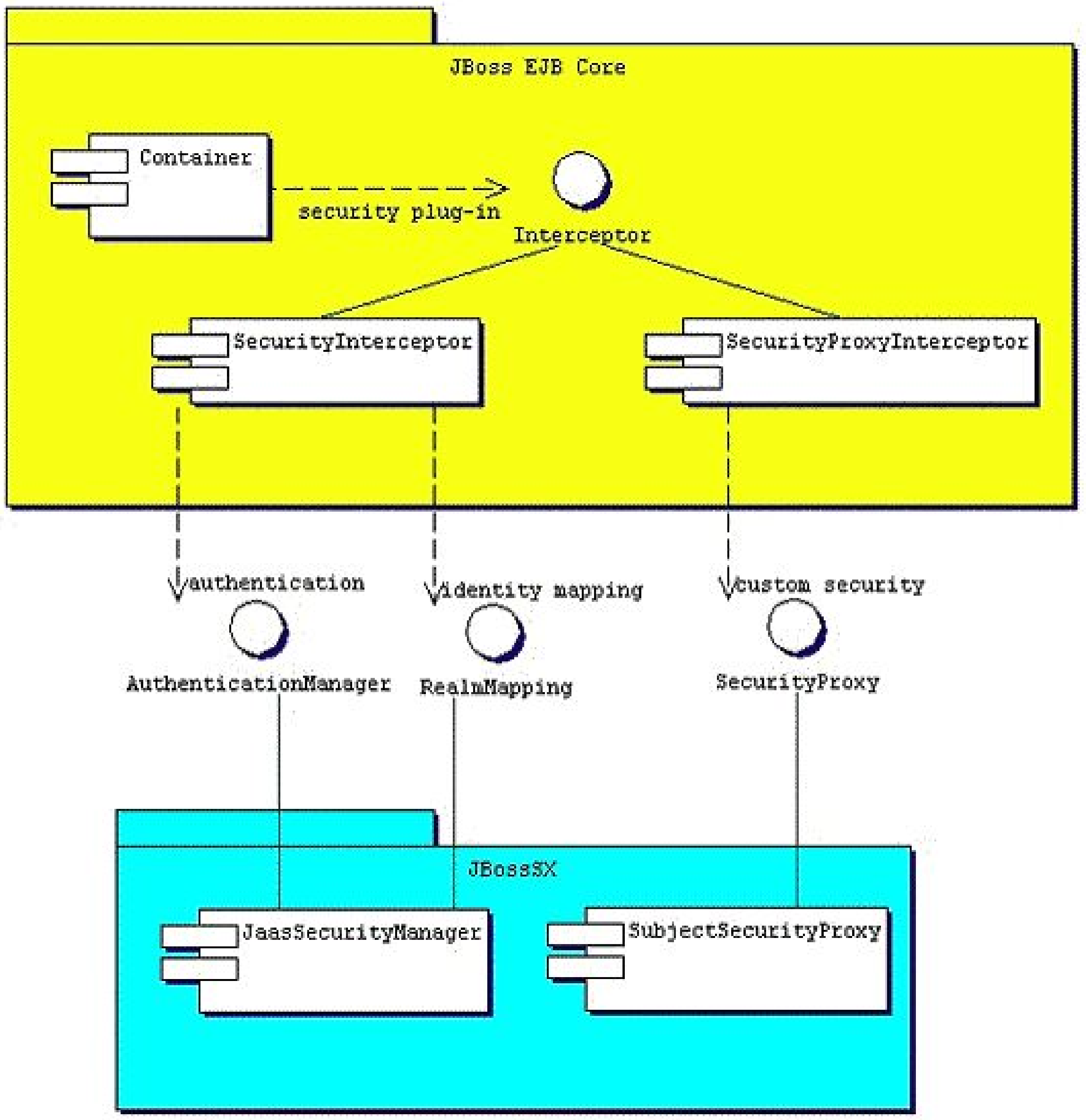

Figure 3.2. The relationship between the JBossSX framework implementation classes and the JBoss server EJB container layer.

3.1. Enabling Declarative Security in JBoss Revisited

Earlier in this chapter, the discussion of the J2EE standard security model ended with a requirement for the use of JBoss server-specific deployment descriptor to enable security. The details of this configuration are presented here. *Figure 3.3, “The security element subsets of the JBoss server jboss.xml and jboss-web.xml deployment descriptors.”* shows the JBoss-specific EJB and web application deployment descriptor's security-related elements.

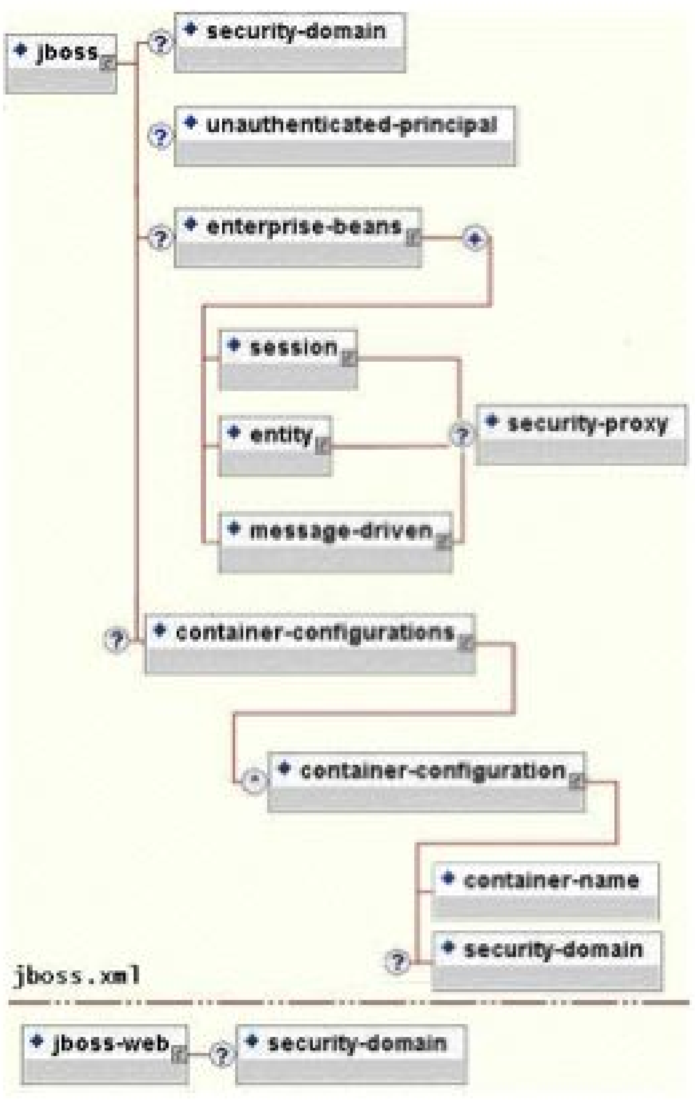

Figure 3.3. The security element subsets of the JBoss server jboss.xml and jboss-web.xml deployment descriptors.

The value of a `security-domain` element specifies the JNDI name of the security manager interface implementation that JBoss uses for the EJB and web containers. This is an object that implements both of the `AuthenticationManager` and `RealmMapping` interfaces. When specified as a top-level element it defines what security domain in effect for all EJBs in the deployment unit.

This is the typical usage because mixing security managers within a deployment unit complicates inter-component operation and administration.

To specify the security domain for an individual EJB, you specify the `security-domain` at the container configuration level. This will override any top-level security-domain element.

The `unauthenticated-principal` element specifies the name to use for the `Principal` object returned by the `EJBContext.getUserPrincipal` method when an unauthenticated user invokes an EJB. Note that this conveys no special permissions to an unauthenticated caller. Its primary purpose is to allow unsecured servlets and JSP pages to invoke unsecured EJBs and allow the target EJB to obtain a non-null `Principal` for the caller using the `getUserPrincipal` method. This is a J2EE specification requirement.

The `security-proxy` element identifies a custom security proxy implementation that allows per-request security checks outside the scope of the EJB declarative security model without embedding security logic into the EJB implementation. This may be an implementation of the `org.jboss.security.SecurityProxy` interface, or just an object that implements methods in the home, remote, local home or local interfaces of the EJB to secure without implementing any common interface. If the given class does not implement the `SecurityProxy` interface, the instance must be wrapped in a `SecurityProxy` implementation that delegates the method invocations to the object. The `org.jboss.security.SubjectSecurityProxy` is an example `SecurityProxy` implementation used by the default JBossSX installation.

Take a look at a simple example of a custom `SecurityProxy` in the context of a trivial stateless session bean. The custom `SecurityProxy` validates that no one invokes the bean's `echo` method with a four-letter word as its argument. This is a check that is not possible with role-based security; you cannot define a `FourLetterEchoInvoker` role because the security context is the method argument, not a property of the caller. The code for the custom `SecurityProxy` is given in *Example 3.1, “The example 1 custom EchoSecurityProxy implementation that enforces the echo argument-based security constraint.”*.

Example 3.1. The example 1 custom EchoSecurityProxy implementation that enforces the echo argument-based security constraint.

```
package org.jboss.book.security.ex1;

import java.lang.reflect.Method;
import javax.ejb.EJBContext;

import org.apache.log4j.Category;

import org.jboss.security.SecurityProxy;

/** A simple example of a custom SecurityProxy implementation
 *  that demonstrates method argument based security checks.
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.4 $
 */
public class EchoSecurityProxy implements SecurityProxy
{
    Category log = Category.getInstance(EchoSecurityProxy.class);
    Method echo;

    public void init(Class beanHome, Class beanRemote,
                     Object securityMgr)
        throws InstantiationException
    {
        log.debug("init, beanHome="+beanHome
                  + ", beanRemote="+beanRemote
                  + ", securityMgr="+securityMgr);
        // Get the echo method for equality testing in invoke
        try {
            Class[] params = {String.class};
            echo = beanRemote.getDeclaredMethod("echo", params);
        } catch(Exception e) {
            String msg = "Failed to finde an echo(String) method";
            log.error(msg, e);
            throw new InstantiationException(msg);
        }
    }

    public void setEJBContext(EJBContext ctx)
    {
        log.debug("setEJBContext, ctx="+ctx);
    }

    public void invokeHome(Method m, Object[] args)
        throws SecurityException
    {
        // We don't validate access to home methods
    }

    public void invoke(Method m, Object[] args, Object bean)
        throws SecurityException
    {
        log.debug"invoke, m="+m);
        // Check for the echo method
        if (m.equals(echo)) {
            // Validate that the msg arg is not 4 letter word
            String arg = (String) args[0];
            if (arg == null || arg.length() == 4)
                throw new SecurityException("No 4 letter words");
        }
        // We are not responsible for doing the invoke
    }
}
```

The `EchoSecurityProxy` checks that the method to be invoked on the bean instance corresponds to the `echo(String)` method loaded the init method. If there is a match, the method argument is obtained and its length compared against 4 or null. Either case results in a `SecurityException` being thrown. Certainly this is a contrived example, but only in its application. It is a common requirement that applications must perform security checks based on the value of method arguments. The point of the example is to demonstrate how custom security beyond the scope of the standard declarative security model can be introduced independent of the bean implementation. This allows the specification and coding of the security requirements to be delegated to security experts. Since the security proxy layer can be done independent of the bean implementation, security can be changed to match the deployment environment requirements.

The associated `jboss.xml` descriptor that installs the `EchoSecurityProxy` as the custom proxy for the `EchoBean` is given in *Example 3.2, “The jboss.xml descriptor, which configures the EchoSecurityProxy as the custom security proxy for the EchoBean.”*.

Example 3.2. The jboss.xml descriptor, which configures the EchoSecurityProxy as the custom security proxy for the EchoBean.

```
<jboss>
    <security-domain>other</security-domain>

    <enterprise-beans>
        <session>
            <ejb-name>EchoBean</ejb-name>
            <security-proxy>org.jboss.book.security.ex1.EchoSecurityProxy</security-proxy>
        </session>
    </enterprise-beans>
</jboss>
```

Now test the custom proxy by running a client that attempts to invoke the `EchoBean.echo` method with the arguments `Hello` and `Four` as illustrated in this fragment:

```
public class ExClient
{
    public static void main(String args[])
        throws Exception
    {
        Logger log = Logger.getLogger("ExClient");
        log.info("Looking up EchoBean");

        InitialContext iniCtx = new InitialContext();
        Object ref = iniCtx.lookup("EchoBean");
        EchoHome home = (EchoHome) ref;
        Echo echo = home.create();

        log.info("Created Echo");
        log.info("Echo.echo('Hello') = "+echo.echo("Hello="));
        log.info("Echo.echo('Four') = "+echo.echo("Four"));
    }
}
```

The first call should succeed, while the second should fail due to the fact that `Four` is a four-letter word. Run the client as follows using Ant from the examples directory:

```
[examples]$ ant -Dchap=security -Dex=1 run-example
run-example1:
...
     [echo] Waiting for 5 seconds for deploy...
     [java] [INFO,ExClient] Looking up EchoBean
     [java] [INFO,ExClient] Created Echo
     [java] [INFO,ExClient] Echo.echo('Hello') = Hello
     [java] Exception in thread "main" java.rmi.AccessException: SecurityException; nested
 exception is:
     [java]     java.lang.SecurityException: No 4 letter words
...
     [java] Caused by: java.lang.SecurityException: No 4 letter words
...
```

The result is that the `echo('Hello')` method call succeeds as expected and the `echo('Four')` method call results in a rather messy looking exception, which is also expected. The above output has been truncated to fit in the book. The key part to the exception is that the

`SecurityException("No 4 letter words")` generated by the `EchoSecurityProxy` was thrown to abort the attempted method invocation as desired.

The JBoss Security Extension Architecture

The preceding discussion of the general JBoss security layer has stated that the JBossSX security extension framework is an implementation of the security layer interfaces. This is the primary purpose of the JBossSX framework. The details of the implementation are interesting in that it offers a great deal of customization for integration into existing security infrastructures. A security infrastructure can be anything from a database or LDAP server to a sophisticated security software suite. The integration flexibility is achieved using the pluggable authentication model available in the JAAS framework.

The heart of the JBossSX framework is `org.jboss.security.plugins.JaasSecurityManager`. This is the default implementation of the `AuthenticationManager` and `RealmMapping` interfaces. *Figure 4.1, “The relationship between the security-domain component deployment descriptor value, the component container and the JaasSecurityManager.”* shows how the `JaasSecurityManager` integrates into the EJB and web container layers based on the `security-domain` element of the corresponding component deployment descriptor.

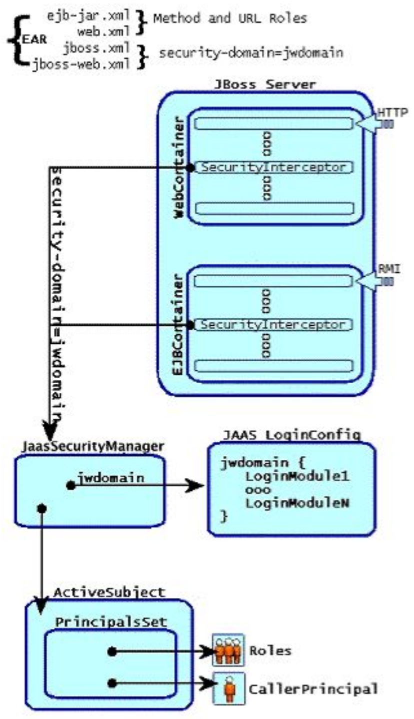

Figure 4.1. The relationship between the security-domain component deployment descriptor value, the component container and the JaasSecurityManager.

Figure 4.1, “The relationship between the security-domain component deployment descriptor value, the component container and the JaasSecurityManager.” depicts an enterprise application that contains both EJBs and web content secured under the security domain `jwdomain`. The EJB and web containers have a request interceptor architecture that includes a security interceptor, which enforces the container security model. At deployment time, the `security-domain` element value in the `jboss.xml` and `jboss-web.xml` descriptors is used to obtain the security manager instance associated with the container. The security interceptor then uses the security manager to perform its role. When a secured component is requested, the security interceptor delegates security checks to the security manager instance associated with the container.

The JBossSX `JaasSecurityManager` implementation performs security checks based on the information associated with the `Subject` instance that results from executing the JAAS login modules configured under the name matching the `security-domain` element value. We will drill into the `JaasSecurityManager` implementation and its use of JAAS in the following section.

4.1. How the JaasSecurityManager Uses JAAS

The `JaasSecurityManager` uses the JAAS packages to implement the `AuthenticationManager` and `RealmMapping` interface behavior. In particular, its behavior derives from the execution of the login module instances that are configured under the name that matches the security domain to which the `JaasSecurityManager` has been assigned. The login modules implement the security domain's principal authentication and role-mapping behavior. Thus, you can use the `JaasSecurityManager` across different security domains simply by plugging in different login module configurations for the domains.

To illustrate the details of the `JaasSecurityManager`'s usage of the JAAS authentication process, you will walk through a client invocation of an EJB home method invocation. The prerequisite setting is that the EJB has been deployed in the JBoss server and its home interface methods have been secured using `method-permission` elements in the `ejb-jar.xml` descriptor, and it has been assigned a security domain named `jwdomain` using the `jboss.xml` descriptor `security-domain` element.

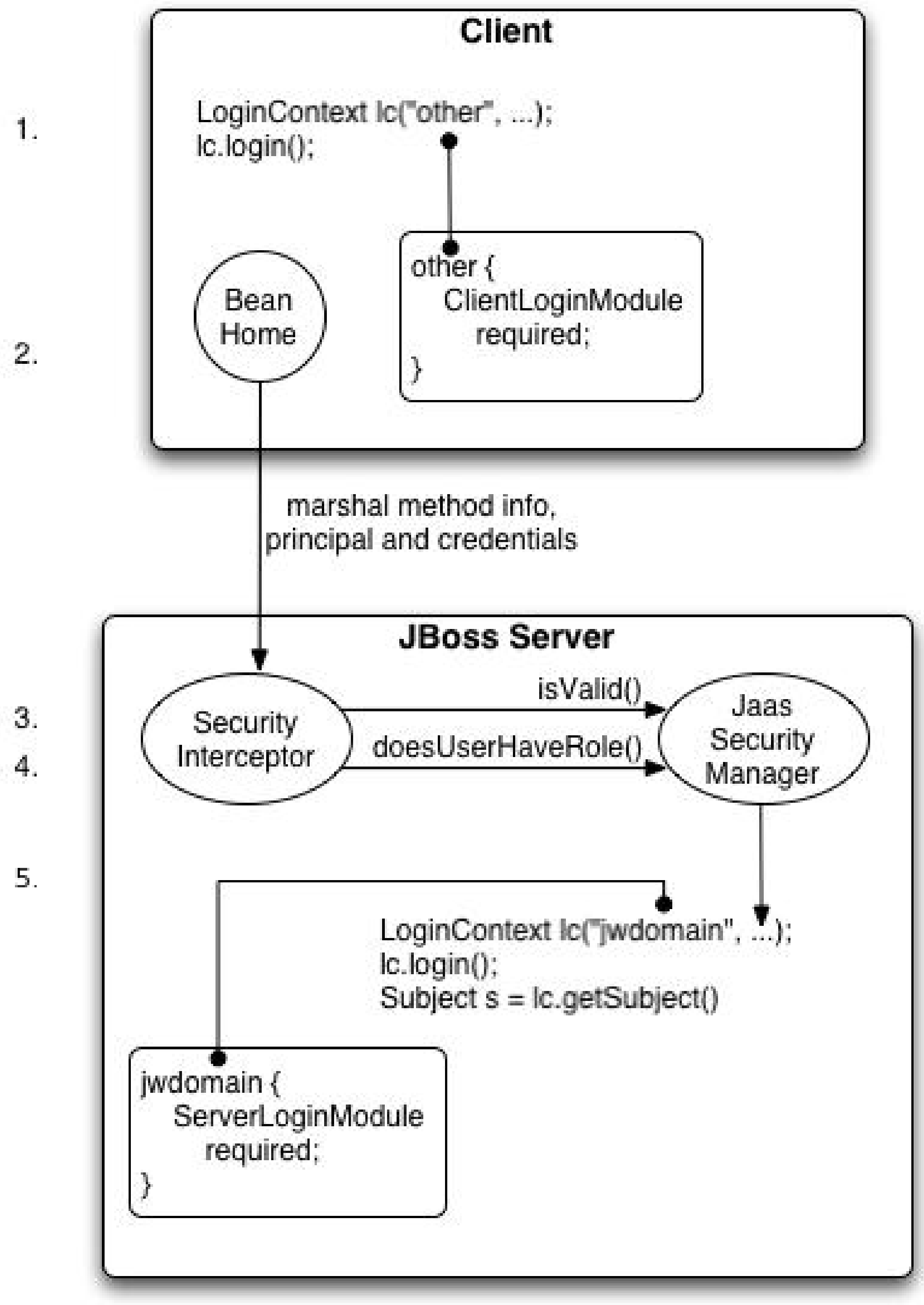

Figure 4.2. An illustration of the steps involved in the authentication and authorization of a secured EJB home method invocation.

Figure 4.2, “An illustration of the steps involved in the authentication and authorization of a secured EJB home method invocation.” provides a view of the client to server communication we will discuss. The numbered steps shown are:

1. The client first has to perform a JAAS login to establish the principal and credentials for authentication, and this is labeled *Client Side Login* in the figure. This is how clients establish their login identities in JBoss. Support for presenting the login information via JNDI `InitialContext` properties is provided via an alternate configuration. A JAAS login entails creating a `LoginContext` instance and passing the name of the configuration to use. The configuration name is `other`. This one-time login associates the login principal and credentials with all subsequent EJB method invocations. Note that the process might not authenticate the user. The nature of the client-side login depends on the login module configuration that the client uses. In this example, the `other` client-side login configuration entry is set up to use the `ClientLoginModule` module (an `org.jboss.security.ClientLoginModule`). This is the default client side module that simply binds the username and password to the JBoss EJB invocation layer for later authentication on the server. The identity of the client is not authenticated on the client.

2. Later, the client obtains the EJB home interface and attempts to create a bean. This event is labeled as *Home Method Invocation*. This results in a home interface method invocation being sent to the JBoss server. The invocation includes the method arguments passed by the client along with the user identity and credentials from the client-side JAAS login performed in step 1.

3. On the server side, the security interceptor first requires authentication of the user invoking the call, which, as on the client side, involves a JAAS login.

4. The security domain under which the EJB is secured determines the choice of login modules. The security domain name is used as the login configuration entry name passed to the `LoginContext` constructor. The EJB security domain is `jwdomain`. If the JAAS login authenticates the user, a JAAS `Subject` is created that contains the following in its `PrincipalsSet`:

   - A `java.security.Principal` that corresponds to the client identity as known in the deployment security environment.

   - A `java.security.acl.Group` named `Roles` that contains the role names from the application domain to which the user has been assigned. `org.jboss.security.SimplePrincipal` objects are used to represent the role names; `SimplePrincipal` is a simple string-based implementation of `Principal`. These roles are used to validate the roles assigned to methods in `ejb-jar.xml` and the `EJBContext.isCallerInRole(String)` method implementation.

   - An optional `java.security.acl.Group` named `CallerPrincipal`, which contains a single `org.jboss.security.SimplePrincipal` that corresponds to the identity of the application domain's caller. The `CallerPrincipal` sole group member will be the value returned by the `EJBContext.getCallerPrincipal()` method. The purpose of this mapping is to allow a `Principal` as known in the operational security environment to map to a `Principal` with a name known to the application. In the absence of a `CallerPrincipal` mapping the deployment security environment principal is used as the `getCallerPrincipal` method value. That is, the operational principal is the same as the application domain principal.

5. The final step of the security interceptor check is to verify that the authenticated user has permission to invoke the requested method This is labeled as *Server Side Authorization* in *Figure 4.2, “An illustration of the steps involved in the authentication and authorization of a secured EJB home method invocation.”*. Performing the authorization this entails the following steps:

   - Obtain the names of the roles allowed to access the EJB method from the EJB container. The role names are determined by `ejb-jar.xml` descriptor role-name elements of all `method-permission` elements containing the invoked method.

   - If no roles have been assigned, or the method is specified in an `exclude-list` element, then access to the method is denied. Otherwise, the `doesUserHaveRole` method is invoked on the security manager by the security interceptor to see if the caller has one of the assigned role names. This method iterates through the role names and checks if the authenticated user's Subject `Roles` group contains a `SimplePrincipal` with the assigned role name. Access is allowed if any role name is a member of the `Roles` group. Access is denied if none of the role names are members.

   - If the EJB was configured with a custom security proxy, the method invocation is delegated to it. If the security proxy wants to deny access to the caller, it will throw a `java.lang.SecurityException`. If no `SecurityException` is thrown, access to the EJB method is allowed and the method invocation passes to the next container interceptor. Note that the `SecurityProxyInterceptor` handles this check and this interceptor is not shown.

Every secured EJB method invocation, or secured web content access, requires the authentication and authorization of the caller because security information is handled as a stateless attribute of the request that must be presented and validated on each request. This can be an expensive operation if the JAAS login involves client-to-server communication. Because of this, the `JaasSecurityManager` supports the notion of an authentication cache that is used to store principal and credential information from previous successful logins. You can specify the authentication cache instance to use as part of the `JaasSecurityManager` configuration as you will see when the associated MBean service is discussed in following section. In the absence of any user-defined cache, a default cache that maintains credential information for a configurable period of time is used.

4.2. The JaasSecurityManagerService MBean

The `JaasSecurityManagerService` MBean service manages security managers. Although its name begins with *Jaas*, the security managers it handles need not use JAAS in their implementation. The name arose from the fact that the default security manager implementation is the `JaasSecurityManager`. The primary role of the `JaasSecurityManagerService` is to externalize the security manager implementation. You can change the security manager implementation by providing an alternate implementation of the `AuthenticationManager` and `RealmMapping` interfaces.

The second fundamental role of the `JaasSecurityManagerService` is to provide a JNDI `javax.naming.spi.ObjectFactory` implementation to allow for simple code-free management of the JNDI name to security manager implementation mapping. It has been mentioned that security is enabled by specifying the JNDI name of the security manager implementation via the `security-domain` deployment descriptor element. When you specify a JNDI name, there has to be an object-binding there to use. To simplify the setup of the JNDI name to security manager bindings, the `JaasSecurityManagerService` manages the association of security manager instances to names by binding a next naming system reference with itself as the JNDI ObjectFactory under the name `java:/jaas`. This allows one to use a naming convention of the form `java:/jaas/XYZ` as the value for the `security-domain` element, and the security manager instance for the `XYZ` security domain will be created as needed for you. The security manager for the domain `XYZ` is created on the first lookup against the `java:/jaas/XYZ` binding by creating an instance of the class specified by the `SecurityManagerClassName` attribute using a constructor that takes the name of the security domain. For example, consider the following container security configuration snippet:

java:/jaas prefix is no longer mandatory

In previous versions of JBoss, the `java:/jaas` prefix in each `securitydomain` deployment descrptor element was required to correctly bind the JNDI name to the security manager bindings. As of JBoss AS 6, it is possible to specify the name of the `securitydomain` only in `jboss.xml` and `jboss-web.xml`. The `java:/jaas` prefix is still supported however, and remains for backwards compatibility.

```
<jboss>
   <!-- Configure all containers to be secured under the "hades" security domain -->
   <security-domain>hades</security-domain>
   <!-- ... -->
</jboss>
```

Any lookup of the name `hades` will return a security manager instance that has been associated with the security domain named `hades`. This security manager will implement the AuthenticationManager and RealmMapping security interfaces and will be of the type specified by the `JaasSecurityManagerService` `SecurityManagerClassName` attribute.

