JBoss Enterprise Application Platform Common Criteria Certification 5

Security Guide

for use with JBoss Enterprise Application Platform 5 Common Criteria Certification

Edition 5.1.0

Last Updated: 2017-11-20

JBoss Enterprise Application Platform Common Criteria Certification 5 Security Guide

for use with JBoss Enterprise Application Platform 5 Common Criteria Certification
Edition 5.1.0

Anil Saldhana

Jaikiran Pai

Marcus Moyses

Peter Skopek

Stephan Mueller

Jared Morgan
Engineering Content Services
jmorgan@redhat.com

Joshua Wulf
Engineering Content Services
jwulf@redhat.com

Legal Notice

Copyright © 2011 Red Hat, Inc.

This document is licensed by Red Hat under the Creative Commons Attribution-ShareAlike 3.0 Unported License. If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

The Security Guide is aimed at System Administrators and Developers, and explains how to implement security in JBoss Enterprise Application Platform 5 and its patch releases. The guide covers Java EE Declarative Security; an introduction to Java Authentication and Authorization Service; the Security Model, and Extension Architecture; managing and configuring Security Domains; replacing clear text passwords with masks in configuration files, and using SSL to secure Remote Method Invocation of EJBs.

Table of Contents

PART I. SECURITY OVERVIEW 5

CHAPTER 1. JAVA EE DECLARATIVE SECURITY OVERVIEW 6
1.1. SECURITY REFERENCES 6
1.2. SECURITY IDENTITY 7
1.3. SECURITY ROLES 9
1.4. EJB METHOD PERMISSIONS 10
1.5. ENTERPRISE BEAN SECURITY ANNOTATIONS 13
1.6. WEB CONTENT SECURITY CONSTRAINTS 14
1.7. ENABLING FORM-BASED AUTHENTICATION 16
1.8. ENABLING DECLARATIVE SECURITY 17

CHAPTER 2. INTRODUCTION TO JAAS 18
2.1. JAAS CORE CLASSES 18
2.1.1. Subject and Principal Classes 18
2.1.2. Subject Authentication 19

CHAPTER 3. JBOSS SECURITY MODEL 23
3.1. ENABLING DECLARATIVE SECURITY REVISITED 25

CHAPTER 4. THE JBOSS SECURITY EXTENSION ARCHITECTURE 30
4.1. HOW THE JAASSECURITYMANAGER USES JAAS 32
4.2. THE JAASSECURITYMANAGERSERVICE MBEAN 35
4.3. THE JAASSECURITYDOMAIN MBEAN 38

PART II. APPLICATION SECURITY 40

CHAPTER 5. OVERVIEW 41

CHAPTER 6. SECURITY DOMAIN SCHEMA 42
6.1. <AUTHENTICATION> 42
6.2. <AUTHORIZATION> 43
6.3. <MAPPING> 44

CHAPTER 7. AUTHENTICATION 46
7.1. CUSTOM CALLBACK HANDLERS 47

CHAPTER 8. AUTHORIZATION 52
8.1. MODULE DELEGATION 56

CHAPTER 9. MAPPING 58

CHAPTER 10. AUDITING 60

CHAPTER 11. DEPLOYING SECURITY DOMAINS 67

CHAPTER 12. LOGIN MODULES 69
12.1. USING MODULES 69
12.1.1. LdapLoginModule 69
12.1.2. Password Stacking 74
12.1.3. Password Hashing 74
12.1.4. Unauthenticated Identity 76
12.1.5. UsersRolesLoginModule 76
12.1.6. DatabaseServerLoginModule 77
12.1.7. BaseCertLoginModule 79

12.1.8. IdentityLoginModule 82
12.1.9. RunAsLoginModule 82
12.1.10. RunAsIdentity Creation 83
12.1.11. ClientLoginModule 84
12.2. CUSTOM MODULES 85
12.2.1. Subject Usage Pattern Support 86
12.2.2. Custom LoginModule Example 91

PART III. ENCRYPTION AND SECURITY 95

CHAPTER 13. SECURE REMOTE PASSWORD PROTOCOL 96
13.1. UNDERSTANDING THE ALGORITHM 100
13.2. CONFIGURE SECURE REMOTE PASSWORD INFORMATION 102
13.3. SECURE REMOTE PASSWORD EXAMPLE 104

CHAPTER 14. JAVA SECURITY MANAGER 108
14.1. USING THE SECURITY MANAGER 108
14.2. DEBUGGING SECURITY POLICY ISSUES 111
14.2.1. Debugging Security Manager 112
14.3. WRITING SECURITY POLICY FOR JBOSS ENTERPRISE APPLICATION PLATFORM 112

CHAPTER 15. SECURING THE EJB RMI TRANSPORT LAYER 115
15.1. SSL ENCRYPTION OVERVIEW 115
15.1.1. Key pairs and Certificates 115
15.2. GENERATE ENCRYPTION KEYS AND CERTIFICATE 116
15.2.1. Generate a self-signed certificate with keytool 116
15.2.1.1. Generate a key pair 116
15.2.1.2. Export a self-signed certificate 118
15.2.2. Configure a client to accept a self-signed server certificate 118
15.3. EJB3 RMI + SSL CONFIGURATION 119
15.4. EJB3 RMI VIA HTTPS CONFIGURATION 121
15.5. EJB2 RMI + SSL CONFIGURATION 126

CHAPTER 16. MASKING PASSWORDS IN XML CONFIGURATION 129
16.1. PASSWORD MASKING OVERVIEW 129
16.2. GENERATE A KEY STORE AND A MASKED PASSWORD 129
16.3. ENCRYPT THE KEY STORE PASSWORD 130
16.4. CREATE PASSWORD MASKS 131
16.5. REPLACE CLEAR TEXT PASSWORDS WITH THEIR PASSWORD MASKS 133
16.6. CHANGING THE PASSWORD MASKING DEFAULTS 134

CHAPTER 17. ENCRYPTING DATA SOURCE PASSWORDS 135
17.1. SECURED IDENTITY 135
17.1.1. Encrypt the data source password 135
17.1.2. Create an application authentication policy with the encrypted password 136
17.1.3. Configure the data source to use the application authentication policy 137
17.2. CONFIGURED IDENTITY WITH PASSWORD BASED ENCRYPTION (PBE) 138

CHAPTER 18. ENCRYPTING THE KEYSTORE PASSWORD IN A TOMCAT CONNECTOR 142
18.1. MEDIUM SECURITY USECASE 144

CHAPTER 19. USING LDAPEXTLOGINMODULE WITH JAASSECURITYDOMAIN 146

CHAPTER 20. FIREWALLS 148

CHAPTER 21. CONSOLES AND INVOKERS 150

21.1. JMX CONSOLE 150
21.2. ADMIN CONSOLE 150
21.3. HTTP INVOKERS 150
21.4. JMX INVOKER 150
21.5. REMOTE ACCESS TO SERVICES, DETACHED INVOKERS 150
21.5.1. A Detached Invoker Example, the MBeanServer Invoker Adaptor Service 152

APPENDIX A. SETTING THE DEFAULT JDK WITH THE /USR/SBIN/ALTERNATIVES UTILITY 162

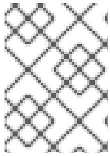
APPENDIX B. REVISION HISTORY 164

PART I. SECURITY OVERVIEW

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

The *Java Enterprise Edition* (Java EE) 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 Java EE 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.



NOTE

EJB and servlet specification security models, as well as JAAS, are covered in detail within Part I, "Security Overview".

CHAPTER 1. JAVA EE DECLARATIVE SECURITY OVERVIEW

Rather than embedding security into your business component, the Java EE 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 Enterprise Java Beans (EJBs) and servlets can declare one or more <security-role-ref> elements. Figure 1.1, “The <security-role-ref> element” describes the <security-role-ref> element, its child elements, and attributes.

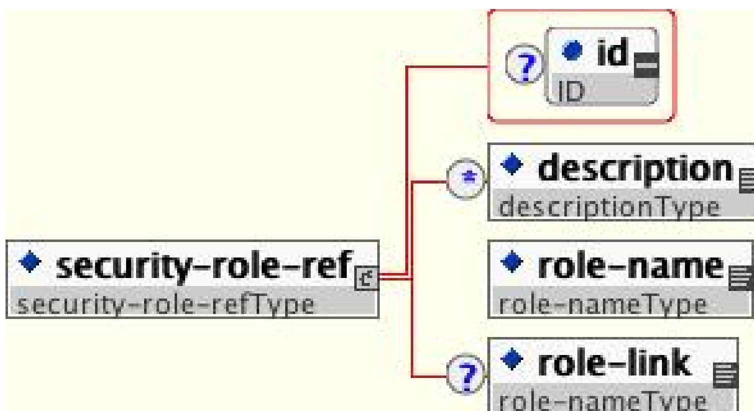


Figure 1.1. The <security-role-ref> element

This element declares that a component is using the <role-name> element's **`role-nameType`** attribute 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> or <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.

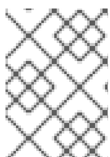
Example 1.1, “ejb-jar.xml descriptor fragment” describes the the use of <security-role-ref> in an **`ejb-jar.xml`** file.