The `JaasSecurityManagerService` MBean is configured by default for use in the standard JBoss distribution, and you can often use the default configuration as is. The configurable attributes of the `JaasSecurityManagerService` include:

- **SecurityManagerClassName**: The name of the class that provides the security manager implementation. The implementation must support both the `org.jboss.security.AuthenticationManager` and `org.jboss.security.RealmMapping` interfaces. If not specified this defaults to the JAAS-based `org.jboss.security.plugins.JaasSecurityManager`.
- **CallbackHandlerClassName**: The name of the class that provides the `javax.security.auth.callback.CallbackHandler` implementation used by the `JaasSecurityManager`. You can override the handler used by the `JaasSecurityManager` if the default implementation (`org.jboss.security.auth.callback.SecurityAssociationHandler`) does not meet your needs. This is a rather deep configuration that generally should not be set unless you know what you are doing.
- **SecurityProxyFactoryClassName**: The name of the class that provides the `org.jboss.security.SecurityProxyFactory` implementation. If not specified this defaults to `org.jboss.security.SubjectSecurityProxyFactory`.
- **AuthenticationCacheJndiName**: Specifies the location of the security credential cache policy. This is first treated as an `ObjectFactory` location capable of returning `CachePolicy` instances on a per-security-domain basis. This is done by appending the name of the security domain to this name when looking up the `CachePolicy` for a domain. If this fails, the location is treated as a single `CachePolicy` for all security domains. As a default, a timed cache policy is used.
- **DefaultCacheTimeout**: Specifies the default timed cache policy timeout in seconds. The default value is 1800 seconds (30 minutes). The value you use for the timeout is a tradeoff between frequent authentication operations and how long credential information may be out of sync with respect to the security information store. If you want to disable caching of security credentials, set this to 0 to force authentication to occur every time. This has no affect if the `AuthenticationCacheJndiName` has been changed from the default value.
- **DefaultCacheResolution**: Specifies the default timed cache policy resolution in seconds. This controls the interval at which the cache current timestamp is updated and should be less than the `DefaultCacheTimeout` in order for the timeout to be meaningful. The default resolution is 60 seconds(1 minute). This has no affect if the `AuthenticationCacheJndiName` has been changed from the default value.
- **DefaultUnauthenticatedPrincipal**: Specifies the principal to use for unauthenticated users. This setting makes it possible to set default permissions for users who have not been authenticated.
- **DefaultCacheFlushPeriod**: Specifies the default period of time in seconds that the authentication cache will flush expired entries. Default value is 3600 or one hour.

The `JaasSecurityManagerService` also supports a number of useful operations. These include flushing any security domain authentication cache at runtime, getting the list of active users in a security domain authentication cache, and any of the security manager interface methods.

Flushing a security domain authentication cache can be used to drop all cached credentials when the underlying store has been updated and you want the store state to be used immediately. The MBean operation signature is: `public void flushAuthenticationCache(String securityDomain)`.

This can be invoked programmatically using the following code snippet:

```
MBeanServer server = ...;
String jaasMgrName = "jboss.security:service=JaasSecurityManager";
ObjectName jaasMgr = new ObjectName(jaasMgrName);
Object[] params = {domainName};
String[] signature = {"java.lang.String"};
server.invoke(jaasMgr, "flushAuthenticationCache", params, signature);
```

Getting the list of active users provides a snapshot of the `Principals` keys in a security domain authentication cache that are not expired. The MBean operation signature is: `public List getAuthenticationCachePrincipals(String securityDomain)`.

This can be invoked programmatically using the following code snippet:

```
MBeanServer server = ...;
String jaasMgrName = "jboss.security:service=JaasSecurityManager";
ObjectName jaasMgr = new ObjectName(jaasMgrName);
Object[] params = {domainName};
String[] signature = {"java.lang.String"};
List users = (List) server.invoke(jaasMgr, "getAuthenticationCachePrincipals",
                                  params, signature);
```

The security manager has a few additional access methods.

```
public boolean isValid(String securityDomain, Principal principal, Object credential);
public Principal getPrincipal(String securityDomain, Principal principal);
public boolean doesUserHaveRole(String securityDomain, Principal principal,
                                Object credential, Set roles);
public Set getUserRoles(String securityDomain, Principal principal, Object credential);
```

They provide access to the corresponding `AuthenticationManager` and `RealmMapping` interface method of the associated security domain named by the `securityDomain` argument.

4.2.1. The JNDIBasedSecurityManagement Bean

In AS 6 most MBeans were replaced by *Micro Container* (MC) Beans. `JaasSecurityManagerService` was not removed to maintain compatibility with previous versions but most of its functionalities are done by the `JNDIBasedSecurityManagement` MC Bean now. This Bean is located in `conf/bootstrap/security.xml`.

In *Example 4.1, “Setting custom values for the JNDIBasedSecurityManagement Bean”* an example of how to set up the `AuthenticationManager` class, `CallbackHandler` class and default values for the authentication cache is shown:

Example 4.1. Setting custom values for the JNDIBasedSecurityManagement Bean

```
<bean name="JNDIBasedSecurityManagement"
      class="org.jboss.security.integration.JNDIBasedSecurityManagement">
   <property name="authenticationMgrClass">org.example.MyAuthenticationManager</property>
   <property name="defaultCacheTimeout">1800</property>
   <property name="defaultCacheResolution">60</property>
   <property name="defaultCacheFlushPeriod">3600</property>
   <property name="callBackHandler"><inject bean="CallbackHandler"/></property>
</bean>

<bean name="CallbackHandler" class="org.example.MyCallbackHandler"/>
```

4.3. The JaasSecurityDomain Bean

The `org.jboss.security.plugins.JaasSecurityDomain` is an extension of `JaasSecurityManager` that adds the notion of a `KeyStore`, a JSSE `KeyManagerFactory` and a `TrustManagerFactory` for supporting SSL and other cryptographic use cases. The additional configurable attributes of the `JaasSecurityDomain` include:

- **keyStoreType**: The type of the `KeyStore` implementation. This is the type argument passed to the `java.security.KeyStore.getInstance(String type)` factory method. The default is `JKS`.
- **keyStoreURL**: A URL to the location of the `KeyStore` database. This is used to obtain an `InputStream` to initialize the `KeyStore`. If the string is not a value URL, it is treated as a file.
- **keyStorePass**: The password associated with the `KeyStore` database contents. The `KeyStorePass` is also used in combination with the `Salt` and `IterationCount` attributes to create a PBE secret key used with the encode/decode operations. The `keyStorePass` attribute value format is one of the following:

  - The plaintext password for the KeyStore. The toCharArray() value of the string is used without any manipulation.
  - A command to execute to obtain the plaintext password. The format is {EXT}... where the ... is the exact command line that will be passed to the Runtime.exec(String) method to execute a platform-specific command. The first line of the command output is used as the password.
  - A class to create to obtain the plaintext password. The format is {CLASS}classname[:ctorarg] where the [:ctorarg] is an optional string that will be passed to the constructor when instantiating the classname. The password is obtained from classname by invoking a toCharArray() method if found, otherwise, the toString() method is used.
- **keyStoreAlias**: Alias of the KeyStore containing the certificate to be used.
- **keyStoreProvider**: Security Provider of the KeyStore.
- **keyStoreProviderArgument**: Argument to be passed to the constructor of the KeyStore security Provider.
- **keyManagerFactoryProvider**: Security Provider of the KeyManagerFactory.
- **keyManagerFactoryAlgorithm**: Algorithm of the KeyManagerFactory.
- **salt**: The PBEParameterSpec salt value.
- **iterationCount**: The PBEParameterSpec iteration count value.
- **trustStoreType**: The type of the TrustStore implementation. This is the type argument passed to the java.security.KeyStore.getInstance(String type) factory method. The default is JKS.
- **trustStoreURL**: A URL to the location of the TrustStore database. This is used to obtain an InputStream to initialize the KeyStore. If the string is not a value URL, it is treated as a file.
- **trustStorePass**: The password associated with the trust store database contents. The trustStorePass has the same configuration options as the keyStorePass.
- **trustStoreProvider**: Security Provider of the TrustStore.
- **trustStoreProviderArgument**: Argument to be passed to the constructor of the TrustStore security Provider.
- **trustManagerFactoryProvider**: Security Provider of the TrustManagerFactory.
- **trustManagerFactoryAlgorithm**: Algorithm of the TrustManagerFactory.

In *Example 4.2, “JaasSecurityDomain example”* an example JaasSecurityDomain Bean is shown:

Example 4.2. JaasSecurityDomain example

```
<bean name="example" class="org.jboss.security.plugins.JaasSecurityDomain">
      <constructor>
         <parameter>example</parameter>
      </constructor>
      <property name="keyStorePass">changeit</property>
      <property name="keyStoreURL">resource:localhost.keystore</property>
      <!-- introduce a JMX annotation to export this bean as an MBean -->
      <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX
              (name="jboss.security:service=JaasSecurityDomain,domain=example",
               exposedInterface=org.jboss.security.plugins.JaasSecurityDomainMBean.class)
      </annotation>
   </bean>
```

JaasSecurityDomain can still be deployed as a MBean

To maintain compatibility with previous versions, `JaasSecurityDomain` can still be deployed as a MBean.

Part II. Security Domains and Components

Static Security Domains

The standard way of configuring security domains for authentication and authorization in JBoss is to use the XML login configuration file. The login configuration policy defines a set of named security domains that each define a stack of login modules that will be called upon to authenticate and authorize users.

The XML configuration file conforms to the DTD given by *Figure 5.1, “The XMLLoginConfig DTD”*. This DTD can be found in `docs/dtd/security_config.dtd`.

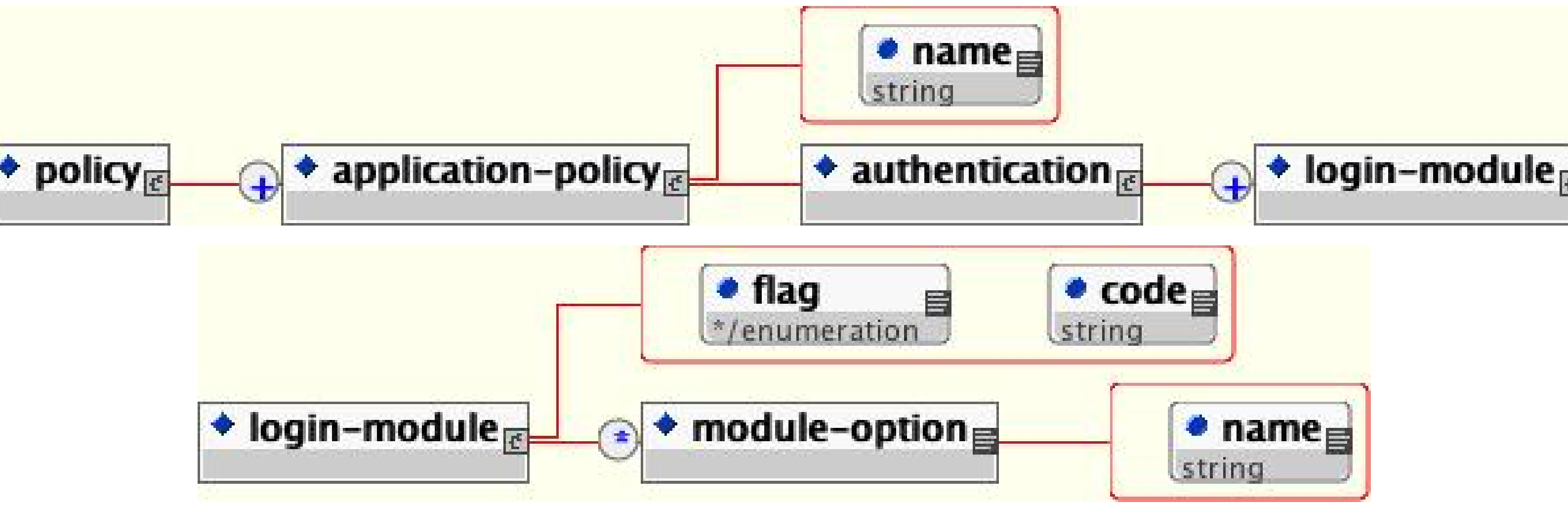

Figure 5.1. The XMLLoginConfig DTD

Example 5.1.

This example describes a simple configuration named jmx-console that is backed by a single login module. The login module is configured by a simple set of name/value configuration pairs that have meaning to the login module in question. We'll see what these options mean later, for now we'll just be concerned with the structure of the configuration file.

```
<application-policy name="example">
    <authentication>
        <login-module code="org.jboss.security.auth.spi.UsersRolesLoginModule"
                                   flag="required">
            <module-option name="usersProperties">users.properties</module-option>
            <module-option name="rolesProperties">roles.properties</module-option>
        </login-module>
    </authentication>
</application-policy>
```

The `name` attribute of the application-policy is the login configuration name. Applications policy elements are bound by that name in JNDI under the the `java:/jaas` context. Applications will link to security domains through this JNDI name in their deployment descriptors. (See the <security-domain> elements in `jboss.xml`, `jboss-web.xml` and `jboss-service.xml` files for examples)

The `code` attribute of the login-module element specifies the class name of the login module implementation. The `required` flag attribute controls the overall behavior of the authentication stack. The allowed values and meanings are:

required
: The login module is required to succeed for the authentication to be successful. If any required module fails, the authentication will fail. The remaining login modules in the stack will be called regardless of the outcome of the authentication.

requisite
: The login module is required to succeed. If it succeeds, authentication continues down the login stack. If it fails, control immediately returns to the application.

sufficient
: The login module is not required to succeed. If it does succeed, control immediately returns to the application. If it fails, authentication continues down the login stack.

optional
: The login module is not required to succeed. Authentication still continues to proceed down the login stack regardless of whether the login module succeeds or fails.

Example 5.2. Security Domain using Multiple Login Modules

This example shows the definition of a security domain that uses multiple login modules. Since both modules are marked as sufficient, only one of them must succeed for login to proceed.

```
<application-policy name="todo">
  <authentication>
    <login-module code="org.jboss.security.auth.spi.LdapLoginModule"
                  flag="sufficient">
      <!-- LDAP configuration -->
    </login-module>
    <login-module code="org.jboss.security.auth.spi.DatabaseServerLoginModule"
                  flag="sufficient">
      <!-- database configuration -->
    </login-module>
  </authentication>
</application-policy>
```

Each login module has its own set of configuration options. These are set as name/value pairs using the module-option elements. Module options are covered in more depth when we look at the individual login modules available in JBoss AS.

Loading Static Security Domains

Authentication security domains are configured statically in the `/server/$PROFILE/conf/login-config.xml` file, or deployed using `jboss-beans.xml` deployment descriptors. For static domains, the `XMLLoginConfig` bean is responsible for loading security configurations specified in `login-config.xml`. The bean definition is located in the `/server/$PROFILE/deploy/security/security-jboss-beans.xml` file. The bean is defined as shown below.

```
<bean name="XMLLoginConfig" class="org.jboss.security.auth.login.XMLLoginConfig">
   <property name="configResource">login-config.xml</property>
</bean>
```

The bean supports the following attributes:

configURL
: Specifies the URL of the XML login configuration file that should be loaded by this MBean on startup. This must be a valid URL string representation.

configResource
: Specifies the resource name of the XML login configuration file that should be loaded by this MBean on startup. The name is treated as a classpath resource for which a URL is located using the thread context class loader.

validateDTD
: Specifies whether the XML configuration should be validated against its DTD. This defaults to true.

The `SecurityConfig` bean is responsible for selecting the `javax.security.auth.login.Configuration` to be used. The default configuration simply references the `XMLLoginConfig` bean.

```
<bean name="SecurityConfig" class="org.jboss.security.plugins.SecurityConfig">
     <property name="mbeanServer"><inject bean="JMXKernel" property="mbeanServer"/></
property>
   <property name="defaultLoginConfig"><inject bean="XMLLoginConfig"/></property>
</bean>
```

There is one configurable attribute:

defaultLoginConfig
: Specifies the bean name of the MC bean that provides the default JAAS login configuration. When the `SecurityConfig` is started, this bean is queried for its `javax.security.auth.login.Configuration` by calling its `getConfiguration(Configuration currentConfig)` operation. If the `defaultLoginConfig` attribute is not specified then the default Sun `Configuration` implementation described in the `Configuration` class JavaDocs is used

Dynamic Security Domains

Historically, the Enterprise Application Platform used the static `$JBOSS_HOME/server/$PROFILE/conf/login-config.xml` file to configure the security domain. Dynamic configuration was provided with the introduction of the DynamicLoginConfig security service. This functionality allowed you to specify a Java Authentication and Authorization Service (JAAS) as part of an application deployment, rather than having to include the configuration information in `login-config.xml`.

JBoss AS 6 now provides an additional, simplified mechanism to configure security domains.

In JBoss AS, the security domain configuration is important for the authentication, authorization, auditing, and mapping functionality associated with Java EE components such a Web or EJBs.

The latest security implementation allows you to create a logically-named deployment descriptor file and specify the security domains within the file. The deployment descriptor can be deployed directly in the deploy folder, or packaged as part of the application JAR or WAR file.

Procedure 7.1. Security Domain Deployment Descriptor

Follow this procedure to configure a security domain deployment descriptor with two domains named web-test and ejb-test.

1. **Create deployment descriptor**

   You must create a deployment descriptor file to contain the security domain configuration.

   The filename takes the format `[domain_name]-jboss-beans.xml`. The *`domain_name`* is arbitrary, however you should choose a name that is meaningful to the application.

   The file must contain the standard XML declaration, and a correctly configured `<deployment>` element.

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns="urn:jboss:bean-deployer:2.0">


</deployment>
```

2. **Define application policies**

   Within the `<deployment>` element, the individual application policies are defined. Each policy specifies the login module to use, and any required options.

In the example below, two application policies are specified. Each policy uses the same login module, and module parameters.

Note

Other login modules are available for use with the Enterprise Application Platform. For more information about the available login modules, refer to *Section 10.1, “Using Modules”*

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns="urn:jboss:bean-deployer:2.0">

  <application-policy xmlns="urn:jboss:security-beans:1.0" name="web-test">
    <authentication>
                    <login-module    code="org.jboss.security.auth.spi.UsersRolesLoginModule"
 flag="required">
        <module-option name="unauthenticatedIdentity">anonymous</module-option>
        <module-option name="usersProperties">u.properties</module-option>
        <module-option name="rolesProperties">r.properties</module-option>
      </login-module>
    </authentication>
  </application-policy>

  <application-policy xmlns="urn:jboss:security-beans:1.0" name="ejb-test">
    <authentication>
                    <login-module    code="org.jboss.security.auth.spi.UsersRolesLoginModule"
 flag="required">
        <module-option name="unauthenticatedIdentity">anonymous</module-option>
        <module-option name="usersProperties">u.properties</module-option>
        <module-option name="rolesProperties">r.properties</module-option>
      </login-module>
    </authentication>
  </application-policy>

</deployment>
```

3. **Deploy or package the deployment descriptor**

   Move the deployment descriptor file to the `deploy` directory of the required server profile in your installation.

   Alternatively, package the deployment descriptor in the `META-INF` directory of the EJB Jar, or the `WEB-INF` directory of your web application (WAR).

Authorization Stacks

If a security domain does not define an authorization module, the default *`jboss-web-policy`* and *`jboss-ejb-policy`* authorization configured in `security-policies-jboss-beans.xml` is used. If you specify an authorization module, or create a custom deployment descriptor file with valid authorization configuration, these settings override the default settings in `security-policies-jboss-beans.xml`.

Overriding the default authorization for EJB or Web components is provided for JACC and XACML, apart from the default modules that implement the specification behavior. Users can provide authorization modules that implement custom behavior. Configuring this functionality allows access control stacks to be pluggable for a particular component, overriding the default authorization contained in `jboss.xml` (for EJBs) and `jboss-web.xml` (for WAR).

Setting authorization for all EJB and WEB components. You can override authorization for all EJBs and Web components, or for a particular component.

Procedure 8.1. Set authorization policies for all EJB and WAR components

This procedure describes how to define JACC Authorization control for all EJB and WAR components. The example defines application policy modules for Web and EJB applications: `jboss-web-policy`, and `jboss-ejb-policy`.

1. **Open the security policy bean**

   Navigate to `$JBOSS_HOME/server/$PROFILE/deploy/security`

   Open the `security-policies-jboss-beans.xml` file.

   By default, the security-policies-jboss-beans.xml file contains the configuration in *Example 8.1, “security-policies default configuration”*

   ### Example 8.1. security-policies default configuration

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns="urn:jboss:bean-deployer:2.0">

        <application-policy   xmlns="urn:jboss:security-beans:1.0"   name="jboss-web-policy"
 extends="other">
     <authorization>
                                                                                      <policy-module
 code="org.jboss.security.authorization.modules.DelegatingAuthorizationModule"
                                   flag="required"/>
```

```
    </authorization>
  </application-policy>

  <application-policy xmlns="urn:jboss:security-beans:1.0" name="jboss-ejb-policy"
 extends="other">
    <authorization>
      <policy-module
 code="org.jboss.security.authorization.modules.DelegatingAuthorizationModule"
                               flag="required"/>
    </authorization>
  </application-policy>

</deployment>
```

2. **Change the application-policy definitions**

   To set a single authorization policy for each component using JACC, amend each `<policy-module>` *`code`* attribute with the name of the JACC authorization module.

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns="urn:jboss:bean-deployer:2.0">

  <application-policy xmlns="urn:jboss:security-beans:1.0" name="jboss-web-policy"
 extends="other">
    <authorization>
      <policy-module

 flag="required"/>
    </authorization>
  </application-policy>

  <application-policy xmlns="urn:jboss:security-beans:1.0" name="jboss-ejb-policy"
 extends="other">
    <authorization>
      <policy-module

 flag="required"/>
    </authorization>
  </application-policy>

  <application-policy xmlns="urn:jboss:security-beans:1.0" name="jacc-test"
 extends="other">
```

```
    <authorization>
                                                                           <policy-module


 flag="required"/>
    </authorization>
  </application-policy>

</deployment>
```

3. **Restart server**

   You have now configured the `security-policy-jboss-beans.xml` file with JACC authorization enabled for each application policy.

   Restart the server to ensure the new security policy takes effect.

Setting authorization for specific EJB and WEB components. If applications require more granular security policies, you can declare multiple authorization security policies for each application policy. New security domains can inherit base settings from another application policy, and override specific settings such as the authorization policy module.

Procedure 8.2. Set authorization policies for specific security domains

This procedure describes how to inherit settings from other application policy definitions, and specify different authorization policies per security domain.

In this procedure, two security domains are defined. The *`test-domain`* security domain uses the UsersRolesLoginModule login module and uses JACC authorization. The *`test-domain-inherited`* security domain inherits the login module information from *`test-domain`*, and specifies XACML authorization must be used.

1. **Open the security policy**

   You can specify the security domain settings in the `login-config.xml` file, or create a deployment descriptor file containing the settings. Choose the deployment descriptor if you want to package the security domain settings with your application.

   - **Locate and open login-config.xml**

     Navigate to the `login-config.xml` file for the server profile you are using and open the file for editing. For example:

     `$JBOSS_HOME/jboss-as/server/$PROFILE/conf/login.config.xml`

- **Create a jboss-beans.xml descriptor**

  Create a *`[prefix]`*`-jboss-beans.xml` descriptor, replacing *`[prefix]`* with a meaningful name (for example, `test-war-jboss-beans.xml`)

  Save this file in the deploy directory of the server profile you are configuring. For example:

  `$JBOSS_HOME``/jboss-as/server/$PROFILE/deploy/test-war-jboss-beans.xml`

2. **Specify the test-domain security domain**

   In the target file chosen in step 1, specify the *`test-domain`* security domain. This domain contains the authentication information, including the `<login-module>` definition, and the JACC authorization policy module definition.

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns="urn:jboss:bean-deployer:2.0">

  <application-policy xmlns="urn:jboss:security-beans:1.0" name="test-domain">
    <authentication>
      <login-module code = "org.jboss.security.auth.spi.UsersRolesLoginModule"
        flag = "required">
        <module-option name = "unauthenticatedIdentity">anonymous</module-option>
        <module-option name="usersProperties">u.properties</module-option>
        <module-option name="rolesProperties">r.properties</module-option>
      </login-module>
    </authentication>
    <authorization>
                                                            <policy-module

 flag="required"/>
    </authorization>
  </application-policy>


</deployment>
```

3. **Append the test-domain-inherited security domain**

   Append the *`test-domain-inherited`* application policy definition after the *`test-domain`* application policy. Set the *`extends`* attribute to `other`, so the login module information is inherited. Specify the XACML authorization module in the `<policy.module>` element.

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns="urn:jboss:bean-deployer:2.0">

  <application-policy xmlns="urn:jboss:security-beans:1.0" name="test-domain">
    <authentication>
      <login-module code = "org.jboss.security.auth.spi.UsersRolesLoginModule"
          flag = "required">
          <module-option name = "unauthenticatedIdentity">anonymous</module-option>
          <module-option name="usersProperties">u.properties</module-option>
          <module-option name="rolesProperties">r.properties</module-option>
       </login-module>
     </authentication>
     <authorization>
                                                                    <policy-module

 flag="required"/>
     </authorization>
  </application-policy>

    <application-policy  xmlns="urn:jboss:security-beans:1.0"  name="test-domain-inherited"
 extends="other">
     <authorization>
                                                                    <policy-module

 flag="required"/>
     </authorization>
  </application-policy>

</deployment>
```

4. **Restart server**

   You have now configured the target file with two security domains that use different authorization methods.

   Restart the server to ensure the new security policy takes effect.

Setting authorization module delegates. *Set authorization policies for all EJB and WAR components* and *Set authorization policies for specific security domains* describe simplistic examples that show how authentication and authorization can be configured in security domains.

Because authorization relates to the type of component (not the layer) you want to protect, you can use delegation within a deployment descriptor to specify different authorization policies to the standard authentication in your implementation.

The delegates must be a subclass of `AuthorizationModuleDelegate`. *Example 8.2, “AuthorizationModuleDelegate class”* describes the base `AuthorizationModuleDelegate` interface.

Example 8.2. AuthorizationModuleDelegate class

```
package org.jboss.security.authorization.modules;

import javax.security.auth.Subject;

import org.jboss.logging.Logger;
import org.jboss.security.authorization.AuthorizationModule;
import org.jboss.security.authorization.PolicyRegistration;
import org.jboss.security.authorization.Resource;
import org.jboss.security.identity.RoleGroup;

//$Id$

/**
 *  Delegate for Authorization Module
 *  @author <a href="mailto:Anil.Saldhana@jboss.org">Anil Saldhana</a>
 *  @since  Jun 19, 2006
 *  @version $Revision$
 */
public abstract class AuthorizationModuleDelegate
{
   protected static Logger log = Logger.getLogger(AuthorizationModuleDelegate.class);
   protected boolean trace = false;

   /**
    * Policy Registration Manager Injected
    */
   protected PolicyRegistration policyRegistration = null;

   /**
    * @see AuthorizationModule#authorize(Resource)
    * @param resource
    * @param subject Authenticated Subject
    * @param role RoleGroup
    * @return
```

```
     */
    public abstract int authorize(Resource resource, Subject subject, RoleGroup role);

    /**
     * Set the PolicyRegistration manager
     * Will be used to query for the policies
     * @param authzManager
     */
    public void setPolicyRegistrationManager(PolicyRegistration pm)
    {
       this.policyRegistration = pm;
    }
}
```

Some examples of authorization delegation are included for reference. *Example 8.3, “EJBJACCPolicyModuleDelegate.java”* describes an authorization module responsible for authorization decisions for the EJB layer. *Example 8.4, “WebJACCPolicyModuleDelegate.java”* describes a JACC-based authorization module helper that controls web layer authorization decisions.

Example 8.3. EJBJACCPolicyModuleDelegate.java

```
package org.jboss.security.authorization.modules.ejb;

import java.lang.reflect.Method;
import java.security.CodeSource;
import java.security.Permission;
import java.security.Policy;
import java.security.Principal;
import java.security.ProtectionDomain;
import java.util.Map;

import javax.security.auth.Subject;
import javax.security.jacc.EJBMethodPermission;
import javax.security.jacc.EJBRoleRefPermission;

import org.jboss.logging.Logger;
import org.jboss.security.authorization.AuthorizationContext;
import org.jboss.security.authorization.PolicyRegistration;
import org.jboss.security.authorization.Resource;
import org.jboss.security.authorization.ResourceKeys;
import org.jboss.security.authorization.modules.AbstractJACCModuleDelegate;
import org.jboss.security.authorization.modules.AuthorizationModuleDelegate;
import org.jboss.security.authorization.resources.EJBResource;
import org.jboss.security.identity.Role;
import org.jboss.security.identity.RoleGroup;


//$Id$

/**
 *  Authorization Module delegate that deals with the authorization decisions
 *  for the EJB Layer
 *  @author <a href="mailto:Anil.Saldhana@jboss.org">Anil Saldhana</a>
 *  @since  Jul 6, 2006
 *  @version $Revision$
 */
public class EJBJACCPolicyModuleDelegate extends AbstractJACCModuleDelegate
{
   private String ejbName = null;
   private Method ejbMethod = null;
   private String methodInterface = null;
   private CodeSource ejbCS = null;
   private String roleName = null;
   private Boolean roleRefCheck = Boolean.FALSE;

   public EJBJACCPolicyModuleDelegate()
   {
      log = Logger.getLogger(getClass());
      trace = log.isTraceEnabled();
   }

   /**
    * @see AuthorizationModuleDelegate#authorize(Resource)
    */
   public int authorize(Resource resource, Subject callerSubject, RoleGroup role)
   {
      if(resource instanceof EJBResource == false)
         throw new IllegalArgumentException("resource is not an EJBResource");

      EJBResource ejbResource = (EJBResource) resource;

      //Get the context map
      Map<String,Object> map = resource.getMap();
      if(map == null)
```

```
        throw new IllegalStateException("Map from the Resource is null");

                                                   this.policyRegistration        =        (PolicyRegistration)
map.get(ResourceKeys.POLICY_REGISTRATION);

      this.ejbCS = ejbResource.getCodeSource();
      this.ejbMethod = ejbResource.getEjbMethod();
      this.ejbName = ejbResource.getEjbName();
      this.methodInterface = ejbResource.getEjbMethodInterface();

      //isCallerInRole checks
      this.roleName = (String)map.get(ResourceKeys.ROLENAME);

      this.roleRefCheck = (Boolean)map.get(ResourceKeys.ROLEREF_PERM_CHECK);
      if(this.roleRefCheck == Boolean.TRUE)
        return checkRoleRef(callerSubject, role);
      else
        return process(callerSubject, role);
   }

   //Private Methods
   /**
    * Process the request
    * @param request
    * @param sc
    * @return
    */
   private int process(Subject callerSubject, Role role)
   {
      EJBMethodPermission methodPerm =
        new EJBMethodPermission(ejbName, methodInterface, ejbMethod);
      boolean policyDecision = checkWithPolicy(methodPerm, callerSubject, role);
      if( policyDecision == false )
      {
        String msg = "Denied: "+methodPerm+", caller=" + callerSubject+", role="+role;
        if(trace)
          log.trace("EJB Jacc Delegate:"+msg);
      }
      return policyDecision ? AuthorizationContext.PERMIT : AuthorizationContext.DENY;
   }

   private int checkRoleRef(Subject callerSubject, RoleGroup callerRoles)
   {
      //This has to be the EJBRoleRefPermission
```

```
     EJBRoleRefPermission ejbRoleRefPerm = new EJBRoleRefPermission(ejbName,roleName);
      boolean policyDecision = checkWithPolicy(ejbRoleRefPerm, callerSubject, callerRoles);
      if( policyDecision == false )
      {
         String msg = "Denied: "+ejbRoleRefPerm+", caller=" + callerSubject;
         if(trace)
            log.trace("EJB Jacc Delegate:"+msg);
      }
      return policyDecision ? AuthorizationContext.PERMIT : AuthorizationContext.DENY;
   }

   private boolean checkWithPolicy(Permission ejbPerm, Subject subject, Role role)
   {
      Principal[] principals = this.getPrincipals(subject, role);
      ProtectionDomain pd = new ProtectionDomain (ejbCS, null, null, principals);
      return Policy.getPolicy().implies(pd, ejbPerm);
   }
}
```

Example 8.4. WebJACCPolicyModuleDelegate.java

```
package org.jboss.security.authorization.modules.web;

import java.io.IOException;
import java.security.CodeSource;
import java.security.Permission;
import java.security.Policy;
import java.security.Principal;
import java.security.ProtectionDomain;
import java.util.Map;
import java.util.Set;

import javax.security.auth.Subject;
import javax.security.jacc.WebResourcePermission;
import javax.security.jacc.WebRoleRefPermission;
import javax.security.jacc.WebUserDataPermission;
import javax.servlet.http.HttpServletRequest;

import org.jboss.logging.Logger;
import org.jboss.security.authorization.AuthorizationContext;
import org.jboss.security.authorization.PolicyRegistration;
import org.jboss.security.authorization.Resource;
import org.jboss.security.authorization.ResourceKeys;
import org.jboss.security.authorization.modules.AbstractJACCModuleDelegate;
import org.jboss.security.authorization.modules.AuthorizationModuleDelegate;
import org.jboss.security.authorization.resources.WebResource;
import org.jboss.security.identity.Role;
import org.jboss.security.identity.RoleGroup;


//$Id:       WebJACCPolicyModuleDelegate.java      62923      2007-05-09      03:08:14Z
 anil.saldhana@jboss.com $

/**
 *  JACC based authorization module helper that deals with the web layer
 *  authorization decisions
 *  @author <a href="mailto:Anil.Saldhana@jboss.org">Anil Saldhana</a>
 *  @since  July 7, 2006
 *  @version $Revision: 62923 $
 */
public class WebJACCPolicyModuleDelegate extends AbstractJACCModuleDelegate
{
   private Policy policy = Policy.getPolicy();
   private HttpServletRequest request = null;
   private CodeSource webCS = null;

   private String canonicalRequestURI = null;

   public WebJACCPolicyModuleDelegate()
   {
      log = Logger.getLogger(WebJACCPolicyModuleDelegate.class);
      trace = log.isTraceEnabled();
   }

   /**
    * @see AuthorizationModuleDelegate#authorize(Resource)
    */
   @SuppressWarnings("unchecked")
   public int authorize(Resource resource, Subject callerSubject, RoleGroup role)
   {
      if(resource instanceof WebResource == false)
         throw new IllegalArgumentException("resource is not a WebResource");

      WebResource webResource = (WebResource) resource;
```

```
     //Get the context map
     Map<String,Object> map = resource.getMap();
     if(map == null)
       throw new IllegalStateException("Map from the Resource is null");

     //Get the Request Object
     request = (HttpServletRequest) webResource.getServletRequest();

     webCS = webResource.getCodeSource();
     this.canonicalRequestURI = webResource.getCanonicalRequestURI();

     String roleName = (String)map.get(ResourceKeys.ROLENAME);
     Principal principal = (Principal)map.get(ResourceKeys.HASROLE_PRINCIPAL);
     Set<Principal> roles = (Set<Principal>)map.get(ResourceKeys.PRINCIPAL_ROLES);
     String servletName = webResource.getServletName();
                                                 Boolean          resourceCheck          =
checkBooleanValue((Boolean)map.get(ResourceKeys.RESOURCE_PERM_CHECK));
                                                 Boolean          userDataCheck          =
checkBooleanValue((Boolean)map.get(ResourceKeys.USERDATA_PERM_CHECK));
                                                  Boolean          roleRefCheck          =
checkBooleanValue((Boolean)map.get(ResourceKeys.ROLEREF_PERM_CHECK));

     validatePermissionChecks(resourceCheck,userDataCheck,roleRefCheck);

     boolean decision = false;

     try
     {
       if(resourceCheck)
         decision = this.hasResourcePermission(callerSubject, role);
       else
       if(userDataCheck)
        decision = this.hasUserDataPermission();
       else
       if(roleRefCheck)
         decision = this.hasRole(principal, roleName, roles, servletName);
       else
         if(trace)
          log.trace("Check is not for resourcePerm, userDataPerm or roleRefPerm.");
     }
     catch(IOException ioe)
     {
       if(trace)
         log.trace("IOException:",ioe);
```

```
      }
      return decision ? AuthorizationContext.PERMIT : AuthorizationContext.DENY;
   }

   /**
    * @see AuthorizationModuleDelegate#setPolicyRegistrationManager(PolicyRegistration)
    */
   public void setPolicyRegistrationManager(PolicyRegistration authzM)
   {
     this.policyRegistration = authzM;
   }

   //**************************************************************************
   //  PRIVATE METHODS
   //**************************************************************************
   /** See if the given JACC permission is implied using the caller as
    * obtained from either the
    * PolicyContext.getContext(javax.security.auth.Subject.container) or
    * the info associated with the requestPrincipal.
    *
    * @param perm - the JACC permission to check
    * @param requestPrincpal - the http request getPrincipal
    * @param caller the authenticated subject obtained by establishSubjectContext
    * @return true if the permission is allowed, false otherwise
    */
   private boolean checkPolicy(Permission perm, Principal requestPrincpal,
         Subject caller, Role role)
   {
      // Get the caller principals, its null if there is no caller
      Principal[] principals = getPrincipals(caller,role);

      return checkPolicy(perm, principals);
   }


   /** See if the given permission is implied by the Policy. This calls
    * Policy.implies(pd, perm) with the ProtectionDomain built from the
    * active CodeSource set by the JaccContextValve, and the given
    * principals.
    *
    * @param perm - the JACC permission to evaluate
    * @param principals - the possibly null set of principals for the caller
    * @return true if the permission is allowed, false otherwise
    */
```

```
private boolean checkPolicy(Permission perm, Principal[] principals)
{
   ProtectionDomain pd = new ProtectionDomain(webCS, null, null, principals);
   boolean allowed = policy.implies(pd, perm);
   if( trace )
   {
      String msg = (allowed ? "Allowed: " : "Denied: ") +perm;
      log.trace(msg);
   }
   return allowed;
}

/**
 * Ensure that the bool is a valid value
 * @param bool
 * @return bool or Boolean.FALSE (when bool is null)
 */
private Boolean checkBooleanValue(Boolean bool)
{
   if(bool == null)
      return Boolean.FALSE;
   return bool;
}


/**
 * Perform hasResourcePermission Check
 * @param request
 * @param response
 * @param securityConstraints
 * @param context
 * @param caller
 * @return
 * @throws IOException
 */
private boolean hasResourcePermission(Subject caller, Role  role)
throws IOException
{
   Principal requestPrincipal = request.getUserPrincipal();
  WebResourcePermission perm = new WebResourcePermission(this.canonicalRequestURI,
                                              request.getMethod());
   boolean allowed = checkPolicy(perm, requestPrincipal, caller, role );
   if( trace )
      log.trace("hasResourcePermission, perm="+perm+", allowed="+allowed);
   return allowed;
}

/**
 * Perform hasRole check
 * @param principal
 * @param role
 * @param roles
 * @return
 */
private boolean hasRole(Principal principal, String roleName,
      Set<Principal> roles, String servletName)
{
   if(servletName == null)
      throw new IllegalArgumentException("servletName is null");

   WebRoleRefPermission perm = new WebRoleRefPermission(servletName, roleName);
   Principal[] principals = {principal};
   if( roles != null )
   {
      principals = new Principal[roles.size()];
      roles.toArray(principals);
   }
   boolean allowed = checkPolicy(perm, principals);
   if( trace )
      log.trace("hasRole, perm="+perm+", allowed="+allowed);
   return allowed;
}

/**
 * Perform hasUserDataPermission check for the realm.
 * If this module returns false, the base class (Realm) will
 * make the decision as to whether a redirection to the ssl
 * port needs to be done
 * @param request
 * @param response
 * @param constraints
 * @return
 * @throws IOException
 */
private boolean hasUserDataPermission() throws IOException
{
   WebUserDataPermission perm = new WebUserDataPermission(this.canonicalRequestURI,
                                         request.getMethod());
      if( trace )
         log.trace("hasUserDataPermission, p="+perm);
      boolean ok = false;
      try
      {
         Principal[] principals = null;
         ok = checkPolicy(perm, principals);
      }
      catch(Exception e)
      {
         if( trace )
            log.trace("Failed to checkSecurityAssociation", e);
      }
      return ok;
   }

   /**
    * Validate that the access check is made only for one of the
    * following
    * @param resourceCheck
    * @param userDataCheck
    * @param roleRefCheck
    */
   private void validatePermissionChecks(Boolean resourceCheck,
         Boolean userDataCheck, Boolean roleRefCheck)
   {
      if(trace)
         log.trace("resourceCheck="+resourceCheck + " : userDataCheck=" + userDataCheck
               + " : roleRefCheck=" + roleRefCheck);
     if((resourceCheck == Boolean.TRUE && userDataCheck == Boolean.TRUE && roleRefCheck
 == Boolean.TRUE )
           || (resourceCheck == Boolean.TRUE && userDataCheck == Boolean.TRUE)
           || (userDataCheck == Boolean.TRUE && roleRefCheck == Boolean.TRUE))
         throw new IllegalStateException("Permission checks must be different");
   }
}
```

Deployment-level Role Mapping

In JBoss AS 6, it is possible to map additional roles at the deployment level from those derived at the security domain level (such as at the EAR level). This is achieved by declaring the `org.jboss.security.mapping.providers.DeploymentRolesMappingProvider` class as the value for the *`class`* attribute in the `<mapping-module>` element. Additionally, the *`type`* attribute must be set to `role`.

By configuring the mapping configuration element within the role-based parameter, you can force additional role interpretation to the declared principals specified for the particular deployment (war, ear, ejb-jar etc).

Important: <rolemapping> deprecated for <mapping>

In previous versions, the `<rolemapping>` element contained the `<mapping-module>` element and class declaration. `<rolemapping>` has now been deprecated, and replaced with the `<mapping>` element.

Example 9.1. <mapping-module> declaration

```
<application-policy name="some-sec-domain">
<authentication>
...
</authentication>
<mapping>
                                                          <mapping-module
 code="org.jboss.security.mapping.providers.DeploymentRolesMappingProvider"
          type="role"/>
</mapping>
...
</application-policy>
```

Once the security domain is configured correctly, you can append the `<security-role>` element group as a child element of the `<assembly-descriptor>` to the `jboss.xml`, or `jboss-web.xml` files.

Example 9.2. <security-role> declaration

```
<assembly-descriptor>
  ...
```

```
    <security-role>
      <role-name>Support</role-name>
      <principal-name>Mark</principal-name>
      <principal-name>Tom</principal-name>
    </security-role>
  ...
</assembly-descriptor>
```

In *Example 9.2, “<security-role> declaration”*, a security role relating to Support principals is implemented in addition to the base security role information contained in `jboss.xml` or `jboss-web.xml`.

JBoss Login Modules

10.1. Using Modules

JBoss AS includes several bundled login modules suitable for most user management needs. JBoss AS can read user information from a relational database, a LDAP server or flat files. In addition to these core login modules, JBoss provides several other login modules that provide user information for very customized needs in JBoss. Before we explore the individual login modules, let's take a look at a few login module configuration options that are common to multiple modules.

10.1.1. Password Stacking

Multiple login modules can be chained together in a stack, with each login module providing both the authentication and authorization components. This works for many use cases, but sometimes authentication and authorization are split across multiple user management stores.

*Section 10.1.7, “LdapLoginModule”*describes how to combine LDAP and a relational database, allowing a user to be authenticated by either system. However, consider the case where users are managed in a central LDAP server but application-specific roles are stored in the application's relational database. The password-stacking module option captures this relationship.

To use password stacking, each login module should set the <module-option> `password-stacking` attribute to `useFirstPass`. If a previous module configured for password stacking has authenticated the user, all the other stacking modules will consider the user authenticated and only attempt to provide a set of roles for the authorization step.

When `password-stacking` option is set to `useFirstPass`, this module first looks for a shared username and password under the property names javax.security.auth.login.name and javax.security.auth.login.password respectively in the login module shared state map.

If found, these properties are used as the principal name and password. If not found, the principal name and password are set by this login module and stored under the property names javax.security.auth.login.name and javax.security.auth.login.password respectively.

Note

When using password stacking, set all modules to be required. This ensures that all modules are considered, and have the chance to contribute roles to the authorization process.

Example 10.1. Password Stacking Sample

This example shows how password stacking could be used.

```
<application-policy name="todo">
  <authentication>
    <login-module code="org.jboss.security.auth.spi.LdapLoginModule"
          flag="required">
      <!-- LDAP configuration -->
      <module-option name="password-stacking">useFirstPass</module-option>
    </login-module>
    <login-module code="org.jboss.security.auth.spi.DatabaseServerLoginModule"
          flag="required">
      <!-- database configuration -->
      <module-option name="password-stacking">useFirstPass</module-option>
    </login-module>
  </authentication>
</application-policy>
```

10.1.2. Password Hashing

Most login modules must compare a client-supplied password to a password stored in a user management system. These modules generally work with plain text passwords, but can be configured to support hashed passwords to prevent plain text passwords from being stored on the server side.

Example 10.2. Password Hashing

The following is a login module configuration that assigns unauthenticated users the principal name `nobody` and contains based64-encoded, MD5 hashes of the passwords in a `usersb64.properties` file. The `usersb64.properties` file can be part of the deployment classpath, or be saved in the `/conf` directory.

```
<policy>
  <application-policy name="testUsersRoles">
    <authentication>
      <login-module code="org.jboss.security.auth.spi.UsersRolesLoginModule"
            flag="required">
        <module-option name="usersProperties">usersb64.properties</module-option>
        <module-option name="rolesProperties">test-users-roles.properties</module-option>
        <module-option name="unauthenticatedIdentity">nobody</module-option>
        <module-option name="hashAlgorithm">MD5</module-option>
        <module-option name="hashEncoding">base64</module-option>
      </login-module>
    </authentication>
  </application-policy>
```

```
</policy>
```

hashAlgorithm
: Name of the `java.security.MessageDigest` algorithm to use to hash the password. There is no default so this option must be specified to enable hashing. Typical values are `MD5` and `SHA`.

hashEncoding
: String that specifies one of three encoding types: `base64`, `hex` or `rfc2617`. The default is `base64`.

hashCharset
: Encoding character set used to convert the clear text password to a byte array. The platform default encoding is the default.

hashUserPassword
: Specifies the hashing algorithm must be applied to the password the user submits. The hashed user password is compared against the value in the login module, which is expected to be a hash of the password. The default is `true`.

hashStorePassword
: Specifies the hashing algorithm must be applied to the password stored on the server side. This is used for digest authentication, where the user submits a hash of the user password along with a request-specific tokens from the server to be compare. The hash algorithm (for digest, this would be `rfc2617`) is utilized to compute a server-side hash, which should match the hashed value sent from the client.

If you must generate passwords in code, `the org.jboss.security.Util` class provides a static helper method that will hash a password using the specified encoding.

```
String hashedPassword = Util.createPasswordHash("MD5",
                                        Util.BASE64_ENCODING,
                                        null,
                                        null,
                                        "password");
```

OpenSSL provides an alternative way to quickly generate hashed passwords.

```
echo -n password | openssl dgst -md5 -binary | openssl base64
```

In both cases, the text password should hash to `X03MO1qnZdYdgyfeuILPmQ==`. This value must be stored in the user store.

10.1.3. Unauthenticated Identity

Not all requests are received in an authenticated format. `unauthenticated identity` is a login module configuration option that assigns a specific identity (guest, for example) to requests that are made with no associated authentication information. This can be used to allow unprotected servlets to invoke methods on EJBs that do not require a specific role. Such a principal has no associated roles and so can only access either unsecured EJBs or EJB methods that are associated with the unchecked permission constraint.

- **unauthenticatedIdentity**: This defines the principal name that should be assigned to requests that contain no authentication information.

10.1.4. Principal Class

Sometimes the implementation of the `Principal` interface provided by JBoss is not enough for the applications needs. In this case customers can use a custom implementation.

- **principalClass**: An option that specifies a `Principal` implementation class. This must support a constructor taking a string argument for the principal name.

10.1.5. UsersRolesLoginModule

`UsersRolesLoginModule` is a simple login module that supports multiple users and user roles loaded from Java properties files. The username-to-password mapping file is called `users.properties` and the username-to-roles mapping file is called `roles.properties`.

The supported login module configuration options include the following:

usersProperties
: Name of the properties resource (file) containing the username to password mappings. This defaults to `<filename_prefix>-users.properties`.

rolesProperties
: Name of the properties resource (file) containing the username to roles mappings. This defaults to `<filename_prefix>-roles.properties`.

This login module supports password stacking, password hashing, and unauthenticated identity.

The properties files are loaded during initialization using the initialize method thread context class loader. This means that these files can be placed into the Java EE deployment JAR, the JBoss configuration directory, or any directory on the JBoss server or system classpath. The primary purpose of this login module is to easily test the security settings of multiple users and roles using properties files deployed with the application.

Example 10.3. UserRolesLoginModule

```
<deployment xmlns="urn:jboss:bean-deployer:2.0">

  <!-- ejb3 test application-policy definition -->
  <application-policy xmlns="urn:jboss:security-beans:1.0" name="ejb3-sampleapp">
    <authentication>
      <login-module code="org.jboss.security.auth.spi.UsersRolesLoginModule" flag="required">
            <module-option name="usersProperties">ejb3-sampleapp-users.properties</module-
option>
             <module-option name="rolesProperties">ejb3-sampleapp-roles.properties</module-
option>
       </login-module>
    </authentication>
  </application-policy>

</deployment>
```

In *Example 10.3, “UserRolesLoginModule”*, the `ejb3-sampleapp-users.properties` file uses a `username=password` format with each user entry on a separate line:

```
username1=password1
username2=password2
...
```

The `ejb3-sampleapp-roles.properties` file referenced in *Example 10.3, “UserRolesLoginModule”* uses the pattern `username=role1,role2,` with an optional group name value. For example:

```
username1=role1,role2,...
username1.RoleGroup1=role3,role4,...
username2=role1,role3,...
```

The username.XXX property name pattern present in `ejb3-sampleapp-roles.properties` is used to assign the username roles to a particular named group of roles where the

`XXX` portion of the property name is the group name. The username=... form is an abbreviation for username.Roles=..., where the `Roles` group name is the standard name the `JaasSecurityManager` expects to contain the roles which define the users permissions.

The following would be equivalent definitions for the `jduke` username:

```
jduke=TheDuke,AnimatedCharacter
jduke.Roles=TheDuke,AnimatedCharacter
```

10.1.6. DatabaseServerLoginModule

The `DatabaseServerLoginModule` is a Java Database Connectivity-based (JDBC) login module that supports authentication and role mapping. Use this login module if you have your username, password and role information stored in a relational database.

Note

This module supports password stacking, password hashing and unauthenticated identity.

The `DatabaseServerLoginModule` is based on two logical tables:

```
Table Principals(PrincipalID text, Password text)
Table Roles(PrincipalID text, Role text, RoleGroup text)
```

The `Principals` table associates the user `PrincipalID` with the valid password and the `Roles` table associates the user `PrincipalID` with its role sets. The roles used for user permissions must be contained in rows with a `RoleGroup` column value of `Roles`.

The tables are logical in that you can specify the SQL query that the login module uses. The only requirement is that the `java.sql.ResultSet` has the same logical structure as the `Principals` and `Roles` tables described previously. The actual names of the tables and columns are not relevant as the results are accessed based on the column index.

To clarify this notion, consider a database with two tables, `Principals` and `Roles`, as already declared. The following statements populate the tables with the following data:

- `PrincipalIDjava` with a `Password` of `echoman` in the `Principals` table
- `PrincipalIDjava` with a role named `Echo` in the `RolesRoleGroup` in the `Roles` table

- `PrincipalIDjava` with a role named `caller_java` in the `CallerPrincipalRoleGroup` in the `Roles` table

```
INSERT INTO Principals VALUES('java', 'echoman')
INSERT INTO Roles VALUES('java', 'Echo', 'Roles')
INSERT INTO Roles VALUES('java', 'caller_java', 'CallerPrincipal')
```

The supported login module configuration options include the following:

dsJndiName
: The JNDI name for the `DataSource` of the database containing the logical `Principals` and `Roles` tables. If not specified this defaults to `java:/DefaultDS`.

principalsQuery
: The prepared statement query equivalent to: `select Password from Principals where PrincipalID=?`. If not specified this is the exact prepared statement that will be used.

rolesQuery
: The prepared statement query equivalent to: `select Role, RoleGroup from Roles where PrincipalID=?`. If not specified this is the exact prepared statement that will be used.

ignorePasswordCase
: A boolean flag indicating if the password comparison should ignore case. This can be useful for hashed password encoding where the case of the hashed password is not significant.

An example `DatabaseServerLoginModule` configuration could be constructed as follows:

```
CREATE TABLE Users(username VARCHAR(64) PRIMARY KEY, passwd VARCHAR(64))
CREATE TABLE UserRoles(username VARCHAR(64), userRoles VARCHAR(32))
```

A corresponding `login-config.xml` entry would be:

```
<policy>
    <application-policy name="testDB">
        <authentication>
            <login-module code="org.jboss.security.auth.spi.DatabaseServerLoginModule"
                             flag="required">
                <module-option name="dsJndiName">java:/MyDatabaseDS</module-option>
                <module-option name="principalsQuery">
                    select passwd from Users username where username=?</module-option>
            <module-option name="rolesQuery">
               select userRoles, 'Roles' from UserRoles where username=?</module-option>
         </login-module>
      </authentication>
   </application-policy>
</policy>
```

10.1.7. LdapLoginModule

`LdapLoginModule` is a `LoginModule` implementation that authenticates against an LDAP server. Use the `LdapLoginModule` if your username and credentials are stored in an LDAP server that is accessible using a JNDI LDAP provider.

> **Note**
>
> This login module also supports unauthenticated identity and password stacking.

The LDAP connectivity information is provided as configuration options that are passed through to the environment object used to create JNDI initial context. The standard LDAP JNDI properties used include the following:

java.naming.factory.initial
: `InitialContextFactory` implementation class name. This defaults to the Sun LDAP provider implementation `com.sun.jndi.ldap.LdapCtxFactory`.

java.naming.provider.url
: LDAP URL for the LDAP server.

java.naming.security.authentication
: Security level to use. This defaults to `simple`.

java.naming.security.protocol
: Transport protocol to use for secure access, such as, SSL.

java.naming.security.principal
: Principal for authenticating the caller to the service. This is built from other properties as described below.

java.naming.security.credentials
: Authentication scheme to use. For example, hashed password, clear-text password, key, certificate, and so on.

The supported login module configuration options include the following:

principalDNPrefix
: Prefix added to the username to form the user distinguished name. See `principalDNSuffix` for more info.

principalDNSuffix
: Suffix added to the username when forming the user distinguished name. This is useful if you prompt a user for a username and you don't want the user to have to enter the fully distinguished name. Using this property and `principalDNSuffix` the `userDN` will be formed as `principalDNPrefix + username + principalDNSuffix`

useObjectCredential
: Value that indicates the credential should be obtained as an opaque `Object` using the `org.jboss.security.auth.callback.ObjectCallback` type of `Callback` rather than as a `char[]` password using a JAAS `PasswordCallback`. This allows for passing non-`char[]` credential information to the LDAP server. The available values are `true` and `false`.

rolesCtxDN
: Fixed, distinguished name to the context to search for user roles.

userRolesCtxDNAttributeName
: Name of an attribute in the user object that contains the distinguished name to the context to search for user roles. This differs from `rolesCtxDN` in that the context to search for a user's roles can be unique for each user.

roleAttributeID
: Name of the attribute containing the user roles. If not specified, this defaults to `roles`.

roleAttributeIsDN
: Flag indicating whether the `roleAttributeID` contains the fully distinguished name of a role object, or the role name. The role name is taken from the value of the `roleNameAttributeId` attribute of the context name by the distinguished name.

  If true, the role attribute represents the distinguished name of a role object. If false, the role name is taken from the value of `roleAttributeID`. The default is `false`.