Example 1.1. ejb-jar.xml descriptor fragment

```
<!-- 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, “web.xml descriptor fragment” shows the use of <security-role-ref> in a `web.xml` file.



NOTE

This fragment is an example only. In deployments, the elements in this section must contain role names and links relevant to the EJB deployment.

Example 1.2. web.xml descriptor fragment

```
<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 Enterprise Java Bean (EJB) can specify the identity another EJB must use when it invokes methods on components using the <security-identity> element. Figure 1.2, “The security-identity element” describes the <security-identity> element, its child elements, and attributes.

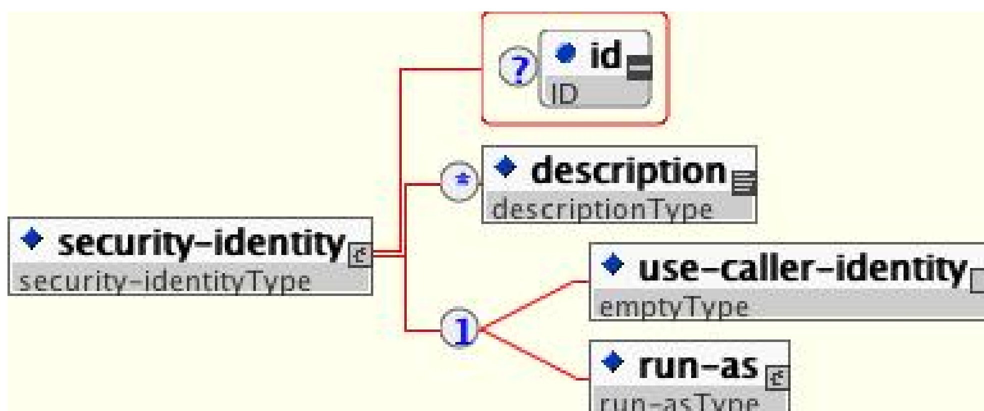


Figure 1.2. The security-identity element

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. This 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> or <role-name> child element to specify that a specific security role supplied by the <role-name> element value must 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> or <role-name> element value.

One use case for the <run-as> element is to prevent external clients from accessing internal EJBs. You configure this behavior by assigning the internal EJB <method-permission> elements, which restrict access to a role never assigned to an external client. EJBs that must in turn use internal EJBs are then configured with a <run-as> or <role-name> equal to the restricted role. The following descriptor fragment describes an example<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, a principal named **anonymous** is assigned to all outgoing calls. If you want another principal to be associated with the call, you must 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 are 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 above.

```
<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 must map to one of the application's declared roles. An application assembler defines logical security roles by declaring <security-role> elements. The role-name attribute value is a logical application role name, such as Administrator, Architect, or Sales_Manager.

The Java EE 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 Java EE 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 Bank_Manager, Teller, or Customer.

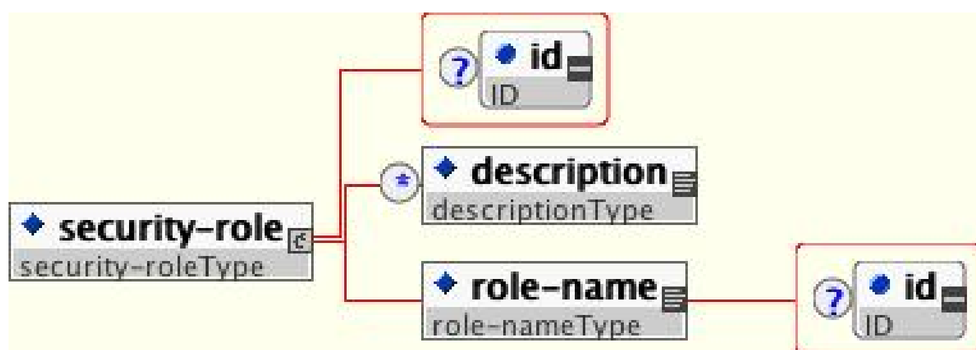


Figure 1.3. <security-role> element

In JBoss, a <security-role> element is only used to map <security-role-ref> or <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, “ejb-jar.xml descriptor fragment” describes the usage of the <security-role> in an **ejb-jar.xml** file.

Example 1.3. ejb-jar.xml descriptor fragment

```
<!-- 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, “example web.xml descriptor fragment” shows the usage of the <security-role> in an **web.xml** file.