  > **Note**
  >
  > In certain directory schemas (e.g., MS ActiveDirectory), role attributes in the user object are stored as DNs to role objects instead of simple names. For implementations that use this schema type, roleAttributeIsDN must be set to true.

roleNameAttributeID
: Name of the attribute of the context pointed to by the `roleCtxDN` distinguished name value which contains the role name. If the `roleAttributeIsDN` property is set to true, this property is used to find the role object's name attribute. The default is `group`.

uidAttributeID
: Name of the attribute in the object containing the user roles that corresponds to the userid. This is used to locate the user roles. If not specified this defaults to `uid`.

matchOnUserDN
: Flag that specifies whether the search for user roles should match on the user's fully distinguished name. If true, the full `userDN` is used as the match value. If false, only the username is used as the match value against the `uidAttributeName` attribute. The default value is `false`.

allowEmptyPasswords
: A flag indicating if empty (length 0) passwords should be passed to the LDAP server. An empty password is treated as an anonymous login by some LDAP servers and this may not be a desirable feature. To reject empty passwords, set this to `false`. If set to `true`, the LDAP server will validate the empty password. The default is `true`.

User authentication is performed by connecting to the LDAP server, based on the login module configuration options. Connecting to the LDAP server is done by creating an `InitialLdapContext` with an environment composed of the LDAP JNDI properties described previously in this section. The Context.SECURITY_PRINCIPAL is set to the distinguished name of the user as obtained by the callback handler in combination with the principalDNPrefix and principalDNSuffix option values, and the Context.SECURITY_CREDENTIALS property is either set to the `String` password or the `Object` credential depending on the useObjectCredential option.

Once authentication has succeeded (`InitialLdapContext` instance is created), the user's roles are queried by performing a search on the `rolesCtxDN` location with search attributes set to the roleAttributeName and uidAttributeName option values. The roles names are obtaining by invoking the `toString` method on the role attributes in the search result set.

Example 10.4. login-config.xml Sample

The following is a sample `login-config.xml` entry.

```
<application-policy name="testLDAP">
  <authentication>
    <login-module code="org.jboss.security.auth.spi.LdapLoginModule"
                 flag="required">
      <module-option name="java.naming.factory.initial">
        com.sun.jndi.ldap.LdapCtxFactory
        </module-option>
      <module-option name="java.naming.provider.url">
        ldap://ldaphost.jboss.org:1389/
      </module-option>
      <module-option name="java.naming.security.authentication">
        simple
```

```
            </module-option>
            <module-option name="principalDNPrefix">uid=</module-option>
            <module-option name="principalDNSuffix">
              ,ou=People,dc=jboss,dc=org
            </module-option>

            <module-option name="rolesCtxDN">
              ou=Roles,dc=jboss,dc=org
            </module-option>
            <module-option name="uidAttributeID">member</module-option>
            <module-option name="matchOnUserDN">true</module-option>

            <module-option name="roleAttributeID">cn</module-option>
            <module-option name="roleAttributeIsDN">false </module-option>
          </login-module>
        </authentication>
      </application-policy>
```

An LDIF file representing the structure of the directory this data operates against is shown below.

Example 10.5. LDIF File Example

```
dn: dc=jboss,dc=org
objectclass: top
objectclass: dcObject
objectclass: organization
dc: jboss
o: JBoss

dn: ou=People,dc=jboss,dc=org
objectclass: top
objectclass: organizationalUnit
ou: People

dn: uid=jduke,ou=People,dc=jboss,dc=org
objectclass: top
objectclass: uidObject
objectclass: person
uid: jduke
cn: Java Duke
sn: Duke
userPassword: theduke
```

```
dn: ou=Roles,dc=jboss,dc=org
objectclass: top
objectclass: organizationalUnit
ou: Roles

dn: cn=JBossAdmin,ou=Roles,dc=jboss,dc=org
objectclass: top
objectclass: groupOfNames
cn: JBossAdmin
member: uid=jduke,ou=People,dc=jboss,dc=org
description: the JBossAdmin group
```

The `java.naming.factory.initial`, `java.naming.factory.url` and `java.naming.security` options in the `testLDAP` login module configuration indicate the following conditions:

- The Sun LDAP JNDI provider implementation will be used
- The LDAP server is located on host `ldaphost.jboss.org` on port 1389
- The LDAP simple authentication method will be use to connect to the LDAP server.

The login module attempts to connect to the LDAP server using a Distinguished Name (DN) representing the user it is trying to authenticate. This DN is constructed from the passed `principalDNPrefix`, the username of the user and the `principalDNSuffix` as described above. In *Example 10.5, "LDIF File Example"*, the username `jduke` would map to `uid=jduke,ou=People,dc=jboss,dc=org`.

Note

The example assumes the LDAP server authenticates users using the `userPassword` attribute of the user's entry (`theduke` in this example). Most LDAP servers operate in this manner, however if your LDAP server handles authentication differently you must ensure LDAP is configured according to your production environment requirements.

Once authentication succeeds, the roles on which authorization will be based are retrieved by performing a subtree search of the `rolesCtxDN` for entries whose `uidAttributeID` match the user. If `matchOnUserDN` is true, the search will be based on the full DN of the user. Otherwise the search will be based on the actual user name entered. In this example, the search is under `ou=Roles,dc=jboss,dc=org` for any entries that have a `member` attribute equal to `uid=jduke,ou=People,dc=jboss,dc=org`. The search would locate `cn=JBossAdmin` under the roles entry.

The search returns the attribute specified in the `roleAttributeID` option. In this example, the attribute is `cn`. The value returned would be `JBossAdmin`, so the jduke user is assigned to the `JBossAdmin` role.

A local LDAP server often provides identity and authentication services, but is unable to use authorization services. This is because application roles don't always map well onto LDAP groups, and LDAP administrators are often hesitant to allow external application-specific data in central LDAP servers. For this reason, the LDAP authentication module is often paired with another login module, such as the database login module, that can provide roles more suitable to the application being developed.

10.1.8. LdapExtLoginModule

The `org.jboss.security.auth.spi.LdapExtLoginModule` is an alternate ldap login module implementation that uses searches for locating both the user to bind as for authentication as well as the associated roles. The roles query will recursively follow distinguished names (DNs) to navigate a hierarchical role structure.

The `LoginModule` options include whatever options your LDAP JNDI provider supports. Examples of standard property names are:

- `Context.INITIAL_CONTEXT_FACTORY` = "java.naming.factory.initial"
- `Context.SECURITY_PROTOCOL` = "java.naming.security.protocol"
- `Context.PROVIDER_URL` = "java.naming.provider.url"
- `Context.SECURITY_AUTHENTICATION` = "java.naming.security.authentication"
- `Context.REFERRAL` = "java.naming.referral"

The authentication happens in 2 steps:

1. An initial bind to the ldap server is done using the bindDN and bindCredential options. The `bindDN` is some user with the ability to search both the `baseCtxDN` and `rolesCtxDN` trees for the user and roles. The user DN to authenticate against is queried using the filter specified by the `baseFilter` attribute (see the `baseFilter` option description for its syntax).
2. The resulting user DN is then authenticated by binding to ldap server using the user DN as the `InitialLdapContext` environment `Context.SECURITY_PRINCIPAL`. The `Context.SECURITY_CREDENTIALS` property is either set to the String password obtained by the callback handler.

If this is successful, the associated user roles are queried using the `rolesCtxDN`, `roleAttributeID`, `roleAttributeIsDN`, `roleNameAttributeID`, and `roleFilter` options.

The full module properties include:

- `baseCtxDN`: The fixed DN of the context to start the user search from.
- `bindDN`: The DN used to bind against the ldap server for the user and roles queries. This is some DN with read/search permissions on the `baseCtxDN` and `rolesCtxDN` values.
- `bindCredential`: The password for the `bindDN`. This can be encrypted if the `jaasSecurityDomain` is specified.
- `jaasSecurityDomain`: The JMX ObjectName of the `JaasSecurityDomain` to use to decrypt the `java.naming.security.principal`. The encrypted form of the password is that returned by the `JaasSecurityDomain.encrypt64(byte[])` method. The `org.jboss.security.plugins.PBEUtils` class can also be used to generate the encrypted form.
- `baseFilter`: A search filter used to locate the context of the user to authenticate. The input username/userDN as obtained from the login module callback will be substituted into the filter anywhere a `{0}` expression is seen. This substitution behavior comes from the standard `DirContext.search(Name, String, Object[], SearchControls cons)` method. A common example for the search filter is `(uid={0})`.
- `rolesCtxDN`: The fixed DN of the context to search for user roles. Consider that this is not the Distinguished Name of where the actual roles are; rather, this is the DN of where the objects containing the user roles are (for example, for Active Directory, this is the DN where the user account is).
- `roleFilter`: A search filter used to locate the roles associated with the authenticated user. The input username/userDN as obtained from the login module callback will be substituted into the filter anywhere a `{0}` expression is seen. The authenticated `userDN` will be substituted into the filter anywhere a `{1}` is seen. An example search filter that matches on the input username is: `(member={0})`. An alternative that matches on the authenticated `userDN` is: `(member={1})`.
- `roleAttributeIsDN`: A flag indicating whether the user's role attribute contains the fully distinguished name of a role object, or the users's role attribute contains the role name. If false, the role name is taken from the value of the user's role attribute. If true, the role attribute represents the distinguished name of a role object. The role name is taken from the value of the `roleNameAttributeId` attribute of the corresponding object. In certain directory schemas (for example, Microsoft Active Directory), role *(group)attributes* in the user object are stored as DNs to role objects instead of as simple names, in which case, this property should be set to true. The default value of this property is false.
- `roleAttributeID`: The name of the role attribute of the context which corresponds to the name of the role. If the `roleAttributeIsDN` property is set to true, this property is the DN of the context to query for the `roleNameAttributeID` attribute. If the `roleAttributeIsDN` property is set to false, this property is the attribute name of the role name.
- `roleNameAttributeID`: The name of the role attribute of the context which corresponds to the name of the role. If the `roleAttributeIsDN` property is set to true, this property is used to find

the role object's name attribute. If the `roleAttributeIsDN` property is set to false, this property is ignored.

- `distinguishedNameAttribute`: The name of an attribute in the user entry that contains the DN of the user. This may be necessary if the DN of the user itself contains special characters (backslash for example) that may prevent correct user mapping. Defaults to distinguishedName. If there is no such attribute, the entry's DN will be used.
- `roleRecursion` : How deep the role search will go below a given matching context. Disable with 0, which is the default.
- `searchTimeLimit`: The timeout in milliseconds for the user/role searches. Defaults to 10000 (10 seconds).
- `searchScope`: Sets the search scope to one of the strings. The default is `SUBTREE_SCOPE`.
  - `OBJECT_SCOPE`: only search the named roles context.
  - `ONELEVEL_SCOPE`: search directly under the named roles context.
  - `SUBTREE_SCOPE`: If the roles context is not a DirContext, search only the object. If the roles context is a *DirContext*, search the subtree rooted at the named object, including the named object itself
- `allowEmptyPasswords`: A flag indicating if `empty(length==0)` passwords should be passed to the LDAP server. An empty password is treated as an anonymous login by some LDAP servers and this may not be a desirable feature. Set this to false to reject empty passwords, true to have the ldap server validate the empty password. The default is true.

10.1.9. BaseCertLoginModule

`BaseCertLoginModule` authenticates users based on X509 certificates. A typical use case for this login module is `CLIENT-CERT` authentication in the web tier.

This login module only performs authentication: you must combine it with another login module capable of acquiring authorization roles to completely define access to a secured web or EJB component. Two subclasses of this login module, `CertRolesLoginModule` and `DatabaseCertLoginModule` extend the behavior to obtain the authorization roles from either a properties file or database.

The `BaseCertLoginModule` needs a `KeyStore` to perform user validation. This is obtained through a `org.jboss.security.SecurityDomain` implementation. Typically, the `SecurityDomain` implementation is configured using the `org.jboss.security.plugins.JaasSecurityDomain` MBean as shown in this `jboss-service.xml` configuration fragment:

```
<mbean code="org.jboss.security.plugins.JaasSecurityDomain"
       name="jboss.ch8:service=SecurityDomain">
  <constructor>
```

```
      <arg type="java.lang.String" value="jmx-console"/>
    </constructor>
    <attribute name="KeyStoreURL">resource:localhost.keystore</attribute>
    <attribute name="KeyStorePass">unit-tests-server</attribute>
</mbean>
```

The configuration creates a security domain with the name `jmx-console`, with a `SecurityDomain` implementation available through JNDI under the name `java:/jaas/jmx-console`. The security domain follows the JBossSX security domain naming pattern.

Procedure 10.1. Secure Web Applications with Certificates and Role-based Authorization

This procedure describes how to secure a web application, such as the `jmx-console.war`, using client certificates and role-based authorization.

1. **Declare Resources and Roles**

   Modify `web.xml` to declare the resources to be secured along with the allowed roles and security domain to be used for authentication and authorization.

```
<?xml version="1.0"?>
<web-app version="2.5"
   xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee   http://java.sun.com/xml/ns/
javaee/web-app_2_5.xsd">

 ...
   <!-- A security constraint that restricts access to the HTML JMX console
   to users with the role JBossAdmin. Edit the roles to what you want and
   uncomment the WEB-INF/jboss-web.xml/security-domain element to enable
   secured access to the HTML JMX console.
   -->
   <security-constraint>
     <web-resource-collection>
       <web-resource-name>HtmlAdaptor</web-resource-name>
       <description>An example security config that only allows users with the
         role JBossAdmin to access the HTML JMX console web application
       </description>
       <url-pattern>/*</url-pattern>
     </web-resource-collection>
     <auth-constraint>
```

```
       <role-name>JBossAdmin</role-name>
     </auth-constraint>
   </security-constraint>

   <login-config>
      <auth-method>BASIC</auth-method>
      <realm-name>JBoss JMX Console</realm-name>
   </login-config>

   <security-role>
      <role-name>JBossAdmin</role-name>
   </security-role>
</web-app>
```

2. **Specify the JBoss Security Domain**

   In the `jboss-web.xml` file, specify the required security domain.

```
<jboss-web>
    <security-domain>jmx-console</security-domain>
</jboss-web>
```

3. **Specify Login Module Configuration**

   Define the login module configuration for the jmx-console security domain you just specified. This is done in the `conf/login-config.xml` file.

```
<application-policy name="jmx-console">
    <authentication>
        <login-module code="org.jboss.security.auth.spi.BaseCertLoginModule"
                      flag="required">
            <module-option name="password-stacking">useFirstPass</module-option>
            <module-option name="securityDomain">jmx-console</module-option>
        </login-module>
        <login-module code="org.jboss.security.auth.spi.UsersRolesLoginModule"
                      flag="required">
            <module-option name="password-stacking">useFirstPass</module-option>
                <module-option  name="usersProperties">jmx-console-users.properties</module-
option>
                  <module-option  name="rolesProperties">jmx-console-roles.properties</module-
option>
```

```
        </login-module>
      </authentication>
    </application-policy>
```

*Secure Web Applications with Certificates and Role-based Authorization*shows the `BaseCertLoginModule` is used for authentication of the client cert, and the `UsersRolesLoginModule` is only used for authorization due to the `password-stacking=useFirstPass` option. Both the `localhost.keystore` and the `jmx-console-roles.properties` require an entry that maps to the principal associated with the client cert.

By default, the principal is created using the client certificate distinguished name, such as the DN specified in *Example 10.6, “Certificate Example”*.

Example 10.6. Certificate Example

```
[conf]$ keytool -printcert -file unit-tests-client.export
Owner: CN=unit-tests-client, OU=JBoss Inc., O=JBoss Inc., ST=Washington,
 C=US
Issuer: CN=jboss.com, C=US, ST=Washington, L=Snoqualmie Pass,
 EMAILADDRESS=admin
@jboss.com, OU=QA, O=JBoss Inc.
Serial number: 100103
Valid from: Wed May 26 07:34:34 PDT 2004 until: Thu May 26 07:34:34 PDT 2005
Certificate fingerprints:
         MD5:  4A:9C:2B:CD:1B:50:AA:85:DD:89:F6:1D:F5:AF:9E:AB
         SHA1: DE:DE:86:59:05:6C:00:E8:CC:C0:16:D3:C2:68:BF:95:B8:83:E9:58
```

The `localhost.keystore` would need the certificate in *Example 10.6, “Certificate Example”* stored with an alias of `CN=unit-tests-client, OU=JBoss Inc., O=JBoss Inc., ST=Washington, C=US`. The `jmx-console-roles.properties` would also need an entry for the same entry. Since the DN contains characters that are normally treated as delimiters, you must escape the problem characters using a backslash ('\') as illustrated below.

```
# A sample roles.properties file for use with the UsersRolesLoginModule
CN\=unit-tests-client,\ OU\=JBoss\ Inc.,\ O\=JBoss\ Inc.,\ ST\=Washington,\
 C\=US=JBossAdmin
admin=JBossAdmin
```

10.1.10. IdentityLoginModule

`IdentityLoginModule` is a simple login module that associates a hard-coded user name to any subject authenticated against the module. It creates a `SimplePrincipal` instance using the name specified by the `principal` option.

> **Note**
>
> This module supports password stacking.

This login module is useful when you need to provide a fixed identity to a service, and in development environments when you want to test the security associated with a given principal and associated roles.

The supported login module configuration options include:

principal
: This is the name to use for the `SimplePrincipal` all users are authenticated as. The principal name defaults to `guest` if no principal option is specified.

roles
: This is a comma-delimited list of roles that will be assigned to the user.

A sample XMLLoginConfig configuration entry is described below. The entry authenticates all users as the principal named `jduke` and assign role names of `TheDuke`, and `AnimatedCharacter`:

```
<policy>
    <application-policy name="testIdentity">
        <authentication>
            <login-module code="org.jboss.security.auth.spi.IdentityLoginModule"
                          flag="required">
                <module-option name="principal">jduke</module-option>
                <module-option name="roles">TheDuke,AnimatedCharacter</module-option>
            </login-module>
        </authentication>
    </application-policy>
</policy>
```

10.1.11. RunAsLoginModule

`RunAsLoginModule` (`org.jboss.security.auth.spi.RunAsLoginModule`) is a helper module that pushes a run as role onto the stack for the duration of the login phase of authentication, and pops the run as role in either the commit or abort phase.

The purpose of this login module is to provide a role for other login modules that must access secured resources in order to perform their authentication (for example, a login module that accesses a secured EJB). `RunAsLoginModule` must be configured ahead of the login modules that require a run as role established.

The only login module configuration option is:

roleName
: Name of the role to use as the run as role during login phase. If not specified a default of `nobody` is used.

10.1.12. ClientLoginModule

`ClientLoginModule` (`org.jboss.security.ClientLoginModule`) is an implementation of `LoginModule` for use by JBoss clients for establishing caller identity and credentials. This simply sets the principal to the value of the `NameCallback` filled in by the `callbackhandler`, and the credential to the value of the `PasswordCallback` filled in by the `callbackhandler` in the security context.

`ClientLoginModule` is the only supported mechanism for a client to establish the current thread's caller. Both stand-alone client applications, and server environments (acting as JBoss EJB clients where the security environment has not been configured to use JBossSX transparently) must use `ClientLoginModule`.

Note that this login module does not perform any authentication. It merely copies the login information provided to it into the JBoss server EJB invocation layer for subsequent authentication on the server. If you need to perform client-side authentication of users you would need to configure another login module in addition to the `ClientLoginModule`.

The supported login module configuration options include the following:

multi-threaded
: Value that specifies the way login threads connect to principal and credential storage sources. When set to true, each login thread has its own principal and credential storage and each separate thread must perform its own login. This is useful in client environments where multiple user identities are active in separate threads. When set to false the login identity and credentials are global variables that apply to all threads in the VM. The default setting is `false`.

restore-login-identity
: Value that specifies whether the `SecurityAssociation` principal and credential seen on entry to the `login()` method are saved and restored on either abort or logout. This is necessary if you must change identities and then restore the original caller identity. If set to `true`, the principal and credential information is saved and restored on abort or logout. If set to `false`, abort and logout clear the `SecurityAssociation`. The default value is `false`.

10.2. Custom Modules

If the login modules bundled with the JBossSX framework do not work with your security environment, you can write your own custom login module implementation. The `JaasSecurityManager` requires a particular usage pattern of the `Subject` principals set. You must understand the JAAS Subject class's information storage features and the expected usage of these features to write a login module that works with the `JaasSecurityManager`.

This section examines this requirement and introduces two abstract base `LoginModule` implementations that can help you implement custom login modules.

You can obtain security information associated with a `Subject` by using the following methods:

```
java.util.Set getPrincipals()
java.util.Set getPrincipals(java.lang.Class c)
java.util.Set getPrivateCredentials()
java.util.Set getPrivateCredentials(java.lang.Class c)
java.util.Set getPublicCredentials()
java.util.Set getPublicCredentials(java.lang.Class c)
```

For `Subject` identities and roles, JBossSX has selected the most logical choice: the principals sets obtained via `getPrincipals()` and `getPrincipals(java.lang.Class)`. The usage pattern is as follows:

- User identities (for example; username, social security number, employee ID) are stored as `java.security.Principal` objects in the `Subject Principals` set. The `Principal` implementation that represents the user identity must base comparisons and equality on the name of the principal. A suitable implementation is available as the `org.jboss.security.SimplePrincipal` class. Other `Principal` instances may be added to the `SubjectPrincipals` set as needed.
- Assigned user roles are also stored in the `Principals` set, and are grouped in named role sets using `java.security.acl.Group` instances. The `Group` interface defines a collection of `Principals` and/or `Groups`, and is a subinterface of `java.security.Principal`.
- Any number of role sets can be assigned to a `Subject`.
- The JBossSX framework uses two well-known role sets with the names `Roles` and `CallerPrincipal`.
  - The `Roles` group is the collection of `Principals` for the named roles as known in the application domain under which the `Subject` has been authenticated. This role set is used by methods like the `EJBContext.isCallerInRole(String)`, which EJBs can use to see if the current caller belongs to the named application domain role. The security interceptor logic that performs method permission checks also uses this role set.
  - The `CallerPrincipalGroup` consists of the single `Principal` identity assigned to the user in the application domain. The `EJBContext.getCallerPrincipal()` method uses the `CallerPrincipal` to allow the application domain to map from the operation environment identity to a user identity suitable for the application. If a `Subject` does not have a `CallerPrincipalGroup`, the application identity is the same used for login.

10.2.1. Custom LoginModule Example

The following information will help you to create a custom Login Module example that extends the `UsernamePasswordLoginModule` and obtains a user's password and role names from a JNDI lookup.

At the end of this section you will have created a custom JNDI context login module that will return a user's password if you perform a lookup on the context using a name of the form `password/<username>` (where `<username>` is the current user being authenticated). Similarly, a lookup of the form `roles/<username>` returns the requested user's roles.

Example 10.7, “ JndiUserAndPass Custom Login Module” shows the source code for the `JndiUserAndPass` custom login module.

Note that because this extends the JBoss `UsernamePasswordLoginModule`, all `JndiUserAndPass` does is obtain the user's password and roles from the JNDI store. The `JndiUserAndPass` does not interact with the JAAS `LoginModule` operations.

Example 10.7. JndiUserAndPass Custom Login Module

```
package org.jboss.book.security.ex2;

import java.security.acl.Group;
import java.util.Map;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.security.auth.Subject;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.login.LoginException;

import org.jboss.security.SimpleGroup;
import org.jboss.security.SimplePrincipal;
import org.jboss.security.auth.spi.UsernamePasswordLoginModule;

/**
 * An example custom login module that obtains passwords and roles
 * for a user from a JNDI lookup.
 *
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.4 $
*/
public class JndiUserAndPass
    extends UsernamePasswordLoginModule
{
    /** The JNDI name to the context that handles the password/username lookup */
```

```
private String userPathPrefix;
/** The JNDI name to the context that handles the roles/ username lookup */
private String rolesPathPrefix;

/**
 * Override to obtain the userPathPrefix and rolesPathPrefix options.
 */
public void initialize(Subject subject, CallbackHandler callbackHandler,
                       Map sharedState, Map options)
{
    super.initialize(subject, callbackHandler, sharedState, options);
    userPathPrefix = (String) options.get("userPathPrefix");
    rolesPathPrefix = (String) options.get("rolesPathPrefix");
}

/**
 *  Get the roles the current user belongs to by querying the
 * rolesPathPrefix + '/' + super.getUsername() JNDI location.
 */
protected Group[] getRoleSets() throws LoginException
{
    try {
        InitialContext ctx = new InitialContext();
        String rolesPath = rolesPathPrefix + '/' + super.getUsername();

        String[] roles = (String[]) ctx.lookup(rolesPath);
        Group[] groups = {new SimpleGroup("Roles")};
        log.info("Getting roles for user="+super.getUsername());
        for(int r = 0; r < roles.length; r ++) {
            SimplePrincipal role = new SimplePrincipal(roles[r]);
            log.info("Found role="+roles[r]);
            groups[0].addMember(role);
        }
        return groups;
    } catch(NamingException e) {
        log.error("Failed to obtain groups for
                   user="+super.getUsername(), e);
        throw new LoginException(e.toString(true));
    }
}

/**
 * Get the password of the current user by querying the
 * userPathPrefix + '/' + super.getUsername() JNDI location.
```

```
      */
    protected String getUsersPassword()
        throws LoginException
    {
        try {
            InitialContext ctx = new InitialContext();
            String userPath = userPathPrefix + '/' + super.getUsername();
            log.info("Getting password for user="+super.getUsername());
            String passwd = (String) ctx.lookup(userPath);
            log.info("Found password="+passwd);
            return passwd;
        } catch(NamingException e) {
            log.error("Failed to obtain password for
                      user="+super.getUsername(), e);
            throw new LoginException(e.toString(true));
        }
    }
}
```

The details of the JNDI store are found in the `org.jboss.book.security.ex2.service.JndiStore` MBean. This service binds an `ObjectFactory` that returns a `javax.naming.Context` proxy into JNDI. The proxy handles lookup operations done against it by checking the prefix of the lookup name against `password` and `roles`.

When the name begins with `password`, a user's password is being requested. When the name begins with `roles` the user's roles are being requested. The example implementation always returns a password of `theduke` and an array of roles names equal to `{"TheDuke", "Echo"}` regardless of what the username is. You can experiment with other implementations as you wish.

The example code includes a simple session bean for testing the custom login module. To build, deploy and run the example, execute the following command in the examples directory.

```
[examples]$ ant -Dchap=security -Dex=2 run-example
...
run-example2:
     [echo] Waiting for 5 seconds for deploy...
     [java] [INFO,ExClient] Login with username=jduke, password=theduke
     [java] [INFO,ExClient] Looking up EchoBean2
     [java] [INFO,ExClient] Created Echo
     [java] [INFO,ExClient] Echo.echo('Hello') = Hello
```

The choice of using the `JndiUserAndPass` custom login module for the server side authentication of the user is determined by the login configuration for the example security domain. The EJB JAR `META-INF/jboss.xml` descriptor sets the security domain.

```
<?xml version="1.0"?>
<jboss>
  <security-domain>security-ex2</security-domain>
</jboss>
```

The SAR `META-INF/login-config.xml` descriptor defines the login module configuration.

```
<application-policy name = "security-ex2">
  <authentication>
    <login-module code="org.jboss.book.security.ex2.JndiUserAndPass"
             flag="required">
      <module-option name = "userPathPrefix">/security/store/password</module-option>
      <module-option name = "rolesPathPrefix">/security/store/roles</module-option>
    </login-module>
  </authentication>
</application-policy>
```

Part III. Encryption and Security

Java Security Manager

To restrict code privileges using Java permissions, you must configure the JBoss server to run under a security manager. This is done by configuring the Java VM options in the `run.conf` in the JBoss server distribution bin directory. The two required VM options are as follows:

java.security.manager
: Used without any value to specify that the default security manager should be used. This is the preferred security manager. You can also pass a value to the `java.security.manager` option to specify a custom security manager implementation. The value must be the fully qualified class name of a subclass of `java.lang.SecurityManager`. This form specifies that the policy file should augment the default security policy as configured by the VM installation.

java.security.policy
: Used to specify the policy file that will augment the default security policy information for the VM. This option takes two forms: `java.security.policy=policyFileURL` and `java.security.policy==policyFileURL`. The first form specifies that the policy file should augment the default security policy as configured by the VM installation. The second form specifies that only the indicated policy file should be used. The `policyFileURL` value can be any URL for which a protocol handler exists, or a file path specification.

Both the `run.bat` and `run.sh` start scripts reference a `JAVA_OPTS` variable specified in `run.conf` (Linux) or `run.conf.bat` (Windows) that sets the required security manager properties.

The next element of Java security is establishing the allowed permissions. If you look at the `$JBOSS_HOME/bin/server.policy.cert` file that is contained in the default configuration file set you will notice it contains the following grant statement:

```
grant signedBy "jboss" {
  permission java.security.AllPermission;
};
```

This statement declares that all code signed by JBoss is trusted. To import the public key to your keystore, follow *Activate Java Security Manager*

Important

Carefully consider what permissions you grant. Be particularly cautious about granting `java.security.AllPermission`: you can potentially allow changes to the system binary, including the JVM runtime environment.

The current set of JBoss specific `java.lang.RuntimePermissions` are described below.

`org.jboss.security.SecurityAssociation.getPrincipalInfo`
: Provides access to the `org.jboss.security.SecurityAssociation` `getPrincipal()` and `getCredential()` methods. The risk involved with using this runtime permission is the ability to see the current thread caller and credentials.

`org.jboss.security.SecurityAssociation.getSubject`
: Provides access to the `org.jboss.security.SecurityAssociation` `getSubject()` method.

`org.jboss.security.SecurityAssociation.setPrincipalInfo`
: Provides access to the `org.jboss.security.SecurityAssociation` `setPrincipal()`, `setCredential()`, `setSubject()`, `pushSubjectContext()`, and `popSubjectContext()` methods. The risk involved with using this runtime permission is the ability to set the current thread caller and credentials.

`org.jboss.security.SecurityAssociation.setServer`
: Provides access to the `org.jboss.security.SecurityAssociation` `setServer` method. The risk involved with using this runtime permission is the ability to enable or disable multithread storage of the caller principal and credential.

`org.jboss.security.SecurityAssociation.setRunAsRole`
: Provides access to the `org.jboss.security.SecurityAssociation` `pushRunAsRole` and `popRunAsRole`, `pushRunAsIdentity` and `popRunAsIdentity` methods. The risk involved with using this runtime permission is the ability to change the current caller run-as role principal.

`org.jboss.security.SecurityAssociation.accessContextInfo`
: Provides access to the `org.jboss.security.SecurityAssociation` `accessContextInfo, "Get"` and `accessContextInfo, "Set"` methods, allowing you to both set and get the current security context info.

`org.jboss.naming.JndiPermission`
: Provides special permissions to files and directories in a specified JNDI tree path, or recursively to all files and subdirectories. A JndiPermission consists of a pathname and a set of valid permissions related to the file or directory.

  The available permissions include: `bind`, `rebind`, `unbind`, `lookup`, `list`, `listBindings`, `createSubcontext`, and `all`.

  Pathnames ending in `/*` indicate the specified permissions apply to all files and directories of the pathname. Pathnames ending in `/-` indicate recursive permissions to all files and subdirectories of the pathname. Pathnames consisting of the special token `<<ALL BINDINGS>>` matches any file in any directory.

`org.jboss.security.srp.SRPPermission`
: A custom permission class for protecting access to sensitive SRP information like the private session key and private key. This permission doesn't have any actions defined. The getSessionKey target provides access to the private session key resulting from the SRP

negotiation. Access to this key will allow you to encrypt and decrypt messages that have been encrypted with the session key.

`org.hibernate.secure.HibernatePermission`
: This permission class provides basic permissions to secure Hibernate sessions. The target for this property is the entity name. The available actions include: insert, delete, update, read, * (all).

`org.jboss.metadata.spi.stack.MetaDataStackPermission`
: Provides a custom permission class for controlling how callers interact with the metadata stack. The available permissions are: `modify` (push/pop onto the stack), `peek` (peek onto the stack), and `*` (all).

`org.jboss.config.spi.ConfigurationPermission`
: Secures setting of configuration properties. Defines only permission target names, and no actions. The targets for this property include: <property name> - property which code has permission to set; * - all properties.

`org.jboss.kernel.KernelPermission`
: Secures access to the kernel configuration. Defines only permission target names and no actions. The targets for this property include: access - access the kernel configuration; configure - configure the kernel (access is implied); * - all of the above.

`org.jboss.kernel.plugins.util.KernelLocatorPermission`
: Secures access to the kernel. Defines only permission target names and no actions. The targets for this property include: kernel - access the kernel; * - access all areas.

Procedure 11.1. Activate Java Security Manager

Follow this procedure to correctly configure the JSM for secure, production-ready operation. This procedure is only required while configuring your server for the first time. In this procedure, `$JAVA_HOME` refers to the installation directory of the JRE.

1. **Import public key to keystore**

   Execute the following command:

   Linux.

   ```
   [home]$ sudo $JAVA_HOME/bin/keytool -import  -alias jboss -file
    JBossPublicKey.RSA -keystore $JAVA_HOME/jre/lib/security/cacerts
   ```

   Windows.

   ```
   C:\> %JAVA_HOME%\bin\keytool -import  -alias jboss -file
    JBossPublicKey.RSA -keystore %JAVA_HOME%\jre\lib\security\cacerts
   ```

2. **Verify key signature**

   Execute the following command in the terminal.

   > **Note**
   >
   > The default JVM Keystore password is `changeit`.

   ```
   $ keytool -list -keystore $JAVA_HOME/jre/lib/security/cacerts
   Enter keystore password:

   Keystore type: JKS
   Keystore provider: SUN

   Your keystore contains 2 entries

   jboss, Aug 12, 2009, trustedCertEntry,
   Certificate fingerprint (MD5):
    93:F2:F1:8B:EF:8A:E0:E3:D0:E7:69:BC:69:96:29:C1
   jbosscodesign2009, Aug 12, 2009, trustedCertEntry,
   Certificate fingerprint (MD5):
    93:F2:F1:8B:EF:8A:E0:E3:D0:E7:69:BC:69:96:29:C1
   ```

3. **Specify additional JAVA_OPTS**

   Linux. Ensure the following block is present in the `$JBOSS_HOME/server/$PROFILE/run.conf` file.

   ```
   ## Specify the Security Manager options
   JAVA_OPTS="$JAVA_OPTS -Djava.security.manager -Djava.security.policy==$DIRNAME/
   server.policy.cert
   -Djava.protocol.handler.pkgs=org.jboss.handlers.stub
   -Djava.security.debug=access:failure
   -Djboss.home.dir=$DIRNAME/../
   -Djboss.server.home.dir=$DIRNAME/../server/default/"
   ```

   > **Note**
   >
   > Placing `run.conf` into the target profile directory will mean the file overrides any other `run.conf` files outside server profiles.

   Windows. Ensure the following block is present in the `$JBOSS_HOME\bin\run.conf.bat` file.

```
rem # Specify the Security Manager options
set "JAVA_OPTS=%JAVA_OPTS% -Djava.security.manager
-Djava.security.policy==%DIRNAME%\server.policy.cert
-Djava.protocol.handler.pkgs=org.jboss.handlers.stub
-Djava.security.debug=access:failure
-Djboss.home.dir=%DIRNAME%/../
-Djboss.server.home.dir=%DIRNAME%/../server/default/"
```

4. **Start the server**

   Start JBoss using the `run.sh` or `run.bat` (Windows) script.

A number of Java debugging flags are available to assist you in determining how the security manager is using your security policy file, and what policy files are contributing permissions. Running the VM as follows shows the possible debugging flag settings:

```
[bin]$ java -Djava.security.debug=help

all            turn on all debugging
access         print all checkPermission results
combiner       SubjectDomainCombiner debugging
configfile     JAAS ConfigFile loading
configparser   JAAS ConfigFile parsing
gssloginconfig GSS LoginConfigImpl debugging
jar            jar verification
logincontext   login context results
policy         loading and granting
provider       security provider debugging
scl            permissions SecureClassLoader assigns

The following can be used with access:

stack         include stack trace
domain        dump all domains in context
failure       before throwing exception, dump stack
              and domain that didn't have permission

The following can be used with stack and domain:

permission=<classname>
              only dump output if specified permission
              is being checked
codebase=<URL>
              only dump output if specified codebase
              is being checked
```

Running with `-Djava.security.debug=all` provides the most output, but the output volume is acutely verbose. This might be a good place to start if you don't understand a given security failure at all. For less verbose output that will still assist with debugging permission failures, use `-Djava.security.debug=access,failure`.

Encrypting EJB connections with SSL

JBoss Application Server uses a socket-based invoker layer for Remote Method Invocation (RMI) of EJB2 and EJB3 Beans. This network traffic is not encrypted by default. Follow the instructions in this chapter to use Secure Sockets Layer (SSL) to encrypt this network traffic.

Procedure 12.1. Configure SSL for EJB3 Overview

1. Generate encryption keys and certificate
2. Configure a secure remote connector
3. Annotate EJB3 beans that will use the secure connector

Procedure 12.2. Configure SSL for EJB2 Overview

1. Generate encryption keys and certificate
2. Configure Unified Invoker for SSL

12.1. SSL Encryption overview

12.1.1. Key pairs and Certificates

Secure Sockets Layer (SSL) encrypts network traffic between two systems. Traffic between the two systems is encrypted using a two-way key, generated during the *handshake* phase of the connection and known only by those two systems.

For secure exchange of the two-way encryption key, SSL makes use of Public Key Infrastructure (PKI), a method of encryption that utilizes a *key pair*. A key pair consists of two separate but matching cryptographic keys - a public key and a private key. The public key is shared with others and is used to encrypt data, and the private key is kept secret and is used to decrypt data that has been encrypted using the public key. When a client requests a secure connection a handshake phase takes place before secure communication can begin. During the SSL handshake the server passes its public key to the client in the form of a certificate. The certificate contains the identity of the server (its URL), the public key of the server, and a digital signature that validates the certificate. The client then validates the certificate and makes a decision about whether the certificate is trusted or not. If the certificate is trusted, the client generates the two-way encryption key for the SSL connection, encrypts it using the public key of the server, and sends it back to the server. The server decrypts the two-way encryption key, using its private key, and further communication between the two machines over this connection is encrypted using the two-way encryption key.

On the server, public/private key pairs are stored in a *key store*, an encrypted file that stores key pairs and trusted certificates. Each key pair within the key store is identified by an *alias* - a unique name that is used when storing or requesting a key pair from the key store. The public key is distributed to clients in the form of a *certificate*, a digital signature which binds together a public key and an identity. On the client, certificates of known validity are kept in the default key store known as a *trust store*.

CA-signed and self-signed certificates. Public Key Infrastructure relies on a chain of trust to establish the credentials of unknown machines. The use of public keys not only encrypts traffic between machines, but also functions to establish the identity of the machine at the other end of a network connection. A "Web of Trust" is used to verify the identity of servers. A server may be unknown to you, but if its public key is signed by someone that you trust, you extend that trust to the server. Certificate Authorities are commercial entities who verify the identity of customers and issue them signed certificates. The JDK includes a `cacerts` file with the certificates of several trusted Certificate Authorities (CAs). Any keys signed by these CAs will be automatically trusted. Large organizations may have their own internal Certificate Authority, for example using Red Hat Certificate System. In this case the signing certificate of the internal Certificate Authority is typically installed on clients as part of a Corporate Standard Build, and then all certificates signed with that certificate are trusted. CA-signed certificates are best practice for production scenarios.

During development and testing, or for small-scale or internal-only production scenarios, you may use a *self-signed certificate*. This is certificate that is not signed by a Certificate Authority, but rather with a locally generated certificate. Since a locally generated certificate is not in the `cacerts` file of clients, you need to export a certificate for that key on the server, and import that certificate on any client that will connect via SSL.

The JDK includes `keytool`, a command line tool for generating key pairs and certificates. The certificates generated by `keytool` can be sent for signing by a CA or can be distributed to clients as a self-signed certificate.

- Generating a self-signed certificate for development use and importing that certificate to a client is described in *Section 12.2.1, “Generate a self-signed certificate with keytool”*.
- Generating a certificate and having it signed by a CA for production use is beyond the scope of this edition. Refer to the manpage for keytool for further information on performing this task.

12.2. Generate encryption keys and certificate

12.2.1. Generate a self-signed certificate with keytool

12.2.1.1. Generate a keypair

The **keytool** command, part of the JDK, is used to generate a new key pair. Keytool can either add the new key pair to an existing key store, or create a new key store at the same time as the key pair.

This key pair will be used to negotiate SSL encryption between the server and remote clients. The following procedure generates a key pair and stores it in a key store called `localhost.keystore`. You will need to make this key store available to the EJB3 invoker on the server. The key pair in our example will be saved in the key store under the alias 'ejb-ssl'. We will need this key alias, and the key pair password you supply (if any), when configuring the EJB3 Remoting connector in *Section 12.3.1, “Create a secure remoting connector for EJB3”*.

Procedure 12.3. Generate a new key pair and add it to the key store "localhost.keystore"

1. In your home directory, issue the following command, substituting a new password for *EJB-SSL_KEYPAIR_PASSWORD*:

   ```
   keytool -genkey -alias ejb-ssl -keypass EJB-SSL_KEYPAIR_PASSWORD
    -keystore localhost.keystore
   ```

2. Enter the key store password, if this key store already exists; otherwise enter and re-enter a password for a new key store that will be created.

3. Issue the command:

   ```
   dir
   ```

   Result: You should now see the file `localhost.keystore`.

> **Note**
>
> Key store files should be stored on a secure file system, and should be readable only by the owner of the JBoss Application Server process.

Note that if no key store is specified on the command line, `keytool` will add the key pair to a new key store called `keystore` in the current user's home directory. This key store file will be a hidden file.

12.2.1.2. Export a self-signed certificate

Once a key pair has been generated for the server to use, a certificate must be created. *Export a certificate* details the steps to export the `ejb-ssl` key from the key store named `localhost.keystore`.

Procedure 12.4. Export a certificate

1. Issue the following command:

```
keytool -export -alias ejb-ssl -file mycert.cer -keystore
 localhost.keystore
```

2. Enter the key store password

   Result: A certificate will be exported to the file `mycert.cer`.

12.2.2. Configure a client to accept a self-signed server certificate

Any client machine that will make remote method invocations over SSL needs to trust the certificate of the server. Since the certificate we generated is self-signed, and has no chain of trust to a known certificate authority, the client must be explicitly configured to trust the certificate or the connection will fail. Configuring a client to trust a self-signed certificate requires importing the self-signed server certificate to a trust store on the client.

A trust store is a key store that contains trusted certificates. Certificates that are in the local trust store will be accepted as valid. If your server uses a self-signed certificate then any clients that will make remote method calls over SSL must have that certificate in their trust store. You must export your public key as a certificate, and then import that certificate to the trust store on those clients.

The certificate created in *Section 12.2.1.2, “Export a self-signed certificate”* must be copied to the client in order to perform the steps detailed in *Import the certificate to the trust store "localhost.truststore"*.

Procedure 12.5. Import the certificate to the trust store "localhost.truststore"

1. Issue the following command on the client:

```
keytool -import -alias ejb-ssl -file mycert.cer -keystore
 localhost.truststore
```

2. Enter the password for this trust store if it already exists; otherwise enter and re-enter the password for the trust store that will be created.

3. Verify the details of the certificate, and if it is the correct one, type 'yes' to import it to the trust store.

   Result: The certificate will be imported to the trust store, and you will be able to establish a secure connection with a server that uses this certificate.

As with the key store, if the trust store specified does not already exist, it will be created. However, in contrast with the key store, there is no default trust store, and one must be specified.

Configure Client to use localhost.truststore. Now that you have imported the self-signed server certificate to a trust store on the client, you must instruct the client to use this trust store. This is done by passing the `localhost.truststore` location to the application using the `javax.net.ssl.trustStore` property, and the trust store password using the `javax.net.ssl.trustStorePassword` property. *Example 12.1, “Invoking the com.acme.Runclient application with a specific trust store”* is an example commandline that would be used to invoke the application **com.acme.RunClient**, which will make remote method calls to an EJB on a JBoss Application Server.

Example 12.1. Invoking the com.acme.Runclient application with a specific trust store

```
java -Djavax.net.ssl.trustStore=${resources}/localhost.truststore \
    -Djavax.net.ssl.trustStorePassword=TRUSTSTORE_PASSWORD
 com.acme.RunClient
```

12.3. EJB3 Configuration

12.3.1. Create a secure remoting connector for EJB3

The file `ejb3-connectors-jboss-beans.xml` in a JBoss Application Server profile's `deploy` directory contains JBoss Remoting connector definitions for EJB3 remote method invocation. *Example 12.2, “Sample Secure EJB3 Connector”* is a sample configuration that defines a secure connector for EJB3 using the key pair created in *Generate a new key pair and add it to the key store "localhost.keystore"*. The `keyPassword` property in the sample configuration is the key pair password that was specified when the key pair was created.

Example 12.2. Sample Secure EJB3 Connector

```
<bean name="EJB3SSLRemotingConnector"
 class="org.jboss.remoting.transport.Connector">
    <property
 name="invokerLocator">sslsocket://${jboss.bind.address}:3843</property>
    <property name="serverConfiguration">
      <inject bean="ServerConfiguration" />
    </property>
    <property name="serverSocketFactory">
      <inject bean="sslServerSocketFactory" />
    </property>
  </bean>

  <bean name="sslServerSocketFactory"
 class="org.jboss.security.ssl.DomainServerSocketFactory">
     <constructor>
        <parameter><inject bean="EJB3SSLDomain"/></parameter>
     </constructor>
```

```
   </bean>
   <bean name="EJB3SSLDomain"
  class="org.jboss.security.plugins.JaasSecurityDomain">
      <constructor>
         <parameter>EJB3SSLDomain</parameter>
      </constructor>
      <property name="keyStoreURL">resource:localhost.keystore</property>
      <property name="keyStorePass">KEYSTORE_PASSWORD</property>
      <property name="keyAlias">ejb-ssl</property>
      <property name="keyPassword">EJB-SSL_KEYPAIR_PASSWORD</property>
   </bean>
```

The sample configuration will create a connector that listens for SSL connections on port 3843. This port will need to be opened on the server firewall for access by clients.

12.3.2. Configure EJB3 Beans for SSL Transport

All EJB3 beans use the unsecured RMI connector by default. In order to enabled a bean to be invoked via SSL, the bean needs to be annotated with `@org.jboss.annotation.ejb.RemoteBinding`.

The annotation in *Example 12.3, “EJB3 bean annotation to enable secure remote invocation”* will bind an EJB3 bean to the JNDI name `StatefulSSL`. The proxy implementing the remote interface, returned to a client when the bean is requested from JNDI, will communicate with the server via SSL.

Example 12.3. EJB3 bean annotation to enable secure remote invocation

```
@RemoteBinding(clientBindUrl="sslsocket://0.0.0.0:3843",
 jndiBinding="StatefulSSL"),
     @Remote(BusinessInterface.class)
     public class StatefulBean implements BusinessInterface
     {
        ...
     }
```

Note

In *Example 12.3, “EJB3 bean annotation to enable secure remote invocation”* the IP address is specified as 0.0.0.0, meaning "all interfaces". This should be changed in practice to the value of the ${jboss.bind.address} system property.

Enabling both secure and insecure invocation of an EJB3 bean. You can enable both secure and insecure remote method invocation of the same EJB3 bean. *Example 12.4, “EJB3 Bean annotation for both secure and insecure remote invocation”* demonstrates the annotations to do this.

Example 12.4. EJB3 Bean annotation for both secure and insecure remote invocation

```
@RemoteBindings({
       @RemoteBinding(clientBindUrl="sslsocket://0.0.0.0:3843",
jndiBinding="StatefulSSL"),
       @RemoteBinding(jndiBinding="StatefulNormal")
    })
    @Remote(BusinessInterface.class)
    public class StatefulBean implements BusinessInterface
    {
       ...
    }
```

If a client requests `StatefulNormal` from JNDI, the returned proxy implementing the remote interface will communicate with the server via the unencrypted socket protocol; and if `StatefulSSL` is requested, the returned proxy implementing the remote interface will communicate with the server via SSL.

12.4. EJB2 Configuration

EJB2 remote invocation uses a single unified invoker, which runs by default on port 4446. The configuration of the unified invoker used for EJB2 remote method invocation is defined in the `deploy/remoting-jboss-beans.xml` file of a JBoss Application Server profile. Add the following SSL Socket Factory bean and an SSL Domain bean in this file.

Example 12.5. SSL Server Factory for EJB2

```
<bean name="sslServerSocketFactoryEJB2"
 class="org.jboss.security.ssl.DomainServerSocketFactory">
  <constructor>
    <parameter><inject bean="EJB2SSLDomain"/></parameter>
  </constructor>
</bean>

<bean name="EJB2SSLDomain"
 class="org.jboss.security.plugins.JaasSecurityDomain">
  <constructor>
    <parameter>EJB2SSLDomain</parameter>
  </constructor>
  <property name="keyStoreURL">resource:localhost.keystore</property>
  <property name="keyStorePass">changeit</property>
  <property name="keyAlias">ejb-ssl</property>
  <property name="keyPassword">EJB-SSL_KEYPAIR_PASSWORD</property>
</bean>
```

Configure SSL Transport for Beans. In the `deploy/remoting-jboss-beans.xml` file in the JBoss Application Server profile, update the code to reflect the information below:

Example 12.6. SSL Transport for Beans

```
...
<bean name="UnifiedInvokerConnector"
 class="org.jboss.remoting.transport.Connector">

 <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX(name="jboss.remoting:service=Conn
 exposedInterface=org.jboss.remoting.transport.ConnectorMBean.class,registerDirectly=true)
  </annotation>
  <property name="serverConfiguration"><inject
 bean="UnifiedInvokerConfiguration"/></property>
  <!-- add this to configure the SSL socket for the UnifiedInvoker -->
  <property name="serverSocketFactory"><inject
 bean="sslServerSocketFactoryEJB2"/></property>
   </bean>
   ...
<bean name="UnifiedInvokerConfiguration"
 class="org.jboss.remoting.ServerConfiguration">
  <constructor>
  <!-- transport: Others include sslsocket, bisocket, sslbisocket, http,
 https, rmi, sslrmi, servlet, sslservlet. -->
    <parameter>sslsocket</parameter><!-- changed from socket to sslsocket
 -->
  </constructor>
      ...
   </bean>
   ...
```

Masking Passwords in XML Configuration

Follow the instructions in this chapter to increase the security of your JBoss AS Installation by masking passwords that would otherwise be stored on the file system as clear text.

13.1. Password Masking Overview

Passwords are secret authentication tokens that are used to limit access to resources to authorised parties only. In order for JBoss services to access password protected resources, the password must be made available to the JBoss service. This can be done by means of command line arguments passed to the JBoss Application Server on start up, however this is not practical in a production environment. In production environments, typically, passwords are made available to JBoss services by their inclusion in configuration files.

All JBoss configuration files should be stored on secure file systems, and should be readable by the JBoss Application Server process owner only. Additionally, you can mask the password in the configuration file for an added level of security. Follow the instructions in this chapter to replace a clear text password in a Microcontainer bean configuration with a password mask. Refer to *Chapter 15, Encrypting Data Source Passwords* for instructions on encrypting Data Source passwords; to *Chapter 16, Encrypting the Keystore Password in a Tomcat Connector* for instructions on encrypting the key store password in Tomcat; and to *Chapter 17, Using LdapExtLoginModule with JaasSecurityDomain* for instructions on encrypting the password for LdapExtLoginModule.

Note

There is no such thing as impenetrable security. All good security measures merely increase the cost involved in unauthorised access of a system. Masking passwords is no exception - it is not impenetrable, but does defeat casual inspection of configuration files, and increases the amount of effort that will be required to extract the password in clear text.

Procedure 13.1. Masking a clear text password overview

1. Generate a key pair to use to encrypt passwords.
2. Encrypt the key store password.
3. Create password masks.
4. Replace clear text passwords with their password masks.

13.2. Generate a key store and a masked password

Generate a key store. Password masking uses a public/private key pair to encrypt passwords. You need to generate a key pair for use in password masking. By default JBoss Enterprise Application Platform 5 expects a key pair with the alias `jboss` in a key store at `jboss-as/bin/password/password.keystore`. The following procedures follow this default configuration. If you wish to change the key store location or key alias you will need to change the default configuration, and should refer to *Section 13.6, "Changing the password masking defaults"* for instructions.

Procedure 13.2. Generate a key pair and key store for password masking

1. At the command line, change directory to the `jboss-as/bin/password` directory.

2. Use `keytool` to generate the key pair with the following command:

   ```
   keytool -genkey -alias jboss -keyalg RSA -keysize 1024  -keystore
    password.keystore
   ```

   Important: You must specify the same password for the key store and key pair

3. **Optional:** Make the resulting password.keystore readable by the JBoss Application Server process owner only.

   On Unix-based systems this is accomplished by using the `chown` command to change ownership to the JBoss Application Server process owner, and `chmod  600 password.keystore` to make the file readable only by the owner.

   This step is recommended to increase the security of your server.

   Note: the JBoss Application Server process owner should not have interactive console login access. In that case you will be performing these operations as another user. Creating masked passwords requires read access to the key store, so you may wish to complete configuration of masked passwords before restricting the key store file permissions.

For more on key stores and the `keytool` command, refer to *Section 12.1, "SSL Encryption overview"*.

13.3. Encrypt the key store password

With password masking, passwords needed by Jboss services are not stored in clear text in xml configuration files. Instead they are stored in a file that is encrypted using a key pair that you provide.

In order to decrypt this file and access the masked passwords at run time, JBoss Application Server needs to be able to use the key pair you created. You provide the key store password to JBoss Application Server by means of the JBoss Password Tool, `password_tool`. This tool will encrypt and store your key store password. Your key store password will then be available to the

JBoss Password Tool for masking passwords, and to the JBoss Application Server for decrypting them at run time.

Procedure 13.3. Encrypt the key store password

1. At the command line, change to the `jboss-as/bin` directory.

2. Run the password tool, using the command `./password_tool.sh` for Unix-based systems, or `password_tool.bat` for Windows-based systems.

   Result: The JBoss Password Tool will start, and will report '`Keystore is null. Please specify keystore below:`'.

3. Select '`0: Encrypt Keystore Password`' by pressing 0, then Enter.

   Result: The password tool responds with '`Enter keystore password`'.

4. Enter the key store password you specified in *Generate a key pair and key store for password masking*.

   Result: The password tool responds with '`Enter Salt (String should be at least 8 characters)`'.

5. Enter a random string of characters to aid with encryption strength.

   Result: The password tool responds with '`Enter Iterator Count (integer value)`'.

6. Enter a whole number to aid with encryption strength.

   Result: The password tool responds with: '`Keystore Password encrypted into password/jboss_keystore_pass.dat`'.

7. Select '`5:Exit`' to exit.

   Result: The password tool will exit with the message: '`Keystore is null. Cannot store.`'. This is normal.

8. **Optional:** Make the resulting file `password/jboss_keystore_pass.dat` readable by the JBoss Application Server process owner only.

   On Unix-based systems this is accomplished by using the `chown` command to change ownership to the JBoss Application Server process owner, and `chmod 600 jboss-keystore_pass.dat` to make the file readable only by the owner.

   This step is recommended to increase the security of your server. Be aware that if this encrypted key is compromised, the security offered by password masking is significantly reduced. This file should be stored on a secure file system.

   Note: the JBoss Application Server process owner should not have interactive console login access. In this case you will be performing these operations as another user. Creating masked passwords requires read access to the key store, so you may wish to complete configuration of masked passwords before restricting the key store file permissions.

Note: You should only perform this key store password encryption procedure once. If you make a mistake entering the keystore password, or you change the key store at a later date, you should delete the `jboss-keystore_pass.dat` file and repeat the procedure. Be aware that if you change the key store any masked passwords that were previously generated will no longer function.

13.4. Create password masks

The JBoss Password Tool maintains an encrypted password file `jboss-as/bin/password/jboss_password_enc.dat`. This file is encrypted using a key pair you provide to the password tool, and it contains the passwords that will be masked in configuration files. Passwords are stored and retrieved from this file by 'domain', an arbitrary unique identifier that you specify to the Password Tool when storing the password, and that you specify as part of the annotation that replaces that clear text password in configuration files. This allows the JBoss Application Server to retrieve the correct password from the file at run time.