Example 1.4. example web.xml descriptor fragment

```
<!-- 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 use <method-permission> element declaration to set the roles that are allowed to invoke an EJB's home and remote interface methods.

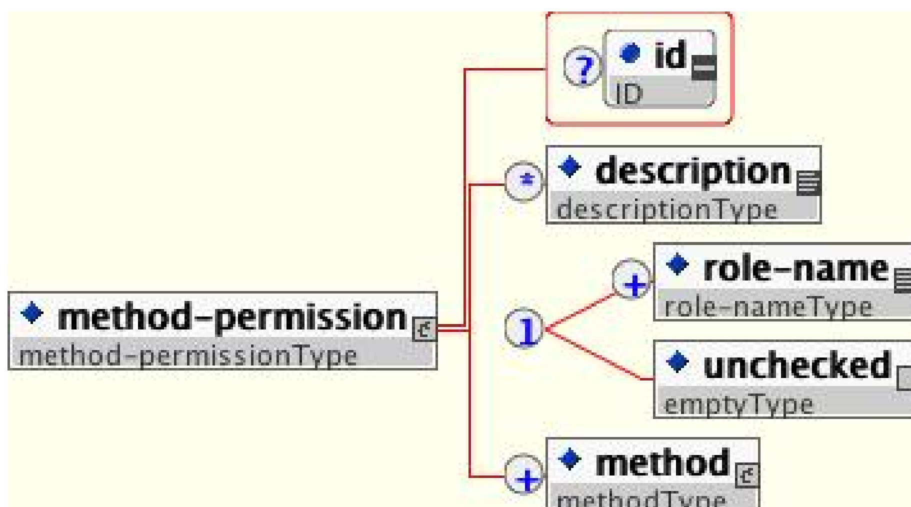


Figure 1.4. The <method-permission> element

Each <method-permission> element contains one or more <role-name> child elements. <role-name> defines 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`.

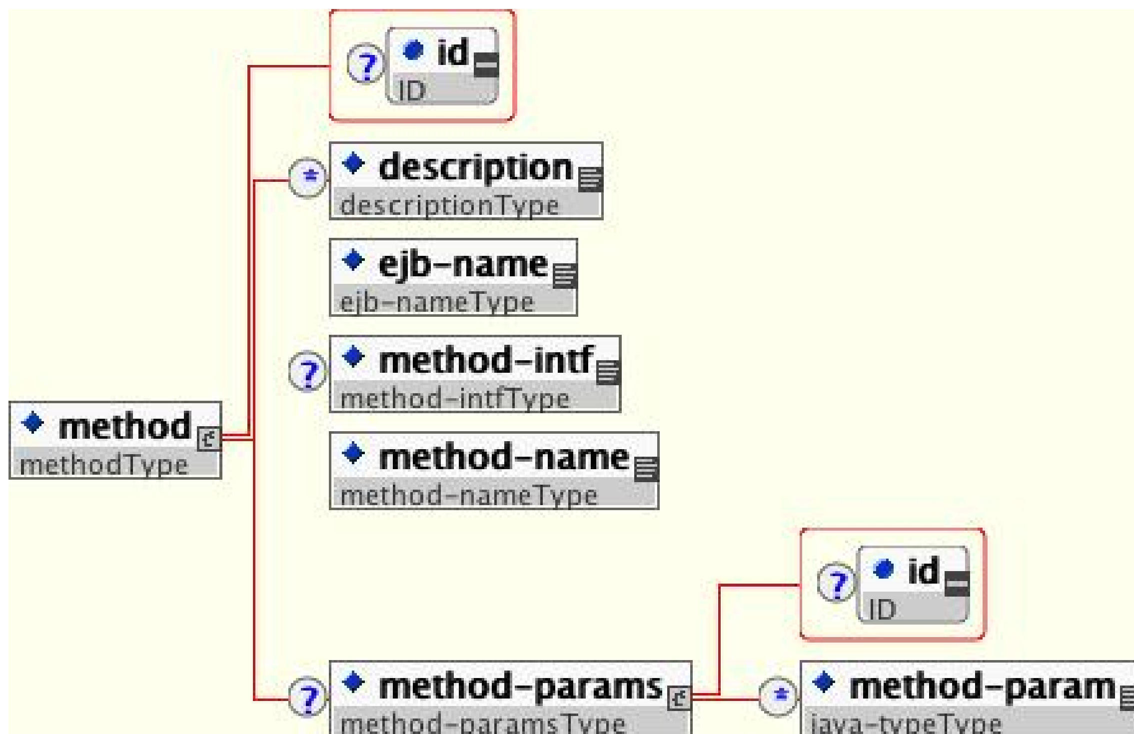


Figure 1.5. <method> element

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, "<method-permission> element usage" provides complete examples of the <method-permission> element usage.

Example 1.5. <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. ENTERPRISE BEAN SECURITY ANNOTATIONS

Enterprise beans use Annotations to pass information to the deployer about security and other aspects of the application. The deployer can set up the appropriate enterprise bean security policy for the application if specified in annotations, or the deployment descriptor.

Any method values explicitly specified in the deployment descriptor override annotation values. If a method value is not specified in the deployment descriptor, those values set using annotations are used. The overriding granularity is on a per-method basis

Those annotations that address security and can be used in an enterprise beans include the following:

@DeclareRoles

Declares each security role declared in the code. For information about configuring roles, refer to the *Java EE 5 Tutorial* Declaring Security Roles Using Annotations .

@RolesAllowed, @PermitAll, and @DenyAll

Specifies method permissions for annotations. For information about configuring annotation method permissions, refer to the *Java EE 5 Tutorial* Specifying Method Permissions Using Annotations.

@RunAs

Configures the propagated security identity of a component. For information about configuring propagated security identities using annotations, refer to the *Java EE 5 Tutorial* Configuring a Component's Propagated Security Identity.

1.6. 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.xml` security-constraint element.

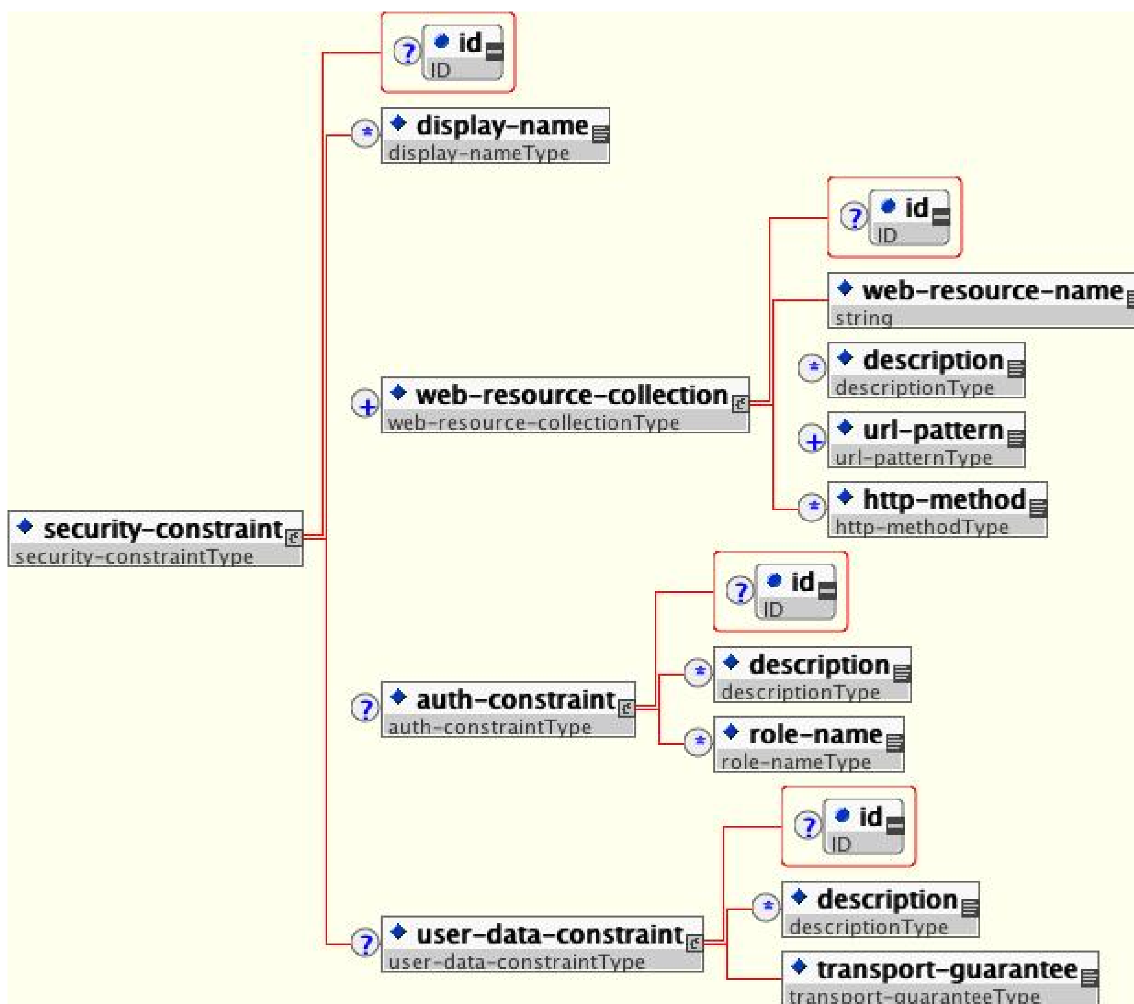


Figure 1.6. <security-constraint> element

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 not 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 the application, and the attributes that are needed by the form login mechanism.

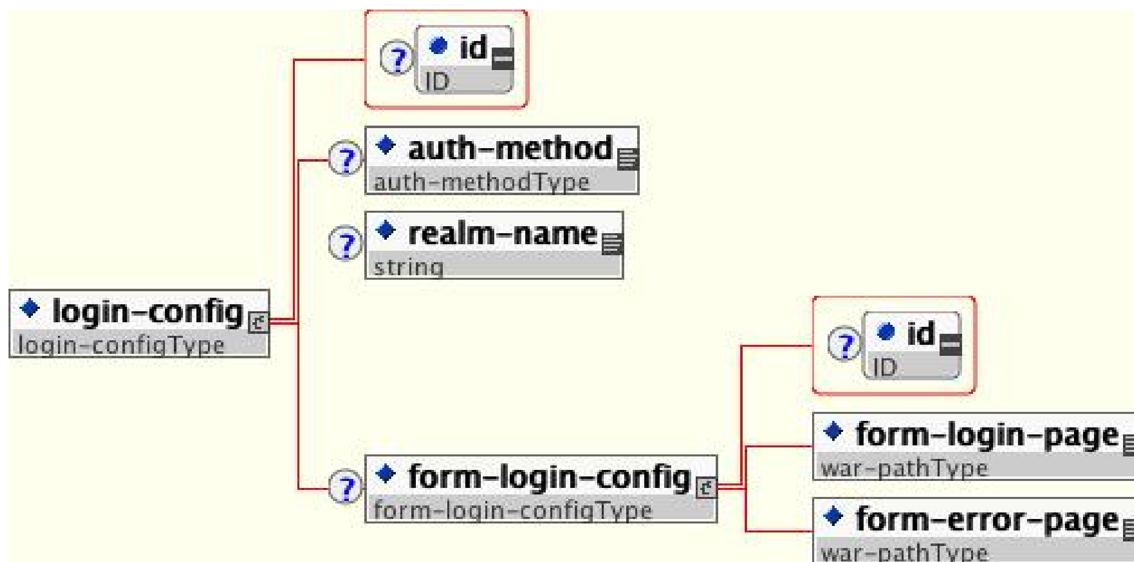


Figure 1.7. <login-config> element

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.

Example 1.6, “ web.xml descriptor fragment” 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. web.xml descriptor fragment

```
<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>
```

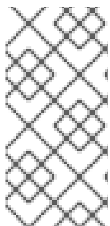
1.7. ENABLING FORM-BASED AUTHENTICATION

Form-based authentication provides flexibility in defining a custom JSP/HTML page for login, and a separate page to which users are directed if an error occurs during login.

Form-based authentication is defined by including **`<auth-method>FORM</auth-method>`** in the <login-config> element of the deployment descriptor, **`web.xml`**. The login and error pages are also defined in <login-config>, as follows:

```
<login-config>
  <auth-method>FORM</auth-method>
  <form-login-config>
    <form-login-page>/login.html</form-login-page>
    <form-error-page>/error.html</form-error-page>
  </form-login-config>
</login-config>
```

When a web application with form-based authentication is deployed, the web container uses **`FormAuthenticator`** to direct users to the appropriate page. JBoss Enterprise Application Platform maintains a session pool so that authentication information does not need to be present for each request. When **`FormAuthenticator`** receives a request, it queries **`org.apache.catalina.session.Manager`** for an existing session. If no session exists, a new session is created. **`FormAuthenticator`** then verifies the credentials of the session.



NOTE

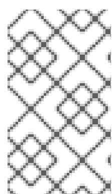
Each session is identified by a session ID, a 16 byte string generated from random values. These values are retrieved from **`/dev/urandom`** (Linux) by default, and hashed with MD5. Checks are performed at session ID creation to ensure that the ID created is unique.

Once verified, the session ID is assigned as part of a cookie, and then returned to the client. This cookie is expected in subsequent client requests and is used to identify the user session.

The cookie passed to the client is a name value pair with several optional attributes. The identifier attribute is called **JSESSIONID**. Its value is a hex-string of the session ID. This cookie is configured to be non-persistent. This means that on the client side it will be deleted when the browser exits. On the server side, sessions expire after 60 seconds of inactivity, at which time session objects and their credential information are deleted.

Say a user attempts to access a web application that is protected with form-based authentication. **FormAuthenticator** caches the request, creates a new session if necessary, and redirects the user to the login page defined in **login-config**. (In the previous example code, the login page is **login.html**.) The user then enters their user name and password in the HTML form provided. User name and password are passed to **FormAuthenticator** via the **j_security_check** form action.

The **FormAuthenticator** then authenticates the user name and password against the realm attached to the web application context. In JBoss Enterprise Application Platform, the realm is **JBossWebRealm**. When authentication is successful, **FormAuthenticator** retrieves the saved request from the cache and redirects the user to their original request.



NOTE

The server recognizes form authentication requests only when the URI ends with **/j_security_check** and at least the **j_username** and **j_password** parameters exist.

1.8. ENABLING DECLARATIVE SECURITY

The Java EE security elements that have been covered so far describe the security requirements only from the application's perspective. Because Java EE security elements declare logical roles, the application deployer maps the roles from the application domain onto the deployment environment. The Java EE specifications omit these application server-specific details.

To map application roles onto the deployment environment, you must specify a security manager that implements the Java EE security model using JBoss server specific deployment descriptors. The details behind the security configuration are discussed in Example 12.11, “ JndiUserAndPass Custom Login Module”.

CHAPTER 2. INTRODUCTION TO JAAS

The JBossSX framework is based on the JAAS API. You must understand the basic elements of the JAAS API before you can 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 guide.

The JAAS 1.0 API consists of a set of Java packages designed for user authentication and authorization. The API implements a Java version of the standard Pluggable Authentication Modules (PAM) framework and extends the Java 2 Platform 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 1.5. Because the JBossSX framework only uses 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 is achievable without changing the JBossSX security manager implementation. You need only change the configuration of the authentication stack JAAS uses.

2.1. 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. Subject and Principal Classes

To authorize access to resources, applications must first 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 user name 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) {...}
```

getPrincipals() returns all principals contained in the subject. **getPrincipals(Class c)** 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. Subject Authentication

Subject Authentication 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 **LoginModule**s.

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 **LoginModule**s. 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 **LoginModule**s 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 **LoginModules** to allow for more than one authentication technology to participate in the authentication process. For example, one **LoginModule** may perform user name/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 user name 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 value of 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 user names 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 send the authentication information 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 **LoginModule**s 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 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. **LoginModule**s 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.

CHAPTER 3. 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, "Security Model Interface Relationships to JBoss Server EJB Container Elements." shows a class diagram of the security interfaces and their relationship to the EJB container architecture.

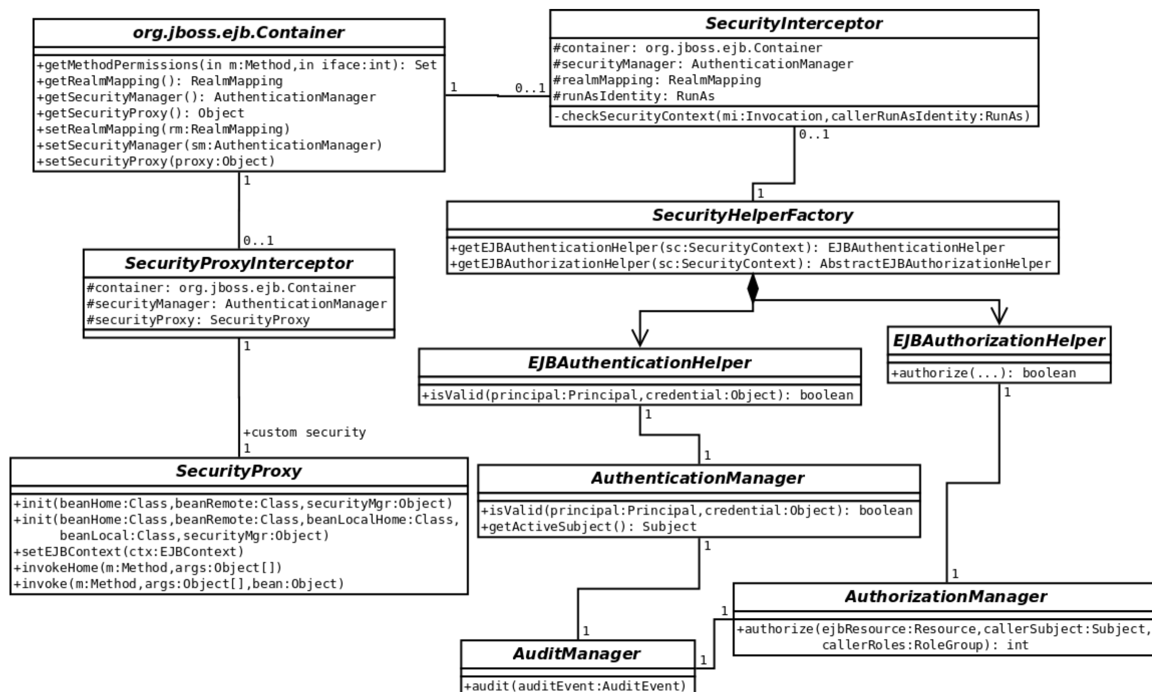


Figure 3.1. Security Model Interface Relationships to JBoss Server EJB Container Elements.

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, "Security Model Interface Relationships to JBoss Server EJB Container Elements." are summarized below.

Security Interface Roles

AuthenticationManager

This interface is responsible for validating credentials associated with *Principals*. Principals are identities, such as user names, 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 Java EE 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.

SecurityDomain

This is an extension of the **AuthenticationManager**, RealmMapping, and **SubjectSecurityManager** interfaces. **SecurityDomain** is the recommended way to implement security in components, because of the advantages the JAAS Subject offers, and the increased support offered to to ASP-style application and resource deployments. A **java.security.KeyStore**, and the JSSE **com.sun.net.ssl.KeyManagerFactory** and **com.sun.net.ssl.TrustManagerFactory** interfaces are included in the class.

RealmMapping

This interface is responsible for principal mapping and role mapping. The **getPrincipal** method takes a 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 Java EE security model are not. The JBossSX framework is simply an implementation of the basic security plug-in interfaces that are based on JAAS.

The component diagram in Figure 3.2, “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, “JBossSX Framework Implementation Classes and the JBoss Server EJB Container Layer.”.

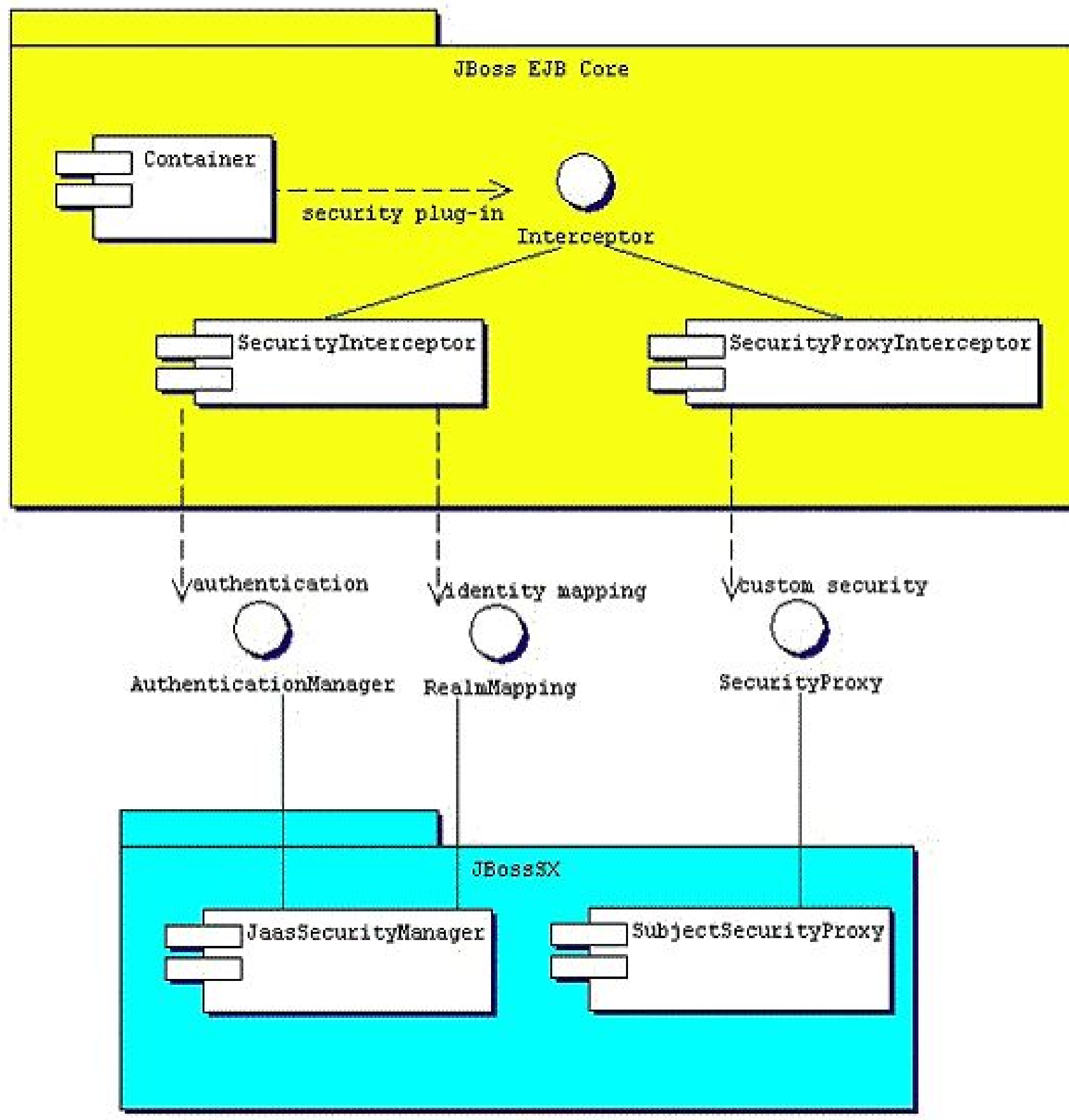


Figure 3.2. JBossSX Framework Implementation Classes and the JBoss Server EJB Container Layer.