Note: If you previously made the key store and encrypted key store password file readable only by the JBoss Application Server process owner, then you need to perform the following procedure as the JBoss Application Server process owner, or else make the keystore (`jboss-as/bin/password/password.keystore`) and encrypted key store password file (`jboss-as/bin/password/jboss_keystore_pass.dat`) readable by your user, and the encrypted passwords file `jboss-as/bin/password/jboss_password_enc.dat` (if it already exists) read and writeable, while you perform this operation.

Procedure 13.4. Create password masks

Prerequisites:

- *Generate a key pair and key store for password masking*.
- *Encrypt the key store password*.

1. At the command line, change to the `jboss-as/bin` directory.
2. Run the password tool, using the command `./password_tool.sh` for Unix-based systems, or `password_tool.bat` for Windows-based systems.

   Result: The JBoss Password Tool will start, and will report '`Keystore is null. Please specify keystore below:`'.
3. Select '`1:Specify KeyStore`' by pressing 1 then Enter.

   Result: The password tool responds with '`Enter Keystore location including the file name`'.

4. Enter the path to the key store you created in *Generate a key pair and key store for password masking*. You can specify an absolute path, or the path relative to `jboss-as/bin`. This should be `password/password.keystore`, unless you have performed an advanced installation and changed the defaults as per *Section 13.6, “Changing the password masking defaults”*.

   Result: The password tool responds with '`Enter Keystore alias`'.

5. Enter the key alias. This should be `jboss`, unless you have performed an advanced installation and changed the defaults as per *Section 13.6, “Changing the password masking defaults”*.

   Result: If the key store and key alias are accessible, the password tool will respond with some log4j WARNING messages, then the line '`Loading domains [`', followed by any existing password masks, and the main menu.

6. Select '`2:Create Password`' by pressing 2, then Enter

   Result: The password tool responds with: '`Enter security domain:`'.

7. Enter a name for the password mask. This is an arbitrary unique name that you will use to identify the password mask in configuration files.

   Result: The password tool responds with: '`Enter passwd:`'.

8. Enter the password that you wish to mask.

   Result: The password tool responds with: '`Password created for domain:`*`mask name`*'

9. Repeat the password mask creation process to create masks for all passwords you wish to mask.

10. Exit the program by choosing '`5:Exit`'

13.5. Replace clear text passwords with their password masks

Clear text passwords in xml configuration files can be replaced with password masks by changing the property assignment for an annotation. Generate password masks for any clear text password that you wish to mask in Microcontainer bean configuration files by following *Create password masks*. Then replace the configuration occurrence of each clear text password with an annotation referencing its mask.

The general form of the annotation is:

Example 13.1. General form of password mask annotation

```
<annotation>@org.jboss.security.integration.password.Password(securityDomain=MASK_NAME,metho
annotation>
```

13.6. Changing the password masking defaults

JBoss Enterprise Application Platform 5 ships with server profiles preconfigured for password masking. By default the server profiles are configured to use the keystore `jboss-as/bin/password/password.keystore`, and the key alias `jboss`. If you store the key pair used for password masking elsewhere, or under a different alias, you will need to update the server profiles with the new location or key alias.

The password masking key store location and key alias is specified in the file `deploy/security/security-jboss-beans.xml` under each of the included JBoss Application Server server profiles.

Example 13.2. Preconfigured Password Masking defaults in security-jboss-beans.xml

```
<!-- Password Mask Management Bean-->
   <bean name="JBossSecurityPasswordMaskManagement"

 class="org.jboss.security.integration.password.PasswordMaskManagement" >
         <property
 name="keyStoreLocation">password/password.keystore</property>
         <property name="keyStoreAlias">jboss</property>
         <property
 name="passwordEncryptedFileName">password/jboss_password_enc.dat</property>
         <property
 name="keyStorePasswordEncryptedFileName">password/jboss_keystore_pass.dat</
property>
   </bean>
```

Overriding SSL Configuration

Many services in JBoss allow usage of SSL for secure communication. To configure SSL, these services require a *KeyStore* for the certificate and private key and possibly a *TrustStore* with the trusted client certificates. Those attributes can be configured using the JDK system properties (`javax.net.ssl.keyStore`, `javax.net.ssl.keyStorePassword`, `javax.net.ssl.trustStore`, `javax.net.ssl.trustStorePassword`) or by a service specific set of attributes.

There can be situations when the AS as a whole should be using just one keystore and truststore for all the services, essentially ignoring all the system properties and service's specific configurations.

Starting in JBoss AS 6 there is a new service that can be installed at bootstrap that can override all the configuration for the *KeyStore* and *TrustStore*, provided that the service uses the default algorithm for the `KeyManagerFactory` (`SunX509` for Sun, JRockit and OpenJDK and `IbmX509` for IBM) and `TrustManagerFactory` (`PKIX` for Sun, JRockit, OpenJDK and IBM).

Here is an example configuration for the service in `conf/bootstrap/security.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
    Security bootstrap configuration
-->
<deployment xmlns="urn:jboss:bean-deployer:2.0">

  ...

  <bean name="JBossSSLConfiguration" class="org.jboss.security.ssl.JBossSSLConfiguration">
      <property name="keyStoreURL">my.keystore</property>
      <property name="keyStorePassword">changeit</property>
   </bean>
</deployment>
```

With this service in place, the `keystoreFile` and `keystorePass` attributes of a HTTPS connector in `deploy/jbossweb.sar/server.xml` would be overridden for example.

These are the properties the JBossSSLConfiguration bean accepts:

- `keyStoreURL`
- `keyStorePassword`

- `keyStoreAlias`
- `keyStoreProvider`
- `keyStoreProviderArgument`
- `trustStoreURL`
- `trustStorePassword`
- `trustStoreProvider`
- `trustStoreProviderArgument`

These properties are the same as the ones in the `JaasSecurityDomain` bean. See *Section 4.3, “The JaasSecurityDomain Bean”* for a detailed description.

The `keyStorePassword` can be masked using the same methods described for the `keyStorePass`.

Note

There is still no support for using the Password annotation (shown in *Chapter 13, Masking Passwords in XML Configuration*) to mask those passwords as the `PasswordMaskManagement` bean is started much later in the boot process.

Encrypting Data Source Passwords

Database connections for the JBoss AS are defined in `*-ds.xml` data source files. These database connection details include clear text passwords. You can increase the security of your server by replacing clear text passwords in datasource files with encrypted passwords.

This chapter presents two different methods for encrypting data source passwords. The first is *Secured Identity*. The second is *Configured Identity with Password Based Encryption (PBE)*.

15.1. Secured Identity

The class `org.jboss.resource.security.SecureIdentityLoginModule` can be used to both encrypt database passwords and to provide a decrypted version of the password when the data source configuration is required by the server. The `SecureIdentityLoginModule` uses a hard-coded password to encrypt/decrypt the data source password.

Procedure 15.1. Overview: Using SecureIdentityLoginModule to encrypt a datasource password

1. Encrypt the data source password.
2. Create an application authentication policy with the encrypted password.
3. Configure the data source to use the application authentication policy.

15.1.1. Encrypt the data source password

The data source password is encrypted using the `SecureIdentityLoginModule` main method by passing in the clear text password. The SecureIdentityLoginModule is provided by `jbosssx.jar`.

Procedure 15.2. Encrypt a datasource password

This procedure is for JBoss Enterprise Application Platform versions 5.1 and later

1. Change directory to the `jboss-as` directory
2. **Linux command.**

   ```
   java -cp client/jboss-logging-spi.jar:lib/jbosssx.jar
    org.jboss.resource.security.SecureIdentityLoginModule PASSWORD
   ```

   Windows command:

   ```
   java -cp client\jboss-logging-spi.jar;lib\jbosssx.jar
    org.jboss.resource.security.SecureIdentityLoginModule PASSWORD
   ```

Result: The command will return an encrypted password.

15.1.2. Create an application authentication policy with the encrypted password

Each JBoss Application Server server profile has a `conf/login-config.xml` file, where application authentication policies are defined for that profile. To create a an application authentication policy for your encrypted password, add a new <application-policy> element to the <policy> element.

Example 15.1, “Example application authentication policy with encrypted data source password” is a fragment of a `login-config.xml` file showing an application authentication policy of name "EncryptDBPassword".

Example 15.1. Example application authentication policy with encrypted data source password

```
 <policy>
 ...
     <!-- Example usage of the SecureIdentityLoginModule -->
     <application-policy name="EncryptDBPassword">
         <authentication>
                       <login-module  code="org.jboss.resource.security.SecureIdentityLoginModule"
 flag="required">
                  <module-option name="username">admin</module-option>
                  <module-option name="password">5dfc52b51bd35553df8592078de921bc</module-
option>

                                                                                  <module

module-option>
              </login-module>
         </authentication>
     </application-policy>
 </policy>
```

SecureIdentityLoginModule module options

username
    Specify the user name to use when establishing a connection to the database.

password
: Provide the encrypted password generated in *Section 15.1.1, “Encrypt the data source password”*.

managedConnectionFactoryName

jboss.jca:name
: Nominate a Java Naming and Directory Interface (JNDI) name for this datasource.

jboss.jca:service
: Specify the transaction type

Transaction types

NoTxCM
: No transaction support

LocalTxCM
: Single resource transaction support

TxCM
: Single resource or distributed transaction support

XATxCM
: Distributed transaction support

15.1.3. Configure the data source to use the application authentication policy

The data source is configured in a `*-ds.xml` file. Remove the <user-name> and <password> elements from this file, and replace them with a <security-domain> element. This element will contain the application authentication policy name specified following *Section 15.1.2, “Create an application authentication policy with the encrypted password”*.

Using the example name from *Section 15.1.2, “Create an application authentication policy with the encrypted password”*, "EncryptDBPassword", will result in a data source file that looks something like *Example 15.2, “Example data source file using secured identity”*.

Example 15.2. Example data source file using secured identity

```
<?xml version="1.0" encoding="UTF-8"?>
<datasources>
   <local-tx-datasource>
      <jndi-name>PostgresDS</jndi-name>
      <connection-url>jdbc:postgresql://127.0.0.1:5432/test?protocolVersion=2</connection-url>
      <driver-class>org.postgresql.Driver</driver-class>
```

```
        <min-pool-size>1</min-pool-size>
        <max-pool-size>20</max-pool-size>

        <!-- REPLACED WITH security-domain BELOW
        <user-name>admin</user-name>
        <password>password</password>
        -->

        <security-domain>EncryptDBPassword</security-domain>

        <metadata>
            <type-mapping>PostgreSQL 8.0</type-mapping>
        </metadata>
    </local-tx-datasource>
</datasources>
```

15.2. Configured Identity with Password Based Encryption

The `org.jboss.resource.security.JaasSecurityDomainIdentityLoginModule` is a login module for statically defining a data source using an encrypted password. that has been encrypted by a JaasSecurityDomain. The base64 format of the data source password may be generated using the PBEUtils command:

```
java -cp jbosssx.jar org.jboss.security.plugins.PBEUtils SALT
 ITERATION-COUNT DOMAIN-PASSWORD DATASOURCE-PASSWORD
```

The commands for PBEUtils arguments are:

SALT
: The Salt attribute from the JaasSecurityDomain (Must only be eight characters long).

ITERATION COUNT
: The IterationCount attribute from the JaasSecurity domain.

DOMAIN-PASSWORD
: The plaintext password that maps to the KeyStorePass attribute from the JaasSecurityDomain.

DATASOURCE-PASSWORD
: The plaintext password for the data source that should be encrypted with the JaasSecurityDomain password.

Example 15.3, “PBEUtils command example” provides an example of the command.

Example 15.3. PBEUtils command example

```
java -cp jbosssx.jar org.jboss.security.plugins.PBEUtils abcdefgh 13 master
''
Encoded password: E5gtGMKcXPP
```

Add the following application policy to the `$JBOSS_HOME/server/$PROFILE/conf/login-config.xml` file.

```
<application-policy name = "EncryptedHsqlDbRealm">
   <authentication>
      <login-module code = "org.jboss.resource.security.JaasSecurityDomainIdentityLoginModule"
       flag = "required">
          <module-option name = "username">sa</module-option>
          <module-option name = "password">E5gtGMKcXPP</module-option>
                                                                                              <module-option          name
           =           "managedConnectionFactoryName">jboss.jca:service=LocalTxCM,name=DefaultDS</
module-option>
                                                                                     <module-option        name        =

module-option>
       </login-module>
    </authentication>
 </application-policy>
```

The `$JBOSS_HOME/server/$PROFILE/docs/examples/jca/hsqldb-encrypted-ds.xml` illustrates that data source configuration along with the JaasSecurityDomain configuration for the keystore:

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- The Hypersonic embedded database JCA connection factory config
that illustrates the use of the JaasSecurityDomainIdentityLoginModule
to use encrypted password in the data source configuration.

$Id: hsqldb-encrypted-ds.xml,v 1.1.2.1 2004/06/04 02:20:52 starksm Exp $ -->
```

```
<datasources>
  <local-tx-datasource>

    <!-- The jndi name of the DataSource, it is prefixed with java:/ -->
    <!-- Datasources are not available outside the virtual machine -->
    <jndi-name>DefaultDS</jndi-name>

    <!-- for tcp connection, allowing other processes to use the hsqldb
    database. This requires the org.jboss.jdbc.HypersonicDatabase mbean.
    <connection-url>jdbc:hsqldb:hsql://localhost:1701</connection-url>
    -->
    <!-- for totally in-memory db, not saved when jboss stops.
    The org.jboss.jdbc.HypersonicDatabase mbean necessary
    <connection-url>jdbc:hsqldb:.</connection-url>
    -->
    <!-- for in-process persistent db, saved when jboss stops. The
    org.jboss.jdbc.HypersonicDatabase mbean is necessary for properly db shutdown
    -->
     <connection-url>jdbc:hsqldb:${jboss.server.data.dir}${/}hypersonic${/}localDB</connection-
url>

    <!-- The driver class -->
    <driver-class>org.hsqldb.jdbcDriver</driver-class>

     <!--example of how to specify class that determines if exception means connection should
 be destroyed-->
                                                                    <!--exception-sorter-class-
name>org.jboss.resource.adapter.jdbc.vendor.DummyExceptionSorter</exception-sorter-class-
name-->

   <!-- this will be run before a managed connection is removed from the pool for use by a client-->
    <!--<check-valid-connection-sql>select * from something</check-valid-connection-sql> -->

    <!-- The minimum connections in a pool/sub-pool. Pools are lazily constructed on first use -->
    <min-pool-size>5</min-pool-size>

    <!-- The maximum connections in a pool/sub-pool -->
    <max-pool-size>20</max-pool-size>

    <!-- The time before an unused connection is destroyed -->
     <!-- NOTE: This is the check period. It will be destroyed somewhere between 1x and 2x this
 timeout after last use -->
     <!-- TEMPORARY FIX! - Disable idle connection removal, HSQLDB has a problem with not
 reaping threads on closed connections -->
```

```
      <idle-timeout-minutes>0</idle-timeout-minutes>

      <!-- sql to call when connection is created
        <new-connection-sql>some arbitrary sql</new-connection-sql>
      -->

      <!-- sql to call on an existing pooled connection when it is obtained from pool
         <check-valid-connection-sql>some arbitrary sql</check-valid-connection-sql>
      -->

     <!-- example of how to specify a class that determines a connection is valid before it is handed
 out from the pool
                                                                      <valid-connection-checker-class-
name>org.jboss.resource.adapter.jdbc.vendor.DummyValidConnectionChecker</valid-
connection-checker-class-name>
      -->

      <!-- Whether to check all statements are closed when the connection is returned to the pool,
           this is a debugging feature that should be turned off in production -->
      <track-statements></track-statements>

      <!-- Use the getConnection(user, pw) for logins
        <application-managed-security></application-managed-security>
      -->

      <!-- Use the security domain defined in conf/login-config.xml -->
      <security-domain>EncryptedHsqlDbRealm</security-domain>

      <!-- This mbean can be used when using in process persistent hypersonic -->
      <depends>jboss:service=Hypersonic,database=localDB</depends>

      <!-- The datasource must depend on the mbean -->
          <depends>jboss.security:service=JaasSecurityDomain,domain=ServerMasterPassword</
depends>
   </local-tx-datasource>

   <!-- The JaasSecurityDomain used for encryption. Use the name
   "jboss.security:service=JaasSecurityDomain,domain=ServerMasterPassword"
   as the value of the JaasSecurityDomainIdentityLoginModule
   jaasSecurityDomain login module option in the EncryptedHsqlDbRealm
   login-config.xml section. Typically this service config should be in
   the conf/jboss-service.xml descriptor.
   The opaque master.password file could be created using:
      java  -cp  jbosssx.jar  org.jboss.security.plugins.FilePassword  12345678  17  master
 server.password

   The corresponding login-config.xml would look like:
    <application-policy name = "EncryptedHsqlDbRealm">
       <authentication>
                                                                  <login-module       code       =
 "org.jboss.resource.security.JaasSecurityDomainIdentityLoginModule"
          flag = "required">
             <module-option name = "username">sa</module-option>
             <module-option name = "password">E5gtGMKcXPP</module-option>
                                                                                   <module-option
  name  =  "managedConnectionFactoryName">jboss.jca:service=LocalTxCM,name=DefaultDS</
module-option>
                                                                                        <module-
option                                   name                                              =

module-option>
          </login-module>
       </authentication>
    </application-policy>
    where the encrypted password was generated using:
     java -cp jbosssx.jar org.jboss.security.plugins.PBEUtils abcdefgh 13 master ''
     Encoded password: E5gtGMKcXPP
   -->
   <mbean code="org.jboss.security.plugins.JaasSecurityDomain"
      name="jboss.security:service=JaasSecurityDomain,domain=ServerMasterPassword">
      <constructor>
         <arg type="java.lang.String" value="ServerMasterPassword"></arg>
      </constructor>
      <!-- The opaque master password file used to decrypt the encrypted
      database password key -->
                                                                                       <attribute

conf/server.password</attribute>
      <attribute name="Salt">abcdefgh</attribute>
      <attribute name="IterationCount">13</attribute>
   </mbean>

   <!-- This mbean can be used when using in process persistent db -->
   <mbean code="org.jboss.jdbc.HypersonicDatabase"
      name="jboss:service=Hypersonic,database=localDB">
      <attribute name="Database">localDB</attribute>
      <attribute name="InProcessMode">true</attribute>
  </mbean>
</datasources>
```

Warning

Remember to use the same Salt and IterationCount in the MBean that was used during the password generation step.

Note

You may see the following error while starting a service that depends on the encrypted data source:

```
Caused by: java.security.InvalidAlgorithmParameterException: Parameters missing
        at com.sun.crypto.provider.SunJCE_af.a(DashoA12275)
        at com.sun.crypto.provider.PBEWithMD5AndDESCipher.engineInit(DashoA12275)
        at javax.crypto.Cipher.a(DashoA12275)
        at javax.crypto.Cipher.a(DashoA12275)
        at javax.crypto.Cipher.init(DashoA12275)
        at javax.crypto.Cipher.init(DashoA12275)
        at org.jboss.security.plugins.JaasSecurityDomain.decode(JaasSecurityDomain.java:325)
        at org.jboss.security.plugins.JaasSecurityDomain.decode64(JaasSecurityDomain.java:351)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:585)
        at org.jboss.mx.interceptor.ReflectedDispatcher.invoke(ReflectedDispatcher.java:155)
        ... 139 more
```

The error most likely means that the following MBean is not yet started as a service:

```
(jboss.security:service=JaasSecurityDomain,domain=ServerMasterPassword)
```

The following element should be included so that the MBean starts before the data source, as per the example `hsqldb-encrypted-ds.xml` code shown previously.

```
<depends>jboss.security:service=JaasSecurityDomain,domain=ServerMasterPassword</
depends>
```

Encrypting the Keystore Password in a Tomcat Connector

SSL with Tomcat requires a secure connector. This means that the keystore/truststore password cannot be passed as an attribute in the connector element of Tomcat's `server.xml`.

A working understanding of the JaasSecurityDomain that supports keystores, truststores, and password based encryption is advised. Please see *Chapter 19, Secure Remote Password Protocol* for more information.

The first step is to add a connector element in `server.xml` in `$JBOSS_HOME/server/$PROFILE/deploy/jbossweb.sar`.

```
<!-- SSL/TLS Connector with encrypted keystore password configuration  -->
    <Connector protocol="HTTP/1.1" SSLEnabled="true"
        port="8443" address="${jboss.bind.address}"
        scheme="https" secure="true" clientAuth="false"
        sslProtocol="TLS"
        securityDomain="encrypt-keystore-password"
        SSLImplementation="org.jboss.net.ssl.JBossImplementation"/>
```

You now need to provide the definition for the JaasSecurityDomain in a `*-service.xml` or in `*-jboss-beans.xml` in the deploy directory. Here is a MBean example:

```
<mbean code="org.jboss.security.plugins.JaasSecurityDomain"
    name="jboss.security:service=PBESecurityDomain">
    <constructor>
      <arg type="java.lang.String" value="encrypt-keystore-password"></arg>
    </constructor>
    <attribute name="KeyStoreURL">resource:localhost.keystore</attribute>
                                                                  <attribute

conf/keystore.password</attribute>
    <attribute name="Salt">abcdefgh</attribute>
    <attribute name="IterationCount">13</attribute>
  </mbean>
```

The Salt and IterationCount are the variables that define the strength of your encrypted password, so you can vary it from what is shown. Just remember to use the changed value when generating the encrypted password.

> **Note**
>
> The Salt must be eight characters long.

Your keystore is the localhost.keystore which will be in your conf directory. The keystore.password is your encrypted password that will reside in the conf directory and will be generated in the next step.

You now need to go to the conf directory of your JBoss AS instance (`default/conf`, for example).

```
java -cp ../lib/jbosssx.jar org.jboss.security.plugins.FilePassword abcdefgh 13 unit-tests-server
 keystore.password
```

Run this on a single line. In the above example, "abcdefgh" is the Salt and 13 is the iteration count; 'unit-tests-server' is the password of the keystore that you are protecting; and keystore.password is the file in which the encrypted password will be stored.

You can then update the Tomcat service MBean to depend on your JaasSecurityDomain MBean because Tomcat has to start after `jboss.security:service=PBESecurityDomain`.

Navigate to `$JBOSS_HOME/server/$PROFILE/deploy/jbossweb.sar/META-INF`. Open `jboss-service.xml` and add the following <depends> tag towards the end.

```
      <depends>jboss.security:service=PBESecurityDomain</depends>
   </mbean>
</server>
```

Note

In case of a native connector the `SSLPassword` attribute can also be encrypted using a JaasSecurityDomain bean. One additional step required is to create the masked password with:

```
java -cp jbosssx.jar org.jboss.security.plugins.PBEUtils SALT
 ITERATION-COUNT DOMAIN-PASSWORD KEYSTORE-PASSWORD
```

Using the encrypted password output given by the above command the native connector can now be set up. Here is an example:

```
<!-- SSL/TLS Connector with encrypted keystore password configuration  -->
   <Connector protocol="HTTP/1.1" SSLEnabled="true"
      port="8443" address="${jboss.bind.address}"
      scheme="https" secure="true" clientAuth="false"
      SSLPassword="KAaxoMQCJH30GZWb96Mov"
      securityDomain="encrypt-keystore-password"
      SSLCertificateFile="server.crt"
      SSLCertificateKeyFile="server.pem" SSLProtocol="TLSv1" />
```

Please see *Chapter 15, Encrypting Data Source Passwords* for related information.

16.1. Medium Security Usecase

A user does not want to encrypt the keystore password but wants to externalize it (outside of `server.xml`) or wants to make use of a predefined JaasSecurityDomain.

Procedure 16.1. Predefined JaasSecurityDomain

1. **Update `jboss-service.xml` to add a connector**

```
<mbean code="org.jboss.security.plugins.JaasSecurityDomain"
   name="jboss.security:service=SecurityDomain">
   <constructor>
     <arg type="java.lang.String" value="jbosstest-ssl"></arg>
   </constructor>
   <attribute name="KeyStoreURL">resource:localhost.keystore</attribute>
```

```
    <attribute name="KeyStorePass">unit-tests-server</attribute>
  </mbean>
```

2. **Add a <depends> tag to the Tomcat service**

   Navigate to `$JBOSS_HOME/server/$PROFILE/deploy/jbossweb.sar`. Open `server.xml` and add the following <depends> element towards the end:

```
<depends>jboss.security:service=SecurityDomain</depends>
  </mbean>
</server>
```

3. **Define the JaasSecurityDomain MBean in a `-service.xml` file**

   `security-service.xml` in the deploy directory, for example.

```
<mbean code="org.jboss.security.plugins.JaasSecurityDomain"
    name="jboss.security:service=SecurityDomain">
    <constructor>
       <arg type="java.lang.String" value="jbosstest-ssl"></arg>
    </constructor>
    <attribute name="KeyStoreURL">resource:localhost.keystore</attribute>
    <attribute name="KeyStorePass">unit-tests-server</attribute>
 </mbean>
```

File Permission issue reading the keystore (or FileNotFoundException)

If you see this error, remember the keystore file should be writable by the user id that is running JBoss Enterprise Application Platform.

Using LdapExtLoginModule with JaasSecurityDomain

This chapter provides guidance on how the LdapExtLoginModule can be used with an encrypted password to be decrypted by a JaasSecurityDomain. This chapter assumes that the LdapExtLoginModule is already running correctly with a non-encrypted password.

The first step is to define the JaasSecurityDomain MBean that is going to be used to decrypt the encrypted version of the password. This can then be added to the `$JBOSS_HOME/server/$PROFILE/conf/jboss-service.xml` or can be added to a `*-service.xml` descriptor in the deploy folder.

```
<mbean code="org.jboss.security.plugins.JaasSecurityDomain"
    name="jboss.security:service=JaasSecurityDomain,domain=jmx-console">
    <constructor>
       <arg type="java.lang.String" value="jmx-console"></arg>
    </constructor>
    <attribute name="KeyStorePass">some_password</attribute>
    <attribute name="Salt">abcdefgh</attribute>
    <attribute name="IterationCount">66</attribute>
 </mbean>
```

This is a simple configuration where the required password, Salt and Iteration Count used for the encryption or decryption are contained within the MBean definition.

It should be noted that the default cipher algorithm used by the JaasSecurityDomain implementation is "PBEwithMD5andDES". This can be modified using the "CipherAlgorithm" attribute.

Ensure that you change the KeyStorePass, Salt, and IterationCount values for your own deployment.

After this MBean has been defined, start the JBoss Enterprise Application Platform. Navigate to the JMX Console (*http://localhost:8080/jmx-console/* by default) and select the `org.jboss.security.plugins.JaasSecurityDomain` MBean.

On the `org.jboss.security.plugins.JaasSecurityDomain` page, look for the `encode64(String password)` method. Pass the plain text version of the `password` being used by the LdapExtLoginModule to this method, and invoke it. The return value should be the encrypted version of the password encoded as Base64.

Within the login module configuration, the following module-options should be set:

```
                                                         <module-option

console</module-option>
  <module-option name="bindCredential">2gx7gcAxcDuaHaJMgO5AVo</module-option>
```

The first option is a new option to specify that the JaasSecurityDomain used previously should be used to decrypt the password.

The bindCredential is then replaced with the encrypted form as Base64.

Firewalls

JBoss AS ships with many socket-based services that require open firewall ports. *Table 18.1, "The ports found in the default configuration"* lists services that listen on ports that must be activated when accessing JBoss behind a firewall. *Table 18.2, "Additional ports in the all configuration"* lists additional ports that exist in the all profile.