3.1. ENABLING DECLARATIVE SECURITY REVISITED

Earlier in this chapter, the discussion of the Java EE 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, “jboss.xml and jboss-web.xml Security Element Subsets.” shows the JBoss-specific EJB and web application deployment descriptor's security-related elements.

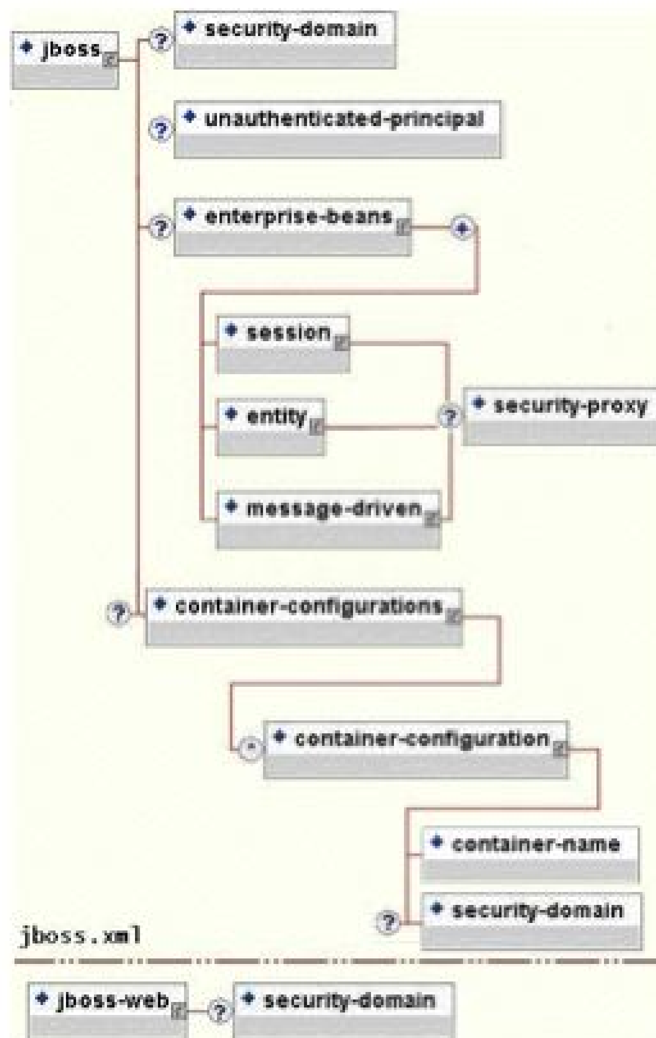


Figure 3.3. jboss.xml and jboss-web.xml Security Element Subsets.

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 is specified 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. It allows per-request security checks outside the scope of the EJB declarative security model, without embedding security logic into the EJB implementation. You can use an implementation of the `org.jboss.security.SecurityProxy` interface. Alternatively, you can use a common interface that uses an object to implement methods in the home, remote, local home, or local interfaces of the EJB. 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, "Custom EchoSecurityProxy Implementation."

Example 3.1. Custom EchoSecurityProxy Implementation.

```
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 find 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, “jboss.xml descriptor”.

Example 3.2. jboss.xml descriptor

```
<jboss>
    <security-domain>java:/jaas/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.

CHAPTER 4. THE JBOSS SECURITY EXTENSION ARCHITECTURE

The preceding discussion of the general JBoss security layer has stated that the *JBoss Security Extension framework* (JBossSX) is an implementation of the security layer interfaces. This is the primary purpose of the JBossSX framework. The framework offers a great deal of customization possibilities 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* (PAM) 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, “Relationship between <security-domain> deployment descriptor value, component container, and 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.

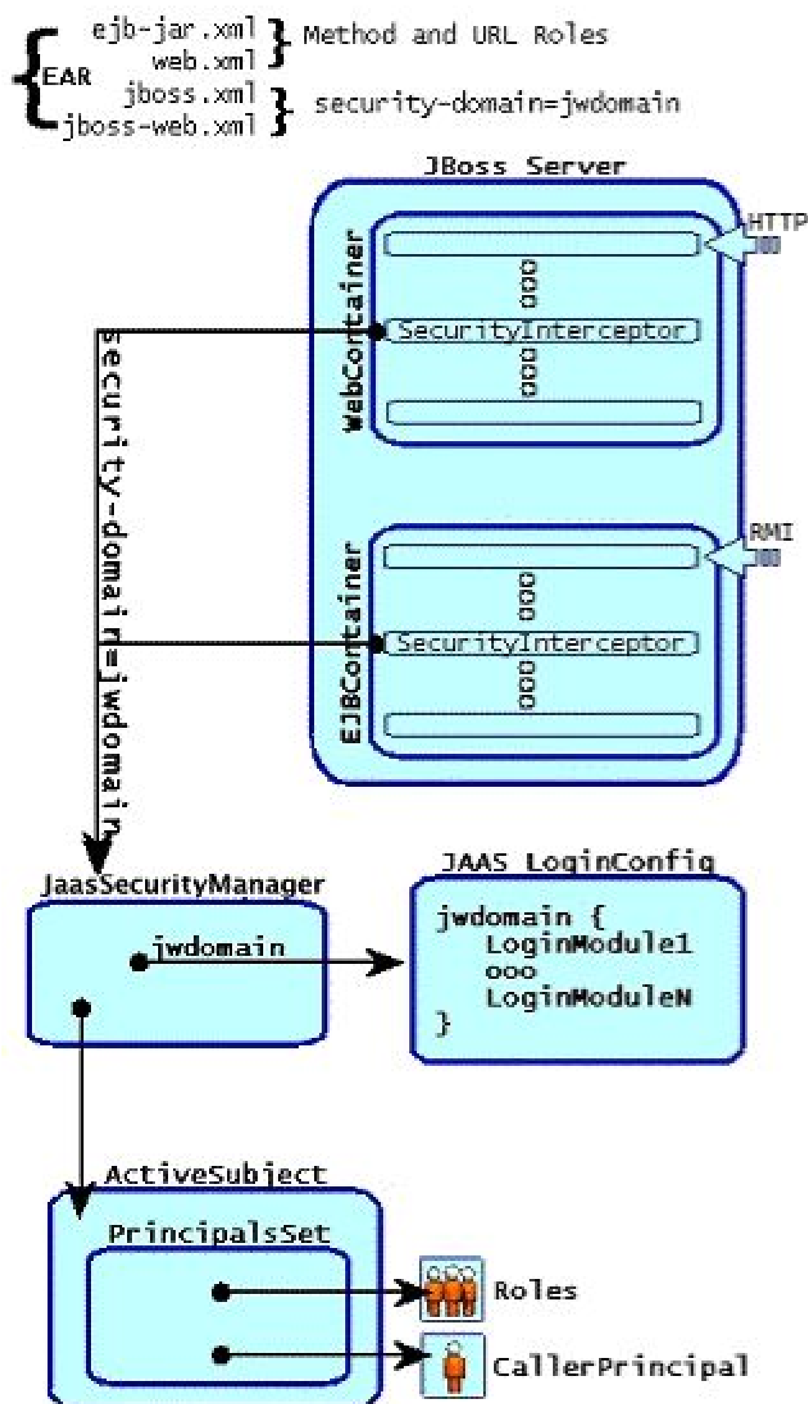


Figure 4.1. Relationship between <security-domain> deployment descriptor value, component container, and JaasSecurityManager.

Figure 4.1, “Relationship between <security-domain> deployment descriptor value, component container, and 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.

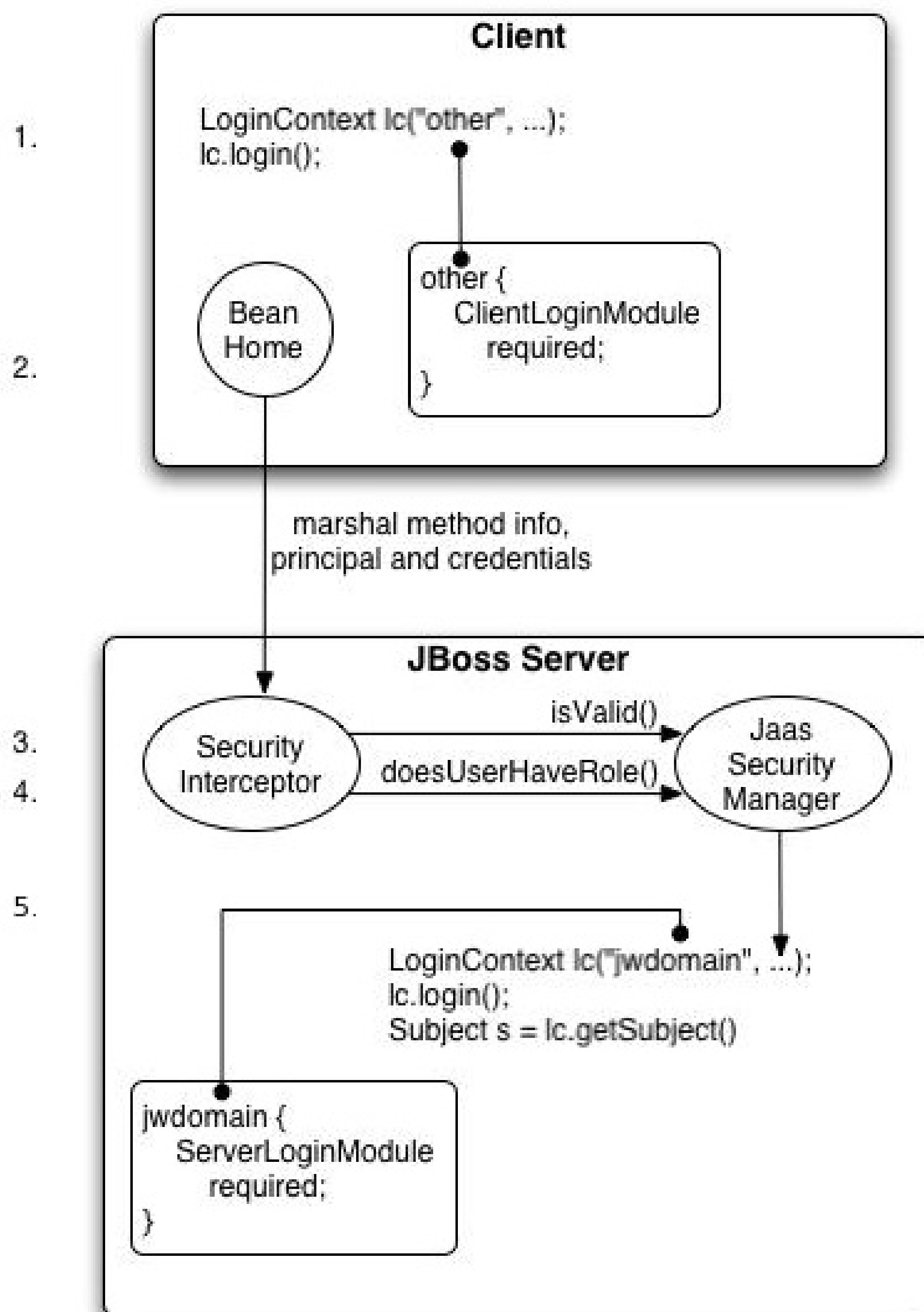


Figure 4.2. Secured EJB Home Method Authentication and Authorization Invocation Steps.

Figure 4.2, "Secured EJB Home Method Authentication and Authorization Invocation Steps." provides a view of the client to server communication. The numbered steps shown are:

1. The client must 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 in 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 user name 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. 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, “Secured EJB Home Method Authentication and Authorization Invocation Steps.” . 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.

- For JBoss Web connections, Tomcat manages elements of security constraint, and role verification.

  When a request is received for a web connection, Tomcat checks the security constraints defined in **web.xml** that match the requested resource and the accessed HTTP method.

  If a constraint exists for the request, Tomcat calls the JaasSecurityManager to perform the principal authentication, which in turn ensures the user roles are associated with that principal object.

  Role verification is performed solely by Tomcat, including checks on parameters such as **allRoles**, and whether **STRICT_MODE** is used.

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.

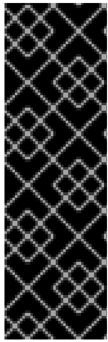
4.2. THE JAASSECURITYMANAGERSERVICE MBEAN

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. 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 permits 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.

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.



IMPORTANT

In Enterprise Application Platform versions prior to v5.0, the "***java:/jaas*** prefix in each **<security-domain>** deployment descriptor element was required to correctly bind the JNDI name of a security domain to the security manager bindings.

As of JBoss Enterprise Application Platform 5, the ***java:/jaas*** prefix is not required for security domain declaration. The **java:/jaas** prefix is still supported, and remains for backward compatibility.

For example, consider the following container security configuration snippet:

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

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

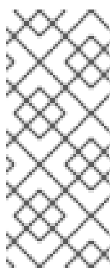
The **JaasSecurityManagerService** MBean is configured by default for use in the standard JBoss distribution, and you can often use the default configuration as is. The configurable attributes of the **JaasSecurityManagerService** include:

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**.



NOTE

You can override the handler used by the **JaasSecurityManager** if the default implementation (**org.jboss.security.auth.callback.SecurityAssociationHandler**) does not meet your needs. Most implementations will find the default handler is sufficient.

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 trade-off 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 time stamp 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.

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.

4.3. THE JAASSECURITYDOMAIN MBEAN

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 does not contain a name/value URL, the value 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 plain text 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.

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** is a simple password and does not have the same configuration options as the **KeyStorePass**.

ManagerServiceName

Sets the JMX object name string of the security manager service MBean. This is used to register the defaults to register the **JaasSecurityDomain** as a the security manager under **java:/jaas/<domain>** where **<domain>** is the name passed to the MBean constructor. The name defaults to **jboss.security:service=JaasSecurityManager**.

PART II. APPLICATION SECURITY

CHAPTER 5. OVERVIEW

Application security encompasses authentication, authorization, mapping and auditing.

Authentication

Process by which the server determines whether a user should be able to access a system or operation.

Authorization

Process by which the server determines whether an authenticated user has permission to access specific privileges or resources in the system or operation.

Mapping

Process by which the server associates authenticated users with predefined authorization profiles.

Auditing

Process by which the server monitors authentication and authorization security events.

In JBoss Enterprise Application Platform 5, authentication, authorization, and mapping policies are configured with application-level granularity using the concept of a security domain.

Security Domain

A set of authentication, authorization, and mapping policies which are defined in XML and are available to applications at runtime using Java Naming and Directory Interface (JNDI).

A security domain can be defined in the server profile or in an application deployment descriptor.

Auditing for EJB3 and the web container are configured independently of each other, and operate at the server level.

JBoss Enterprise Application Platform 5 uses a pluggable module framework to implement security, providing separation of security implementation and application design. The pluggable module framework provides flexibility in application deployment, configuration, and future development. Modules implementing security functionality are plugged into the framework and provide security functions to applications. Applications plug into the security framework through security domains.

Applications can be deployed in new scenarios with altered security configuration by associating them with a different security domain. New security methods can be implemented by plugging JBoss, custom-built, or third-party modules into the framework. Applications gain the security functionality of that module with no application recoding, by simple reconfiguration of the security domain.

Chapter 6, *Security Domain Schema* describes the XML configuration of a security domain.

Chapter 7, *Authentication* contains detailed information about security domain authentication policy.

Chapter 8, *Authorization* contains detailed information about security domain authorization policy.

Chapter 9, *Mapping* contains detailed information about security domain mapping policy.

CHAPTER 6. SECURITY DOMAIN SCHEMA

The security domain schema is constructed using XML.

The XSD that defines the structure of the security domain is declared in the security-beans_1_0.xsd file. The location of the file varies depending on which version of the JBoss Enterprise Application Platform you use.

5.1

`/schema/security-beans_1_0.xsd` inside the `jboss-as/lib/jbosssx.jar/`

5.0, 5.0.1

`/schema/security-beans_1_0.xsd` inside the `jboss-as/common/lib/jbosssx.jar`

The schema is the same regardless of which JBoss Enterprise Application Platform server you use.

Example 6.1. Security Domain Schema

```
<application-policy xmlns="urn:jboss:security-beans:1.0" name="">
  <authentication>
    <login-module code="" flag"
[required|requisite|sufficient|optional]" extends="">
      <module-option name=""></module-option>
    </login-module>
  </authentication>
  <authorization>
    <policy-module code="" flag="
[required|requisite|sufficient|optional]"/>
  </authorization>
  <mapping>
    <mapping-module code="" type"" />
  </mapping>
</application-policy>
```

Security Domain Element Descriptions

<application-policy>

The elements of a security domain, regardless of how it is deployed within the system, are contained within the <application-policy> element. The element uses the XML namespace, as declared in the ***xmlns*** attribute.

The ***name*** attribute sets the name of the security domain referenced by an application. The security domain name is bound in JNDI under the the `java:/jaas` context, and is accessed by applications via reference in their deployment descriptors.

An <application-policy> element can contain a number of child elements that set the behavior for the security domain and all applications that use it. These elements are described in greater detail in Section 6.1, "<authentication>", Section 6.2, "<authorization>", and Section 6.3, "<mapping>".

6.1. <AUTHENTICATION>

The <authentication> element contains the following child elements.