Table 18.1. The ports found in the default configuration

Port	Type	Service
1098	TCP	`org.jboss.naming.NamingService`
1099	TCP	`org.jboss.naming.NamingService`
4444	TCP	`org.jboss.invocation.jrmp.server.JRMPInvoker`
4445	TCP	`org.jboss.invocation.pooled.server.PooledInvoker`
8009	TCP	`org.jboss.web.tomcat.tc4.EmbeddedTomcatService`
8080	TCP	`org.jboss.web.tomcat.tc4.EmbeddedTomcatService`
8083	TCP	`org.jboss.web.WebService`
8093	TCP	`org.jboss.mq.il.uil2.UILServerILService`

Table 18.2. Additional ports in the all configuration

Port	Type	Service
1100	TCP	`org.jboss.ha.jndi.HANamingService`
1101	TCP	`org.jboss.ha.jndi.HANamingService`
1102	UDP	`org.jboss.ha.jndi.HANamingService`
1161	UDP	`org.jboss.jmx.adaptor.snmp.agent.SnmpAgentService`
1162	UDP	`org.jboss.jmx.adaptor.snmp.trapd.TrapdService`
1389	TCP	`ldaphost.jboss.org.LdapLoginModule`
3843[a]	TCP	`org.jboss.ejb3.SSLRemotingConnector`
3528	TCP	`org.jboss.invocation.iiop.IIOPInvoker`
3873	TCP	`org.jboss.ejb3.RemotingConnectors`
4447	TCP	`org.jboss.invocation.jrmp.server.JRMPInvokerHA`
10099	RMI	`org.jboss.security.srp.SRPRemoteServerInterface`
45566[b]	UDP	`org.jboss.ha.framework.server.ClusterPartition`

[a]Necessary only if SSL transport is configured for EJB3

[b] Plus two additional anonymous UDP ports, one can be set using the `rcv_port`, and the other cannot be set.

Secure Remote Password Protocol

The Secure Remote Password (SRP) protocol is an implementation of a public key exchange handshake described in the Internet standards working group request for comments 2945(RFC2945). The RFC2945 abstract states:

> This document describes a cryptographically strong network authentication mechanism known as the Secure Remote Password (SRP) protocol. This mechanism is suitable for negotiating secure connections using a user-supplied password, while eliminating the security problems traditionally associated with reusable passwords. This system also performs a secure key exchange in the process of authentication, allowing security layers (privacy and/or integrity protection) to be enabled during the session. Trusted key servers and certificate infrastructures are not required, and clients are not required to store or manage any long-term keys. SRP offers both security and deployment advantages over existing challenge-response techniques, making it an ideal drop-in replacement where secure password authentication is needed.

The complete RFC2945 specification can be obtained from *http://www.rfc-editor.org/rfc.html*. Additional information on the SRP algorithm and its history can be found at *http://www-cs-students.stanford.edu/~tjw/srp/*.

Algorithms like Diffie-Hellman and RSA are known as public key exchange algorithms. The concept of public key algorithms is that you have two keys, one public that is available to everyone, and one that is private and known only to you. When someone wants to send encrypted information to you, then encrypt the information using your public key. Only you are able to decrypt the information using your private key. Contrast this with the more traditional shared password based encryption schemes that require the sender and receiver to know the shared password. Public key algorithms eliminate the need to share passwords.

The JBossSX framework includes an implementation of SRP that consists of the following elements:

- An implementation of the SRP handshake protocol that is independent of any particular client/server protocol
- An RMI implementation of the handshake protocol as the default client/server SRP implementation
- A client side JAAS `LoginModule` implementation that uses the RMI implementation for use in authenticating clients in a secure fashion
- A JMX MBean for managing the RMI server implementation. The MBean allows the RMI server implementation to be plugged into a JMX framework and externalizes the configuration of the verification information store. It also establishes an authentication cache that is bound into the JBoss server JNDI namespace.

- A server side JAAS `LoginModule` implementation that uses the authentication cache managed by the SRP JMX MBean.

Figure 19.1, “The JBossSX components of the SRP client-server framework.” describes the key components involved in the JBossSX implementation of the SRP client/server framework.

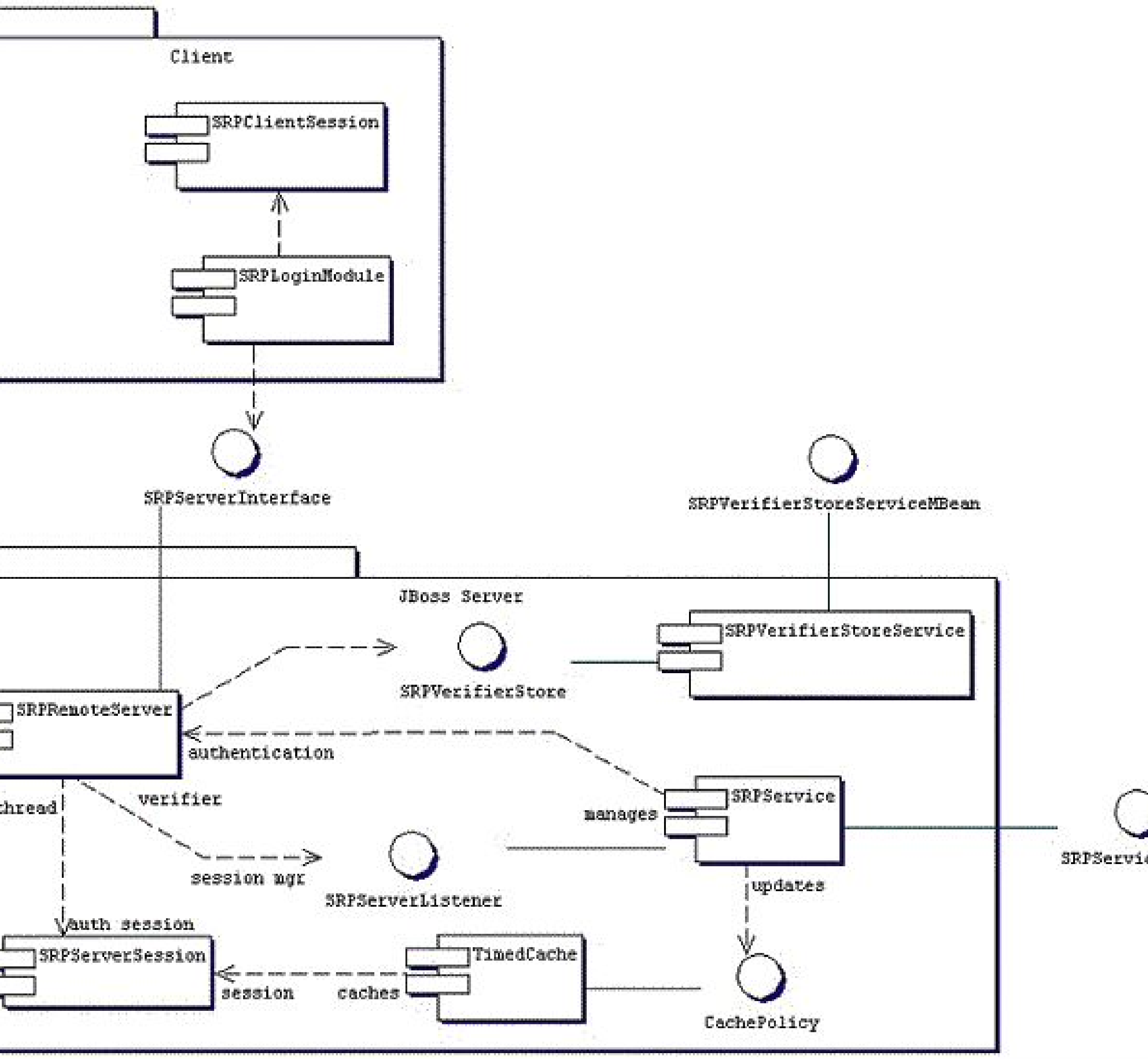

Figure 19.1. The JBossSX components of the SRP client-server framework.

On the client side, SRP shows up as a custom JAAS `LoginModule` implementation that communicates with the authentication server through an `org.jboss.security.srp.SRPServerInterface` proxy. A client enables authentication using SRP by creating a login configuration entry that includes the `org.jboss.security.srp.jaas.SRPLoginModule`. This module supports the following configuration options:

principalClassName
: Constant value, set to `org.jboss.security.srp.jaas.SRPPrincipal`.

srpServerJndiName
: JNDI name of the `SRPServerInterface` object used to communicate with the SRP authentication server. If both `srpServerJndiName` and `srpServerRmiUrl` options are specified, `srpServerJndiName` takes priority over `srpServerRmiUrl`.

srpServerRmiUrl
: RMI protocol URL string for the location of the `SRPServerInterface` proxy used to communicate with the SRP authentication server.

externalRandomA
: Flag that specifies whether the random component of the client public key "A" should come from the user callback. This can be used to input a strong cryptographic random number coming from a hardware token. If set to `true`, the feature is activated.

hasAuxChallenge
: Flag that specifies whether a string will be sent to the server as an additional challenge for the server to validate. If the client session supports an encryption cipher then a temporary cipher will be created using the session private key and the challenge object sent as a `javax.crypto.SealedObject`. If set to `true`, the feature is activated.

multipleSessions
: Flag that specifies whether a given client may have multiple SRP login sessions active. If set to `true`, the feature is activated.

Any other passed options that do not match one of the previously named options are treated as a JNDI property to use for the environment passed to the `InitialContext` constructor. This is useful if the SRP server interface is not available from the default `InitialContext`.

The `SRPLoginModule` and the standard `ClientLoginModule` must be configured to allow SRP authentication credentials to be used for access validation to security Java EE components. An example login configuration is described in *Example 19.1, “Login Configuration Entry”*.

Example 19.1. Login Configuration Entry

```
srp {
    org.jboss.security.srp.jaas.SRPLoginModule required
    srpServerJndiName="SRPServerInterface"
    ;

    org.jboss.security.ClientLoginModule required
    password-stacking="useFirstPass"
    ;
};
```

On the JBoss server side, there are two MBeans that manage the objects that collectively make up the SRP server. The primary service is the `org.jboss.security.srp.SRPService` MBean. The other MBean is`org.jboss.security.srp.SRPVerifierStoreService`.

`org.jboss.security.srp.SRPService` is responsible for exposing an RMI accessible version of the SRPServerInterface as well as updating the SRP authentication session cache.

The configurable SRPService MBean attributes include the following:

JndiName
: Specifies the name from which the SRPServerInterface proxy should be available. This is the location where the `SRPService` binds the serializable dynamic proxy to the `SRPServerInterface`. The default value is `srp/SRPServerInterface`.

VerifierSourceJndiName
: Specifies the name of the `SRPVerifierSource` implementation the `SRPService` must use. The source JNDI name defaults to `srp/DefaultVerifierSource`.

AuthenticationCacheJndiName
: Specifies the name under which the `org.jboss.util.CachePolicy` authentication implementation used for caching authentication information is bound. The SRP session cache is made available for use through this binding. The authentication JNDI cache defaults to `srp/AuthenticationCache`.

ServerPort
: RMI port for the `SRPRemoteServerInterface`. The default value is 10099.

ClientSocketFactory
: Optional custom `java.rmi.server.RMIClientSocketFactory` implementation class name used during the export of the `SRPServerInterface`. The default value is `RMIClientSocketFactory`.

ServerSocketFactory
: Optional custom `java.rmi.server.RMIServerSocketFactory` implementation class name used during the export of the `SRPServerInterface`. The default value is `RMIServerSocketFactory`.

AuthenticationCacheTimeout
: Cache policy timeout (in seconds). The default value is 1800 (30 minutes).

AuthenticationCacheResolution
: Specifies the timed cache policy resolution (in seconds). This controls the interval between checks for timeouts. The default value is 60 (1 minute).

RequireAuxChallenge
: Set if the client must supply an auxiliary challenge as part of the verify phase. This gives control over whether the `SRPLoginModule` configuration used by the client must have the `useAuxChallenge` option enabled.

OverwriteSessions
: Specifies whether a successful user authentication for an existing session should overwrite the current session. This controls the behavior of the server SRP session cache when clients have not enabled the multiple session per user mode. If set to `false`, the second user authentication attempt will succeed, however the resulting SRP session will not overwrite the previous SRP session state. The default value is `false`.

VerifierStoreJndiName
: Specifies the location of the SRP password information store implementation that must be provided and made available through JNDI.

`org.jboss.security.srp.SRPVerifierStoreService` is an example MBean service that binds an implementation of the `SRPVerifierStore` interface that uses a file of serialized objects as the persistent store. Although not realistic for a production environment, it does allow for testing of the SRP protocol and provides an example of the requirements for an `SRPVerifierStore` service.

The configurable `SRPVerifierStoreService` MBean attributes include the following:

JndiName
: JNDI name from which the `SRPVerifierStore` implementation should be available. If not specified it defaults to `srp/DefaultVerifierSource`.

StoreFile
: Location of the user password verifier serialized object store file. This can be either a URL or a resource name to be found in the classpath. If not specified it defaults to `SRPVerifierStore.ser`.

The `SRPVerifierStoreService` MBean also supports *`addUser`* and *`delUser`* operations for addition and deletion of users. The signatures are:

```java
public void addUser(String username, String password) throws IOException;
public void delUser(String username) throws IOException;
```

An example configuration of these services is presented in *Example 19.2, “The SRPVerifierStore interface”*.

19.1. Understanding the Algorithm

The appeal of the SRP algorithm is that is allows for mutual authentication of client and server using simple text passwords without a secure communication channel.

Note

Additional information on the SRP algorithm and its history can be found at *http://srp.stanford.edu/*.

There are six steps that are performed to complete authentication:

1. The client side `SRPLoginModule` retrieves from the naming service the SRPServerInterface instance for the remote authentication server.

2. The client side `SRPLoginModule` next requests the SRP parameters associated with the username attempting the login. There are a number of parameters involved in the SRP algorithm that must be chosen when the user password is first transformed into the verifier form used by the SRP algorithm. Rather than hard-coding the parameters (which could be done with minimal security risk), the JBossSX implementation allows a user to retrieve this information as part of the exchange protocol. The `getSRPParameters(username)` call retrieves the SRP parameters for the given username.

3. The client side `SRPLoginModule` begins an SRP session by creating an `SRPClientSession` object using the login username, clear-text password, and SRP parameters obtained from step 2. The client then creates a random number A that will be used to build the private SRP session key. The client then initializes the server side of the SRP session by invoking the `SRPServerInterface.init` method and passes in the username and client generated random number `A`. The server returns its own random number `B`. This step corresponds to the exchange of public keys.

4. The client side `SRPLoginModule` obtains the private SRP session key that has been generated as a result of the previous messages exchanges. This is saved as a private credential in the login `Subject`. The server challenge response `M2` from step 4 is verified by invoking the `SRPClientSession.verify` method. If this succeeds, mutual authentication of the client to server, and server to client have been completed. The client side `SRPLoginModule` next creates a challenge `M1` to the server by invoking `SRPClientSession.response` method passing the server random number `B` as an argument. This challenge is sent to the server via the `SRPServerInterface.verify` method and server's response is saved as `M2`. This step corresponds to an exchange of challenges. At this point the server has verified that the user is who they say they are.

5. The client side `SRPLoginModule` saves the login username and `M1` challenge into the `LoginModule` sharedState map. This is used as the Principal name and credentials by the

standard JBoss `ClientLoginModule`. The `M1` challenge is used in place of the password as proof of identity on any method invocations on Java EE components. The `M1` challenge is a cryptographically strong hash associated with the SRP session. Its interception via a third partly cannot be used to obtain the user's password.

6. At the end of this authentication protocol, the SRPServerSession has been placed into the SRPService authentication cache for subsequent use by the `SRPCacheLoginModule`.

Although SRP has many interesting properties, it is still an evolving component in the JBossSX framework and has some limitations of which you should be aware. Issues of note include the following:

- Where authentication is performed, the way in which JBoss detaches the method transport protocol from the component container could allow a user to snoop the SRP `M1` challenge and effectively use the challenge to make requests as the associated username. Custom interceptors can be used to prevent the issue, by encrypting the challenge using the SRP session key.

- The SRPService maintains a cache of SRP sessions that time out after a configurable period. Once they time out, any subsequent Java EE component access will fail because there is currently no mechanism for transparently renegotiating the SRP authentication credentials. You must either set the authentication cache timeout quite high, or handle re-authentication in your code on failure.

  Note

  The SRPService supports timeout durations up to 2,147,483,647 seconds, or approximately 68 years.

- There can only be one SRP session for a given username by default. The session is classed as stateful, because the negotiated SRP session produces a private session key that can be used for encryption and decryption between the client and server. JBoss supports multiple SRP sessions per user, however it is not possible to encrypt data with one session key, and decrypt it with another.

To use end-to-end SRP authentication for Java EE component calls, you must configure the security domain under which the components are secured to use the `org.jboss.security.srp.jaas.SRPCacheLoginModule`. The `SRPCacheLoginModule` has a single configuration option named `cacheJndiName` that sets the JNDI location of the SRP authentication `CachePolicy` instance. This must correspond to the `AuthenticationCacheJndiName` attribute value of the `SRPService` MBean.

The `SRPCacheLoginModule` authenticates user credentials by obtaining the client challenge from the `SRPServerSession` object in the authentication cache and comparing this to the challenge passed as the user credentials. *Figure 19.2, “SRPCacheLoginModule with SRP Session Cache”* illustrates the operation of the SRPCacheLoginModule.login method implementation.

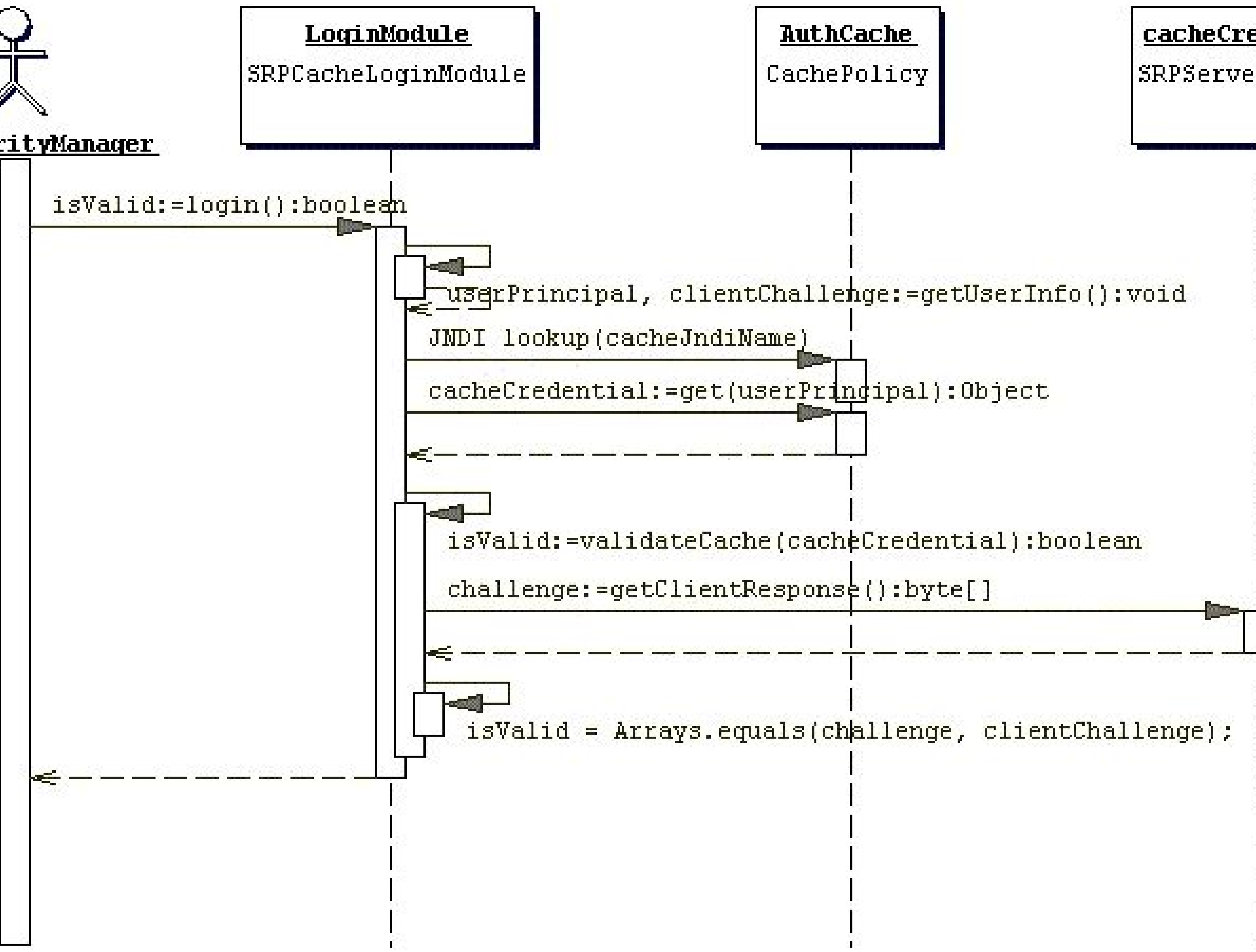
rityManager
LoginModule
SRPCacheLoginModule
AuthCache
CachePolicy
cacheCre
SRPServe
isValid:=login():boolean
userPrincipal, clientChallenge:=getUserInfo():void
JNDI lookup(cacheJndiName)
cacheCredential:=get(userPrincipal):Object
isValid:=validateCache(cacheCredential):boolean
challenge:=getClientResponse():byte[]
isValid = Arrays.equals(challenge, clientChallenge);

Figure 19.2. SRPCacheLoginModule with SRP Session Cache

19.2. Configure Secure Remote Password Information

You must create a MBean service that provides an implementation of the `SRPVerifierStore` interface that integrates with your existing security information stores. The `SRPVerifierStore` interface is shown in *Example 19.2, “The SRPVerifierStore interface”*.

Note

The default implementation of the `SRPVerifierStore` interface is not recommended for a production security environment because it requires all password hash information to be available as a file of serialized objects.

Example 19.2. The SRPVerifierStore interface

```
package org.jboss.security.srp;

import java.io.IOException;
import java.io.Serializable;
import java.security.KeyException;

public interface SRPVerifierStore
{
    public static class VerifierInfo implements Serializable
    {

        public String username;


        public byte[] salt;
        public byte[] g;
        public byte[] N;
    }


    public VerifierInfo getUserVerifier(String username)
        throws KeyException, IOException;

    public void setUserVerifier(String username, VerifierInfo info)
        throws IOException;


     public void verifyUserChallenge(String username, Object auxChallenge)
         throws SecurityException;
}
```

The primary function of a `SRPVerifierStore` implementation is to provide access to the `SRPVerifierStore.VerifierInfo` object for a given username. The `getUserVerifier(String)` method is called by the `SRPService` at that start of a user

SRP session to obtain the parameters needed by the SRP algorithm. The elements of the `VerifierInfo` objects are:

username
: The user's name or id used to login.

verifier
: One-way hash of the password or PIN the user enters as proof of identity. The `org.jboss.security.Util` class has a `calculateVerifier` method that performs that password hashing algorithm. The output password takes the form `H(salt | H(username | ':' | password))`, where `H` is the SHA secure hash function as defined by RFC2945. The username is converted from a string to a `byte[]` using UTF-8 encoding.

salt
: Random number used to increase the difficulty of a brute force dictionary attack on the verifier password database in the event that the database is compromised. The value should be generated from a cryptographically strong random number algorithm when the user's existing clear-text password is hashed.

g
: SRP algorithm primitive generator. This can be a well known fixed parameter rather than a per-user setting. The `org.jboss.security.srp.SRPConf` utility class provides several settings for `g`, including a suitable default obtained via `SRPConf.getDefaultParams().g()`.

N
: SRP algorithm safe-prime modulus. This can be a well known fixed parameter rather than a per-user setting. The `org.jboss.security.srp.SRPConf` utility class provides several settings for `N` including a good default which can obtained via `SRPConf.getDefaultParams().N()`.

Procedure 19.1. Integrate Existing Password Store

Read this procedure to understand the steps involved to integrate your existing password store.

1. **Create Hashed Password Information Store**

   If your passwords are already stored in an irreversible hashed form, then this can only be done on a per-user basis (for example, as part of an upgrade procedure).

   You can implement `setUserVerifier(String, VerifierInfo)` as a `noOp` method, or a method that throws an exception stating that the store is read-only.

2. **Create SRPVerifierStore Interface**

   You must create a custom `SRPVerifierStore` interface implementation that understands how to obtain the `VerifierInfo` from the store you created.

   The `verifyUserChallenge(String, Object)` can be used to integrate existing hardware token based schemes like SafeWord or Radius into the SRP algorithm. This interface method is called only when the client `SRPLoginModule` configuration specifies the `hasAuxChallenge` option.

3. **Create JNDI MBean**

   You must create a MBean that exposes the `SRPVerifierStore` interface available to JNDI, and exposes any configurable parameters required.

   The default `org.jboss.security.srp.SRPVerifierStoreService` will allow you to implement this, however you can also implement the MBean using a Java properties file implementation of `SRPVerifierStore` (refer to *Section 19.3, “Secure Remote Password Example”*).

19.3. Secure Remote Password Example

The example presented in this section demonstrates client side authentication of the user via SRP as well as subsequent secured access to a simple EJB using the SRP session challenge as the user credential. The test code deploys an EJB JAR that includes a SAR for the configuration of the server side login module configuration and SRP services.

The server side login module configuration is dynamically installed using the `SecurityConfig` MBean. A custom implementation of the `SRPVerifierStore` interface is also used in the example. The interface uses an in-memory store that is seeded from a Java properties file, rather than a serialized object store as used by the `SRPVerifierStoreService`.

This custom service is `org.jboss.book.security.ex3.service.PropertiesVerifierStore`. The following shows the contents of the JAR that contains the example EJB and SRP services.

```
[examples]$ jar tf output/security/security-ex3.jar
META-INF/MANIFEST.MF
META-INF/ejb-jar.xml
META-INF/jboss.xml
org/jboss/book/security/ex3/Echo.class
org/jboss/book/security/ex3/EchoBean.class
org/jboss/book/security/ex3/EchoHome.class
roles.properties
users.properties
security-ex3.sar
```

The key SRP related items in this example are the SRP MBean services configuration, and the SRP login module configurations. The `jboss-service.xml` descriptor of the `security-ex3.sar` is described in *Example 19.3, “The security-ex3.sar jboss-service.xml Descriptor”*.

The example client side and server side login module configurations are described in *Example 19.4, “The client side standard JAAS configuration”* and *Example 19.5, “The server side XMLLoginConfig configuration”* give .

Example 19.3. The security-ex3.sar jboss-service.xml Descriptor

```
<server>
    <!-- The custom JAAS login configuration that installs
         a Configuration capable of dynamically updating the
         config settings -->

    <mbean code="org.jboss.book.security.service.SecurityConfig"
           name="jboss.docs.security:service=LoginConfig-EX3">
        <attribute name="AuthConfig">META-INF/login-config.xml</attribute>
        <attribute name="SecurityConfigName">jboss.security:name=SecurityConfig</attribute>
    </mbean>

    <!-- The SRP service that provides the SRP RMI server and server side
         authentication cache -->
    <mbean code="org.jboss.security.srp.SRPService"
           name="jboss.docs.security:service=SRPService">
        <attribute name="VerifierSourceJndiName">srp-test/security-ex3</attribute>
        <attribute name="JndiName">srp-test/SRPServerInterface</attribute>
       <attribute name="AuthenticationCacheJndiName">srp-test/AuthenticationCache</attribute>
        <attribute name="ServerPort">0</attribute>
        <depends>jboss.docs.security:service=PropertiesVerifierStore</depends>
    </mbean>

    <!-- The SRP store handler service that provides the user password verifier
         information -->
    <mbean code="org.jboss.security.ex3.service.PropertiesVerifierStore"
           name="jboss.docs.security:service=PropertiesVerifierStore">
        <attribute name="JndiName">srp-test/security-ex3</attribute>
    </mbean>
</server>
```

The example services are the `ServiceConfig` and the `PropertiesVerifierStore` and `SRPService` MBeans. Note that the `JndiName` attribute of the `PropertiesVerifierStore` is equal to the `VerifierSourceJndiName` attribute of the `SRPService`, and that the `SRPService` depends on the `PropertiesVerifierStore`. This is required because the `SRPService` needs an implementation of the `SRPVerifierStore` interface for accessing user password verification information.

Example 19.4. The client side standard JAAS configuration

```
srp {
    org.jboss.security.srp.jaas.SRPLoginModule required
    srpServerJndiName="srp-test/SRPServerInterface"
    ;

    org.jboss.security.ClientLoginModule required
    password-stacking="useFirstPass"
    ;
};
```

The client side login module configuration makes use of the `SRPLoginModule` with a `srpServerJndiName` option value that corresponds to the JBoss server component `SRPService` JndiName attribute value(`srp-test/SRPServerInterface`). The `ClientLoginModule` must also be configured with the `password-stacking="useFirstPass"` value to propagate the user authentication credentials generated by the `SRPLoginModule` to the EJB invocation layer.

Example 19.5. The server side XMLLoginConfig configuration

```
<application-policy name="security-ex3">
    <authentication>
        <login-module code="org.jboss.security.srp.jaas.SRPCacheLoginModule"
                      flag = "required">
          <module-option name="cacheJndiName">srp-test/AuthenticationCache</module-option>
        </login-module>
        <login-module code="org.jboss.security.auth.spi.UsersRolesLoginModule"
                      flag = "required">
           <module-option name="password-stacking">useFirstPass</module-option>
        </login-module>
    </authentication>
</application-policy>
```

There are two issues to note about the server side login module configuration:

1. The `cacheJndiName=srp-test/AuthenticationCache` configuration option tells the `SRPCacheLoginModule` the location of the `CachePolicy` that contains the `SRPServerSession` for users who have authenticated against the `SRPService`. This value corresponds to the `SRPServiceAuthenticationCacheJndiName` attribute value.

2. The configuration includes a `UsersRolesLoginModule` with the `password-stacking=useFirstPass` configuration option. You must use a second login module with the `SRPCacheLoginModule` because SRP is only an authentication technology. To set the principal's roles that in turn determine the associated permissions, a second login module must be configured to accept the authentication credentials validated by the `SRPCacheLoginModule`.

The UsersRolesLoginModule is augmenting the SRP authentication with properties file based authorization. The user's roles are obtained from the roles.properties file included in the EJB JAR.

Run the example 3 client by executing the following command from the book examples directory:

```
[examples]$ ant -Dchap=security -Dex=3 run-example
...
run-example3:
     [echo] Waiting for 5 seconds for deploy...
     [java] Logging in using the 'srp' configuration
     [java] Created Echo
     [java] Echo.echo()#1 = This is call 1
     [java] Echo.echo()#2 = This is call 2
```

In the examples/logs directory, the ex3-trace.log file contains a detailed trace of the client side of the SRP algorithm. The traces show step-by-step the construction of the public keys, challenges, session key and verification.

Observe that the client takes a long time to run, relative to the other simple examples. The reason for this is the construction of the client's public key. This involves the creation of a cryptographically strong random number, and this process takes longer when it first executes. Subsequent authentication attempts within the same VM are much faster.

Note that Echo.echo()#2 fails with an authentication exception. The client code sleeps for 15 seconds after making the first call to demonstrate the behavior of the SRPService cache expiration. The SRPService cache policy timeout has been set to 10 seconds to force this issue. As discussed in *Section 19.3, “Secure Remote Password Example”* you must set the cache timeout correctly, or handle re-authentication on failure.

Consoles and Invokers

JBoss AS ships with several administrative access points that must be secured or removed to prevent unauthorized access to administrative functions in a deployment. This chapter discusses the various administration services and how to secure them.

20.1. JMX Console

The `jmx-console.war` found in the `deploy` directory provides an HTML view into the JMX Microkernel. As such, it provides access to administrative actions like shutting down the server, stopping services, deploying new services, etc. It should either be secured like any other web application, or removed.

20.2. Admin Console

The Admin Console replaces the Web Console, and uses JBoss Operations Network security elements to secure the console. For more information, refer to the *JBoss Admin Console Quick Start User Guide*.

20.3. HTTP Invokers

The `http-invoker.sar` found in the `deploy` directory is a service that provides RMI/HTTP access for EJBs and the JNDI `Naming` service. This includes a servlet that processes posts of marshaled `org.jboss.invocation.Invocation` objects that represent invocations that should be dispatched onto the `MBeanServer`. Effectively this allows access to MBeans that support the detached invoker operation via HTTP POST requests. Securing this access point involves securing the `JMXInvokerServlet` servlet found in the `http-invoker.sar/invoker.war/WEB-INF/web.xml` descriptor. There is a secure mapping defined for the `/restricted/JMXInvokerServlet` path by default. Remove the other paths and configure the `http-invoker` security domain setup in the `http-invoker.sar/invoker.war/WEB-INF/jboss-web.xml` deployment descriptor.

Note

See the *Admin Console Quick Start Guide* for in-depth information on securing the HTTP invoker.

20.4. JMX Invoker

The `jmx-invoker-service.xml` is a configuration file that exposes the JMX MBeanServer interface via an RMI compatible interface using the RMI/JRMP detached invoker service.

20.5. Remote Access to Services, Detached Invokers

In addition to the MBean services notion that allows for the ability to integrate arbitrary functionality, JBoss also has a detached invoker concept that allows MBean services to expose functional interfaces via arbitrary protocols for remote access by clients. The notion of a detached invoker is that remoting and the protocol by which a service is accessed is a functional aspect or service independent of the component. Therefore, you can make a naming service available for use via RMI/JRMP, RMI/HTTP, RMI/SOAP, or any arbitrary custom transport.

The discussion of the detached invoker architecture will begin with an overview of the components involved. The main components in the detached invoker architecture are shown in *Figure 20.1, "The main components in the detached invoker architecture"*.

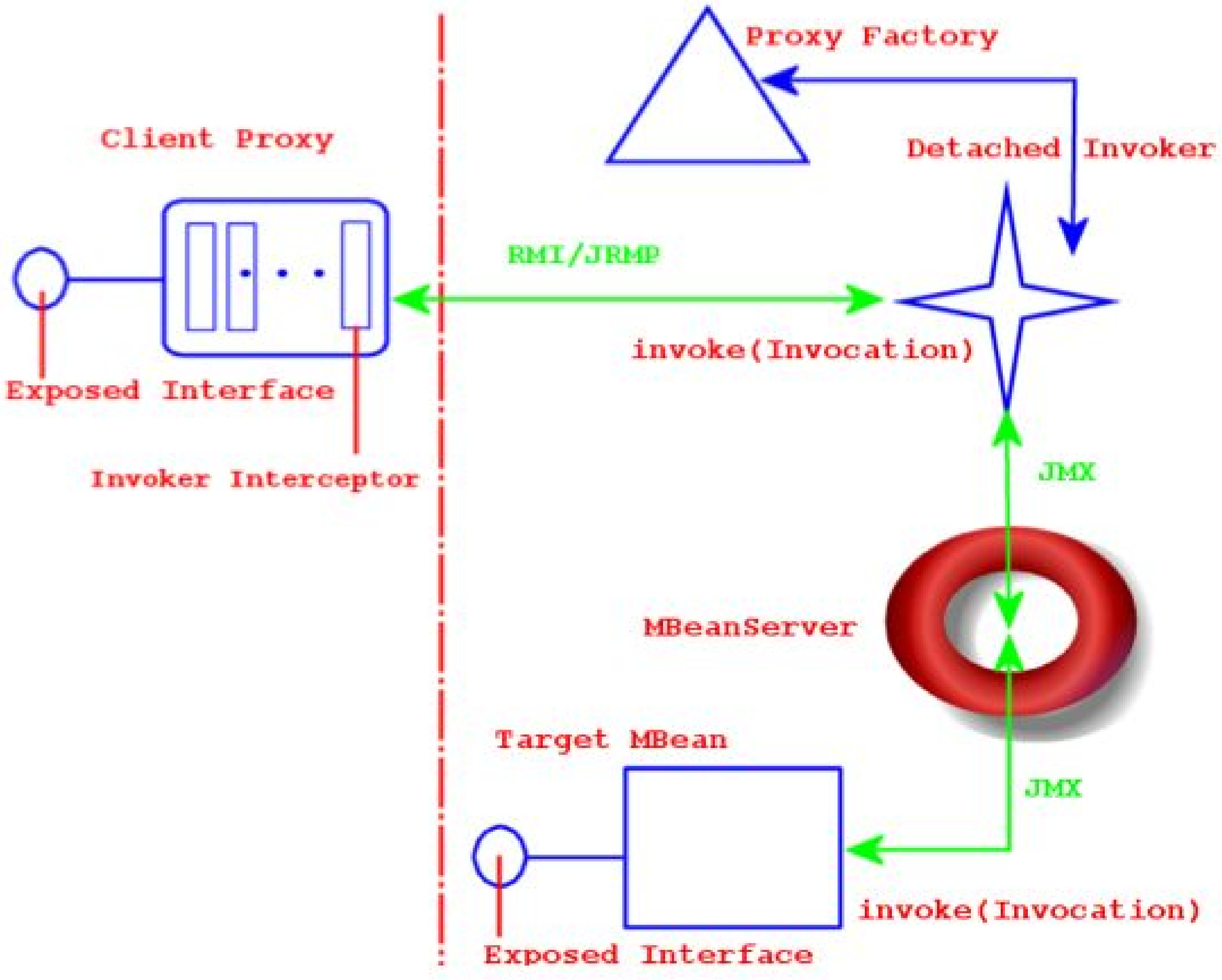

Figure 20.1. The main components in the detached invoker architecture

On the client side, there exists a client proxy which exposes the interface(s) of the MBean service. This is the same smart, compile-less dynamic proxy that is used for EJB home and remote interfaces. The only difference between the proxy for an arbitrary service and the EJB is the set of interfaces exposed as well as the client side interceptors found inside the proxy. The client interceptors are represented by the rectangles found inside of the client proxy. An interceptor is an assembly line type of pattern that allows for transformation of a method invocation and/or return values. A client obtains a proxy through some lookup mechanism, typically JNDI. Although RMI is indicated in *Figure 20.1, "The main components in the detached invoker architecture"*, the only real requirement on the exposed interface and its types is that they are serializable between the client server over JNDI as well as the transport layer.

The choice of the transport layer is determined by the last interceptor in the client proxy, which is referred to as the *Invoker Interceptor* in *Figure 20.1, “The main components in the detached invoker architecture”*. The invoker interceptor contains a reference to the transport specific stub of the server side *Detached Invoker* MBean service. The invoker interceptor also handles the optimization of calls that occur within the same VM as the target MBean. When the invoker interceptor detects that this is the case the call is passed to a call-by-reference invoker that simply passes the invocation along to the target MBean.

The detached invoker service is responsible for making a generic invoke operation available via the transport the detached invoker handles. The `Invoker` interface illustrates the generic invoke operation.

```
package org.jboss.invocation;

import java.rmi.Remote;
import org.jboss.proxy.Interceptor;
import org.jboss.util.id.GUID;


public interface Invoker
    extends Remote
{
    GUID ID = new GUID();

    String getServerHostName() throws Exception;

    Object invoke(Invocation invocation) throws Exception;
}
```

The Invoker interface extends `Remote` to be compatible with RMI, but this does not mean that an invoker must expose an RMI service stub. The detached invoker service simply acts as a transport gateway that accepts invocations represented as the `org.jboss.invocation.Invocation` object over its specific transport, unmarshalls the invocation, forwards the invocation onto the destination MBean service, represented by the *Target MBean* in *Figure 20.1, “The main components in the detached invoker architecture”*, and marshalls the return value or exception resulting from the forwarded call back to the client.

The `Invocation` object is just a representation of a method invocation context. This includes the target MBean name, the method, the method arguments, a context of information associated with the proxy by the proxy factory, and an arbitrary map of data associated with the invocation by the client proxy interceptors.

The configuration of the client proxy is done by the server side proxy factory MBean service, indicated by the *Proxy Factory* component in *Figure 20.1, “The main components in the detached invoker architecture”*. The proxy factory performs the following tasks:

- Create a dynamic proxy that implements the interface the target MBean wishes to expose.
- Associate the client proxy interceptors with the dynamic proxy handler.
- Associate the invocation context with the dynamic proxy. This includes the target MBean, detached invoker stub and the proxy JNDI name.
- Make the proxy available to clients by binding the proxy into JNDI.

The last component in *Figure 20.1, “The main components in the detached invoker architecture”* is the *Target MBean* service that wishes to expose an interface for invocations to remote clients. The steps required for an MBean service to be accessible through a given interface are:

- Define a JMX operation matching the signature: `public Object invoke(org.jboss.invocation.Invocation) throws Exception`
- Create a `HashMap<Long, Method>` mapping from the exposed interface `java.lang.reflect.Methods` to the long hash representation using the `org.jboss.invocation.MarshalledInvocation.calculateHash` method.
- Implement the `invoke(Invocation)` JMX operation and use the interface method hash mapping to transform from the long hash representation of the invoked method to the `java.lang.reflect.Method` of the exposed interface. Reflection is used to perform the actual invocation on the object associated with the MBean service that actually implements the exposed interface.

20.5.1. A Detached Invoker Example, the MBeanServer Invoker Adaptor Service

This section presents the `org.jboss.jmx.connector.invoker.InvokerAdaptorService` and its configuration for access via RMI/JRMP as an example of the steps required to provide remote access to an MBean service.

Example 20.1. The InvokerAdaptorService MBean

The `InvokerAdaptorService` is a simple MBean service that exists to fulfill the target MBean role in the detached invoker pattern.

```
package org.jboss.jmx.connector.invoker;
public interface InvokerAdaptorServiceMBean
    extends org.jboss.system.ServiceMBean
{
    Class getExportedInterface();
    void setExportedInterface(Class exportedInterface);
    Object invoke(org.jboss.invocation.Invocation invocation)
        throws Exception;
}

package org.jboss.jmx.connector.invoker;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.UndeclaredThrowableException;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

import javax.management.MBeanServer;
import javax.management.ObjectName;

import org.jboss.invocation.Invocation;
import org.jboss.invocation.MarshalledInvocation;
import org.jboss.mx.server.ServerConstants;
import org.jboss.system.ServiceMBeanSupport;
import org.jboss.system.Registry;

public class InvokerAdaptorService
    extends ServiceMBeanSupport
    implements InvokerAdaptorServiceMBean, ServerConstants
{
    private static ObjectName mbeanRegistry;

    static {
        try {
            mbeanRegistry = new ObjectName(MBEAN_REGISTRY);
        } catch (Exception e) {
            throw new RuntimeException(e.toString());
        }
    }

    private Map marshalledInvocationMapping = new HashMap();
    private Class exportedInterface;

    public Class getExportedInterface()
    {
        return exportedInterface;
    }
```

```
public void setExportedInterface(Class exportedInterface)
{
    this.exportedInterface = exportedInterface;
}

protected void startService()
    throws Exception
{
    // Build the interface method map
    Method[] methods = exportedInterface.getMethods();
    HashMap tmpMap = new HashMap(methods.length);
    for (int m = 0; m < methods.length; m ++) {
        Method method = methods[m];
        Long hash = new Long(MarshalledInvocation.calculateHash(method));
        tmpMap.put(hash, method);
    }

    marshalledInvocationMapping = Collections.unmodifiableMap(tmpMap);
    // Place our ObjectName hash into the Registry so invokers can
    // resolve it
    Registry.bind(new Integer(serviceName.hashCode()), serviceName);
}

protected void stopService()
    throws Exception
{
    Registry.unbind(new Integer(serviceName.hashCode()));
}


public Object invoke(Invocation invocation)
    throws Exception
{
    // Make sure we have the correct classloader before unmarshalling
    Thread thread = Thread.currentThread();
    ClassLoader oldCL = thread.getContextClassLoader();

    // Get the MBean this operation applies to
    ClassLoader newCL = null;
    ObjectName objectName = (ObjectName)
        invocation.getValue("JMX_OBJECT_NAME");
    if (objectName != null) {
        // Obtain the ClassLoader associated with the MBean deployment
```

```
            newCL = (ClassLoader)
                server.invoke(mbeanRegistry, "getValue",
                              new Object[] { objectName, CLASSLOADER },
                              new String[] { ObjectName.class.getName(),
                                             "java.lang.String" });
        }

        if (newCL != null && newCL != oldCL) {
            thread.setContextClassLoader(newCL);
        }

        try {
            // Set the method hash to Method mapping
            if (invocation instanceof MarshalledInvocation) {
                MarshalledInvocation mi = (MarshalledInvocation) invocation;
                mi.setMethodMap(marshalledInvocationMapping);
            }

            // Invoke the MBeanServer method via reflection
            Method method = invocation.getMethod();
            Object[] args = invocation.getArguments();
            Object value = null;
            try {
                String name = method.getName();
                Class[] sig = method.getParameterTypes();
                Method mbeanServerMethod =
                    MBeanServer.class.getMethod(name, sig);
                value = mbeanServerMethod.invoke(server, args);
            } catch(InvocationTargetException e) {
                Throwable t = e.getTargetException();
                if (t instanceof Exception) {
                    throw (Exception) t;
                } else {
                    throw new UndeclaredThrowableException(t, method.toString());
                }
            }

            return value;
        } finally {
            if (newCL != null && newCL != oldCL) {
                thread.setContextClassLoader(oldCL);
            }
        }
    }
```

```
}
```

To help understand the components that make up the `InvokerAdaptorServiceMBean`, the code has been split into logical blocks, with commentary about how each block interoperates.

Example 20.2. Block One

```
package org.jboss.jmx.connector.invoker;
public interface InvokerAdaptorServiceMBean
    extends org.jboss.system.ServiceMBean
{
    Class getExportedInterface();
    void setExportedInterface(Class exportedInterface);

    Object invoke(org.jboss.invocation.Invocation invocation)
        throws Exception;
}

package org.jboss.jmx.connector.invoker;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.UndeclaredThrowableException;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

import javax.management.MBeanServer;
import javax.management.ObjectName;

import org.jboss.invocation.Invocation;
import org.jboss.invocation.MarshalledInvocation;
import org.jboss.mx.server.ServerConstants;
import org.jboss.system.ServiceMBeanSupport;
import org.jboss.system.Registry;

public class InvokerAdaptorService
    extends ServiceMBeanSupport
    implements InvokerAdaptorServiceMBean, ServerConstants
{
    private static ObjectName mbeanRegistry;

    static {
        try {
            mbeanRegistry = new ObjectName(MBEAN_REGISTRY);
        } catch (Exception e) {
            throw new RuntimeException(e.toString());
        }
    }

    private Map marshalledInvocationMapping = new HashMap();
    private Class exportedInterface;

    public Class getExportedInterface()
    {
        return exportedInterface;
    }

    public void setExportedInterface(Class exportedInterface)
    {
        this.exportedInterface = exportedInterface;
    }
...
```

The `InvokerAdaptorServiceMBean` Standard MBean interface of the `InvokerAdaptorService` has a single `ExportedInterface` attribute and a single `invoke(Invocation)` operation.

`ExportedInterface`
: The attribute allows customization of the type of interface the service exposes to clients. This must be compatible with the `MBeanServer` class in terms of method name and signature.

`invoke(Invocation)`
: The operation is the required entry point that target MBean services must expose to participate in the detached invoker pattern. This operation is invoked by the detached invoker services that have been configured to provide access to the `InvokerAdaptorService`.

Example 20.3. Block Two

```
    protected void startService()
        throws Exception
    {
        // Build the interface method map
        Method[] methods = exportedInterface.getMethods();
        HashMap tmpMap = new HashMap(methods.length);
        for (int m = 0; m &lt; methods.length; m ++) {
            Method method = methods[m];
            Long hash = new Long(MarshalledInvocation.calculateHash(method));
            tmpMap.put(hash, method);
        }

        marshalledInvocationMapping = Collections.unmodifiableMap(tmpMap);
        // Place our ObjectName hash into the Registry so invokers can
        // resolve it
        Registry.bind(new Integer(serviceName.hashCode()), serviceName);
    }
    protected void stopService()
        throws Exception
    {
        Registry.unbind(new Integer(serviceName.hashCode()));
    }
```

This code block builds the HashMap<Long, Method> of the `exportedInterface` Class using the `org.jboss.invocation.MarshalledInvocation.calculateHash(Method)` utility method.

Because `java.lang.reflect.Method` instances are not serializable, a `MarshalledInvocation` version of the non-serializable `Invocation` class is used to marshall the invocation between the client and server. The `MarshalledInvocation` replaces the Method instances with their corresponding hash representation. On the server side, the `MarshalledInvocation` must be told what the hash to Method mapping is.

This code block creates a mapping between the `InvokerAdaptorService` service name and its hash code representation. This is used by detached invokers to determine what the target MBean `ObjectName` of an `Invocation` is.

When the target MBean name is stored in the `Invocation`, its store as its hashCode because `ObjectName`s are relatively expensive objects to create. The `org.jboss.system.Registry` is a global map like construct that invokers use to store the hash code to `ObjectName` mappings in.

Example 20.4. Block Three

```
    public Object invoke(Invocation invocation)
        throws Exception
    {
        // Make sure we have the correct classloader before unmarshalling
        Thread thread = Thread.currentThread();
        ClassLoader oldCL = thread.getContextClassLoader();

        // Get the MBean this operation applies to
        ClassLoader newCL = null;
        ObjectName objectName = (ObjectName)
            invocation.getValue(&quot;JMX_OBJECT_NAME&quot;);
        if (objectName != null) {
            // Obtain the ClassLoader associated with the MBean deployment
            newCL = (ClassLoader)
                server.invoke(mbeanRegistry, &quot;getValue&quot;,
                              new Object[] { objectName, CLASSLOADER },
                              new String[] { ObjectName.class.getName(),
                                             &quot;java.lang.String&quot; });
        }

        if (newCL != null &amp;&amp; newCL != oldCL) {
            thread.setContextClassLoader(newCL);
        }
```

This code block obtains the name of the MBean on which the MBeanServer operation is being performed, and then looks up the class loader associated with the MBean's SAR deployment. This information is available via the `org.jboss.mx.server.registry.BasicMBeanRegistry`, a JBoss JMX implementation-specific class.

It is generally necessary for an MBean to establish the correct class loading context because the detached invoker protocol layer may not have access to the class loaders needed to unmarshall the types associated with an invocation.

Example 20.5. Block Four

```
...
        try {
            // Set the method hash to Method mapping
            if (invocation instanceof MarshalledInvocation) {
                MarshalledInvocation mi = (MarshalledInvocation) invocation;
                mi.setMethodMap(marshalledInvocationMapping);
            }
...
```

This code block installs the `ExposedInterface` class method hash to method mapping if the invocation argument is of type `MarshalledInvocation`. The method mapping calculated in *Example 20.3, “Block Two”*is used here.

A second mapping is performed from the `ExposedInterface` method to the matching method of the MBeanServer class. The `InvokerServiceAdaptor` decouples the `ExposedInterface` from the `MBeanServer` class in that it allows an arbitrary interface. This is required because the standard `java.lang.reflect.Proxy` class can only proxy interfaces. It also allows you to only expose a subset of the MBeanServer methods and add transport specific exceptions such as `java.rmi.RemoteException` to the `ExposedInterface` method signatures.

Example 20.6. Block Five

```
...
            // Invoke the MBeanServer method via reflection
            Method method = invocation.getMethod();
            Object[] args = invocation.getArguments();
            Object value = null;
            try {
                String name = method.getName();
                Class[] sig = method.getParameterTypes();
                Method mbeanServerMethod =
                    MBeanServer.class.getMethod(name, sig);
                value = mbeanServerMethod.invoke(server, args);
            } catch(InvocationTargetException e) {
                Throwable t = e.getTargetException();
                if (t instanceof Exception) {
                    throw (Exception) t;
                } else {
                    throw new UndeclaredThrowableException(t, method.toString());
                }
            }

            return value;
        } finally {
            if (newCL != null &amp;&amp; newCL != oldCL) {
                thread.setContextClassLoader(oldCL);
            }
        }
    }
}
```

The code block dispatches the MBeanServer method invocation to the `InvokerAdaptorService` MBeanServer instance to which the was deployed. The server instance variable is inherited from the `ServiceMBeanSupport` superclass.

Any exceptions that result from the reflective invocation are handled, including unwrapping any declared exceptions thrown by the invocation. The MBean code completes with the return of the successful MBeanServer method invocation result.

Note

The `InvokerAdaptorService` MBean does not deal directly with any transport specific details. There is the calculation of the method hash to Method mapping, but this is a transport independent detail.

Now take a look at how the `InvokerAdaptorService` may be used to expose the same `org.jboss.jmx.adaptor.rmi.RMIAdaptor` interface via RMI/JRMP as seen in Connecting to JMX Using RMI.

We start by presenting the proxy factory and `InvokerAdaptorService` configurations found in the default setup in the `jmx-invoker-adaptor-service.sar` deployment. *Example 20.7, “Default jmx-invoker-adaptor-server.sar deployment descriptor”* shows the `jboss-service.xml` descriptor for this deployment.

Example 20.7. Default jmx-invoker-adaptor-server.sar deployment descriptor

```
<server>
    <!-- The JRMP invoker proxy configuration for the InvokerAdaptorService -->
    <mbean code="org.jboss.invocation.jrmp.server.JRMPProxyFactory"
           name="jboss.jmx:type=adaptor,name=Invoker,protocol=jrmp,service=proxyFactory">
        <!-- Use the standard JRMPInvoker from conf/jboss-service.xml -->
        <attribute name="InvokerName">jboss:service=invoker,type=jrmp</attribute>
        <!-- The target MBean is the InvokerAdaptorService configured below -->
        <attribute name="TargetName">jboss.jmx:type=adaptor,name=Invoker</attribute>
        <!-- Where to bind the RMIAdaptor proxy -->
        <attribute name="JndiName">jmx/invoker/RMIAdaptor</attribute>
        <!-- The RMI compatible MBeanServer interface -->
        <attribute name="ExportedInterface">org.jboss.jmx.adaptor.rmi.RMIAdaptor</attribute>
        <attribute name="ClientInterceptors">
            <iterceptors>
                <interceptor>org.jboss.proxy.ClientMethodInterceptor</interceptor>
                <interceptor>
                    org.jboss.jmx.connector.invoker.client.InvokerAdaptorClientInterceptor
                </interceptor>
                <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
            </iterceptors>
        </attribute>
        <depends>jboss:service=invoker,type=jrmp</depends>
    </mbean>
    <!-- This is the service that handles the RMIAdaptor invocations by routing
         them to the MBeanServer the service is deployed under. -->
    <mbean code="org.jboss.jmx.connector.invoker.InvokerAdaptorService"
           name="jboss.jmx:type=adaptor,name=Invoker">
        <attribute name="ExportedInterface">org.jboss.jmx.adaptor.rmi.RMIAdaptor</attribute>
    </mbean>
</server>
```

The first MBean, `org.jboss.invocation.jrmp.server.JRMPProxyFactory`, is the proxy factory MBean service that creates proxies for the RMI/JRMP protocol. The configuration of this service as shown in *Example 20.7, "Default jmx-invoker-adaptor-server.sar deployment descriptor"* states that the JRMPInvoker will be used as the detached invoker, the `InvokerAdaptorService` is the target mbean to which requests will be forwarded, that the proxy will expose the `RMIAdaptor` interface, the proxy will be bound into JNDI under the name `jmx/invoker/RMIAdaptor`, and the proxy will contain 3 interceptors: `ClientMethodInterceptor`, `InvokerAdaptorClientInterceptor`, `InvokerInterceptor`. The configuration of the `InvokerAdaptorService` simply sets the RMIAdaptor interface that the service is exposing.

The last piece of the configuration for exposing the `InvokerAdaptorService` via RMI/JRMP is the detached invoker. The detached invoker we will use is the standard RMI/JRMP invoker used by the EJB containers for home and remote invocations, and this is the `org.jboss.invocation.jrmp.server.JRMPInvoker` MBean service configured in the `conf/jboss-service.xml` descriptor. That we can use the same service instance emphasizes the detached nature of the invokers. The JRMPInvoker simply acts as the RMI/JRMP endpoint for all RMI/JRMP proxies regardless of the interface(s) the proxies expose or the service the proxies utilize.