<authentication>

This element contains <login-module> elements, that control which authentication modules are used when authenticating users who connect through the application.

When multiple <login-module> elements are present, they form a collective group of requirements that must be met before authentication is verified. This collective group is called a *stack*.

<login-module>

This element uses the ***code*** attribute to specify what login module implementation an application can use, and the ***flag*** attribute to tell the application how to parse each login module present in the stack. The ***flag*** attribute supports the following values:

required

The module must succeed for authentication to be successful. If any required <login-module> fails, the authentication will fail. The remaining login modules in the stack are called regardless of the outcome of the authentication.

requisite

The module is required to succeed. If it succeeds, authentication continues down the stack. If the module 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 the module fails, authentication continues down the stack.

optional

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

Each <login-module> contains a set of <module-option> elements that further define settings required by the login module implementation.

<module-option>

Each login module has its own set of configuration options. The name attribute specifies the property required by the login module, and the value is declared in the CDATA of the <module-option> element. Module options depend on what login module you choose. Section 12.1, “Using Modules” covers module options in greater detail.

6.2. <AUTHORIZATION>

<authorization>

This element contains <policy-module> elements that define the policy module used to authorize application users, and whether the module is required:

When multiple <policy-module> elements are present, they form a collective group of requirements that must be met before authorization is verified. This collective group is called a *stack*.

<policy-module>

This element uses the ***code*** attribute to specify what policy module implementation an application can use, and the ***flag*** attribute to tell the application how to parse each policy module present in the policy stack. The ***flag*** attribute supports the following values:

required

The module must succeed for authorization to be successful. If any required <policy-module> fails, the authorization attempt will fail. The remaining modules in the stack are called regardless of the outcome of the module.

requisite

The module is required to succeed. If it succeeds, authorization continues down the 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, authorization continues down the stack.

optional

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

6.3. <MAPPING>

<mapping>

This element contains <mapping-module> elements that are used to define the parameters of the mapping-module elements

When multiple <mapping-module> elements are present, they form a collective group of requirements that must be met before mapping can succeed. This collective group is called a *stack*.

<mapping-module>

This element uses the ***code*** attribute to specify what mapping module implementation an application can use, and the ***flag*** attribute to tell the application how to parse each mapping module present in the policy stack. The flag attribute supports the following values:

required

The module must succeed for mapping to be successful. If any required <mapping-module> fails, the authentication will fail. The remaining modules in the stack are called regardless of the outcome of the authentication.

requisite

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

sufficient

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

optional

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

CHAPTER 7. AUTHENTICATION

The following examples describe ways you can use application policies in a security domain.

For clarity, only the authentication policy is declared in the examples, however you can include <authorization>, and <mapping> elements in the same <application-policy>. Refer to Section 6.1, "<authentication>" for detailed information about the <authentication> element.

Example 7.1. Single login stack authentication policy

This example describes a simple security domain configuration named **jmx-console** that uses a single login module, **UsersRolesLoginModule** (refer to Section 12.1.5, "UsersRolesLoginModule").

The login module is supplied user and role properties from files in the **jboss-as/server/*$PROFILE*/conf/props** directory.

In this instance, the <login-module> must succeed or authentication fails.

```
<application-policy xmlns="urn:jboss:security-beans:1.0 name="jmx-
console">
   <authentication>
       <login-module
code="org.jboss.security.auth.spi.UsersRolesLoginModule"
flag="required">
          <module-option name="usersProperties">props/jmx-console-
users.properties</module-option>
          <module-option name="rolesProperties">props/jmx-console-
roles.properties</module-option>
       </login-module>
   </authentication>
</application-policy>
```

Example 7.2. Multiple login stack authentication policy

This example describes a security domain configuration named **web-console** that uses two login modules in the authentication login module stack.

One <login-module> obtains login credentials using the **LdapLoginModule** (refer to Section 12.1.1, "LdapLoginModule"), whereas the other <login-module> obtains authentication credentials using **BaseCertLoginModule** (refer to Section 12.1.7, "BaseCertLoginModule").

In this instance, both modules are marked as sufficient, therefore only one of them must succeed for authentication to be successful.

```
<application-policy xmlns="urn:jboss:security-beans:1.0 name="web-
console">
    <authentication>
    <!-- LDAP configuration -->
    <login-module code="org.jboss.security.auth.spi.LdapLoginModule"
         flag="sufficient" />
    <!-- database configuration -->
    <login-module
```

```
code="org.jboss.security.auth.spi.BaseCertLoginModule"
                      flag="sufficient" />

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

7.1. CUSTOM CALLBACK HANDLERS

Implementing callback handlers into authentication procedures allows a login module to authenticate a user independent of the client application authentication method.

You can implement callback handlers using the following methods:

- Specify the CallbackHandlerClassName attribute in the **conf/jboss-service.xml** JaasSecurityManagerService MBean definition.
- Inject a callback handler instance into the **deploy/security/security-jboss-beans.xml** JNDISecurityManagement bean.

Procedure 7.1. Set callback handler using attributes

This procedure describes how to specify a callback handler in the **jboss-service.xml** configuration file.

1. **Open the configuration file**
   Navigate to ***$JBOSS_HOME*/server/*$PROFILE*/conf/**

   Open the **jboss-service.xml** file.

   By default, the **jboss-service.xml** file contains the configuration in Example 7.3, “jboss-service default configuration”

   Example 7.3. jboss-service default configuration

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

   <!--
   ==================================================================
   == -->
   <!-- Security
   -->
   <!--
   ==================================================================
   == -->

   <!-- JAAS security manager and realm mapping -->
     <mbean
   code="org.jboss.security.plugins.JaasSecurityManagerService"
   name="jboss.security:service=JaasSecurityManager">
     <!-- A flag which indicates whether the SecurityAssociation
   server mode
     is set on service creation. This is true by default since the
   ```

```
  SecurityAssociation should be thread local for multi-threaded
server
  operation.-->
  <attribute name="ServerMode">true</attribute>

  <attribute
name="SecurityManagerClassName">org.jboss.security.plugins.JaasSec
urityManager</attribute>

  <attribute
name="DefaultUnauthenticatedPrincipal">anonymous</attribute>

  <!-- DefaultCacheTimeout: Specifies the default timed cache
policy timeout
  in seconds.
  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.-->

  <attribute name="DefaultCacheTimeout">1800</attribute>

  <!-- 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. This has no affect if
the
  AuthenticationCacheJndiName has been changed from the default
value.-->

  <attribute name="DefaultCacheResolution">60</attribute>

  <!-- DeepCopySubjectMode: This set the copy mode of subjects
done by the
  security managers to be deep copies that makes copies of the
subject
  principals and credentials if they are cloneable. It should be
set to
  true if subject include mutable content that can be corrupted
when
  multiple threads have the same identity and cache flushes/logout
clearing
  the subject in one thread results in subject references
affecting other
  threads.-->

  <attribute name="DeepCopySubjectMode">false</attribute>

  </mbean>

...
```

2. **Append the attribute**
   To set the custom callback handler, append an <attribute> element as a child of the <mbean> element, and specify the fully qualified name of your callback handler. Refer to Example 7.4, “jboss-service appended callback handler” for an example <attribute> element, with the callback handler specified.

Example 7.4. jboss-service appended callback handler

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

<!--
================================================================
== -->
<!-- Security
-->
<!--
================================================================
== -->

<!-- JAAS security manager and realm mapping -->
  <mbean
code="org.jboss.security.plugins.JaasSecurityManagerService"
name="jboss.security:service=JaasSecurityManager">
  <!-- A flag which indicates whether the SecurityAssociation
server mode
  is set on service creation. This is true by default since the
  SecurityAssociation should be thread local for multi-threaded
server
  operation.-->
  <attribute name="ServerMode">true</attribute>

  <attribute
name="SecurityManagerClassName">org.jboss.security.plugins.JaasSec
urityManager</attribute>

  <attribute
name="DefaultUnauthenticatedPrincipal">anonymous</attribute>

  <!-- DefaultCacheTimeout: Specifies the default timed cache
policy timeout
  in seconds.
  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.-->

  <attribute name="DefaultCacheTimeout">1800</attribute>

  <!-- 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. This has no affect if
the
  AuthenticationCacheJndiName has been changed from the default
value.-->

  <attribute name="DefaultCacheResolution">60</attribute>

  <!-- DeepCopySubjectMode: This set the copy mode of subjects
done by the
  security managers to be deep copies that makes copies of the
subject
  principals and credentials if they are cloneable. It should be
set to
  true if subject include mutable content that can be corrupted
when
  multiple threads have the same identity and cache flushes/logout
clearing
  the subject in one thread results in subject references
affecting other
  threads.-->

  <attribute name="DeepCopySubjectMode">false</attribute>

  <attribute
name="CallbackHandlerClassName">org.jboss.security.plugins.
[Custom_Callback_Handler_Name]</attribute>

  </mbean>

...
```

3. **Restart server**
   You have now configured the **jboss-service.xml** file to use a custom callback handler.

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

Procedure 7.2. Set security callback handler using injection

This procedure describes how to inject a security callback handler instance into the JNDISecurityManagement bean.

1. **Create custom callback instance**
   You must create an instance of the custom callback handler, and register it.

2. **Open the configuration file**
   Navigate to **$JBOSS_HOME/server/*$PROFILE*/deploy/security/**

   Open the **security-jboss-beans.xml** file.

By default, the **security-jboss-beans.xml** file contains the JNDIBasedSecurityManagement bean configuration in Example 7.5, “security-jboss-beans default configuration”

Example 7.5. security-jboss-beans default configuration

```
<!-- JNDI Based Security Management -->
<bean name="JBossSecuritySubjectFactory"
class="org.jboss.security.integration.JBossSecuritySubjectFactory"
/>
```

3. **Append the injection property**
   To inject the callback handler, append a <property> element as a child of the JNDIBasedSecurityManagement <mbean> element. Specify the callback handler using the the <property> and <inject> elements described in Example 7.4, “jboss-service appended callback handler”.

Example 7.6. security-jboss-beans callback handler

```
<bean name="JBossSecuritySubjectFactory"
class="org.jboss.security.integration.JBossSecuritySubjectFactory"
>
   <property name="securityManagement">
      <inject bean="JNDIBasedSecurityManagement" />
   </property>
</bean>
```

4. **Restart server**
   You have now configured the **security-jboss-beans.xml** file to inject your custom callback handler.

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

CHAPTER 8. AUTHORIZATION

Authorization relates to the type of component you want to protect, rather than the layer it resides in.

A security domain does not explicitly require an authorization policy. If an authorization policy is not specified, the default ***jboss-web-policy*** and ***jboss-ejb-policy*** authorization configured in **jboss-as/server/*$PROFILE*/deploy/security/security-policies-jboss-beans.xml** is used.

If you do choose to specify an authorization policy, or create a custom deployment descriptor file with a valid authorization policy, these settings override the default settings in **security-policies-jboss-beans.xml**.

Users can provide authorization policies that implement custom behavior. Configuring custom behavior allows authorization 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).

Overriding the default authorization for EJB or Web components is provided for Java Authorization Contract for Containers (JACC) and Extensible Access Control Markup Language (XACML), apart from the default modules that implement the specification behavior.

Refer to Section 6.2, “<authorization>” for information about the <authorization> element schema.

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

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

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-jboss-beans.xml”.

   Example 8.1. security-policies-jboss-beans.xml

```
<?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.DelegatingAuthoriza
tionModule" 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.DelegatingAuthoriza
tionModule" 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
code="org.jboss.security.authorization.modules.JACCAuthorizationModu
le" 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.JACCAuthorizationModu
le" flag="required"/>
      </authorization>
   </application-policy>

   <application-policy xmlns="urn:jboss:security-beans:1.0"
name="jacc-test" extends="other">
      <authorization>
         <policy-module
code="org.jboss.security.authorization.modules.JACCAuthorizationModu
le" 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 security domains, and override specific settings such as the authorization policy module.

Procedure 8.2. Set authorization policies for specific security domains

You can override authorization for a particular component.

This procedure describes how to inherit settings from other security domain 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 **jboss-as/server/$PROFILE/conf/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.

     $JBOSS_HOME/jboss-as/server/*$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.

     jboss-as/server/*$PROFILE*/deploy/*[prefix]*-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
code="org.jboss.security.authorization.modules.JACCAuthorizationModu
le" 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
code="org.jboss.security.authorization.modules.JACCAuthorizationModu
le" flag="required"/>
      </authorization>
   </application-policy>

   <application-policy xmlns="urn:jboss:security-beans:1.0"
name="test-domain-inherited" extends="other">
      <authorization>
         <policy-module
code="org.jboss.security.authorization.modules.XACMLAuthorizationMod
ule" 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.

8.1. MODULE DELEGATION

Procedure 8.1, “Set authorization policies for all EJB and WAR components” and Procedure 8.2, “Set authorization policies for specific security domains ” describe simplistic examples that show how basic authentication can be configured in security domains.

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

The **`org.jboss.security.authorization.modules.AuthorizationModuleDelegate`** class provides a number of subclasses that allow you to implement module delegation:

- **`AbstractJACCModuleDelegate`**
- **`WebPolicyModuleDelegate`**
- **`EJBPolicyModuleDelegate`**
- **`WebXACMLPolicyModuleDelegate`**
- **`WebJACCPolicyModuleDelegate`**
- **`EJBXACMLPolicyModuleDelegate`**
- **`EJBJACCPolicyModuleDelegate`**

You can create your own authorization delegation module, providing the module extends the **`org.jboss.security.authorization.modules.AuthorizationModuleDelegate`** class.

To implement the delegation module, you declare the delegation modules within the <module-option> element of your <authorization> policy. Each module is prefixed with the component it relates to, as shown in Example 8.2, “Delegation Module Declaration” .

Example 8.2. Delegation Module Declaration

```
<application-policy xmlns="urn:jboss:security-beans:1.0" name="test-
domain" extends="other">
  <authorization>
    <policy-module code="xxx.yyy.MyAuthorizationModule"
flag="required">
      <module-option
name="delegateMap">web=xxx.yyy.mywebauthorizationdelegate,ejb=xxx.yyy.my
ejbauthorizationdelegate</module-option>
```

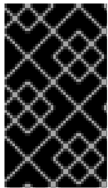
```
    </policy-module>
  </authorization>
</application-policy>
```

CHAPTER 9. MAPPING

In JBoss Enterprise Application Platform 5, 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 ***code*** attribute in the <mapping-module> element. Additionally, the ***type*** attribute must be set to **role**. Refer to Section 6.3, “<mapping>” for information about the <mapping> element schema.

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

In versions prior to JBoss Enterprise Application Platform 5, 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="test-domain">
   <authentication>
   ...
   </authentication>
   <mapping>
      <mapping-module
code="org.jboss.security.mapping.providers.DeploymentRolesMappingProvide
r" 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 **WEB-INF/jboss-web.xml** (**.war** or **.sar**) file.

Example 9.2. <security-role> declaration

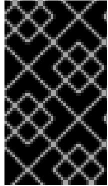
```
<assembly-descriptor>
   ...
   <security-role>
      <role-name>Support</role-name>
      <principal-name>Mark</principal-name>
      <principal-name>Tom</principal-name>
   </security-role>
   ...
</assembly-descriptor>
```

A security role relating to Support principals is implemented in addition to the base security role information contained in **`WEB-INF/jboss-web.xml`**.

CHAPTER 10. AUDITING

Certain government organizations mandate auditing in enterprise applications to ensure the software components of an implementation are traceable, and operating within their design parameters. Additionally, government regulations and standards require audit controls in addition to standard application auditing.

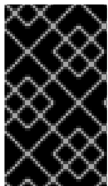
System administrators can enable security event auditing to constantly monitor the operation of the security domain, and deployed Web and EJB applications.



IMPORTANT

Security event auditing may introduce a performance impact on servers that manage high event volumes. Auditing is deactivated by default, and should be configured to be available on-demand.

Activating security event auditing differs between Web and EJB components. Procedure 10.1, “Enable the security audit feature” describes the minimum steps to enable the audit service for EJBs in your implementation . Procedure 10.2, “Enable security auditing for Web containers” describes how to enable security event auditing for Web containers.



IMPORTANT

Web container event auditing can expose sensitive user information. Administrators must ensure appropriate data protection procedures such as password hashing are implemented when configuring security auditing for Web container events.

Procedure 10.1. Enable the security audit feature

1. **Open the log4j configuration file**
   Navigate to **$JBOSS_HOME/server/*$PROFILE*/conf/**

   Open the **jboss-log4j.xml** file using a text editor.

2. **Uncomment the security audit category**
   By default, the Security Audit Provider category definition in the **jboss-log4j.xml** file is commented out. Uncomment the category definition shown in Example 10.1, “log4j Security Audit Provider category”.

Example 10.1. log4j Security Audit Provider category

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

<!-- Limit the verbose MC4J EMS (lib used by admin-console)
categories -->
<category name="org.mc4j.ems">
  <priority value="WARN"/>
</category>

<!-- Show the evolution of the DataSource pool in the logs
[inUse/Available/Max]
<category
name="org.jboss.resource.connectionmanager.JBossManagedConnectionP
ool">
```

```
  <priority value="TRACE"/>
</category>
   -->

<!-- Category specifically for Security Audit Provider -->
<category
name="org.jboss.security.audit.providers.LogAuditProvider"
additivity="false">
<priority value="TRACE"/>
  <appender-ref ref="AUDIT"/>
</category>


<!-- Limit the org.jboss.serial (jboss-serialization) to INFO as
its DEBUG is verbose -->
<category name="org.jboss.serial">
  <priority value="INFO"/>
</category>
```

3. **Uncomment the audit appender**
   By default, the AUDIT appender definition in the **`jboss-log4j.xml`** file is commented out. Uncomment the appender definition shown in Example 10.1, "log4j Security Audit Provider category".

Example 10.2. log4j Security Audit Provider category

```
...
<!--  Emit events as JMX notifications
<appender name="JMX"
class="org.jboss.monitor.services.JMXNotificationAppender">
  <errorHandler
class="org.jboss.logging.util.OnlyOnceErrorHandler"/>
  <param name="Threshold" value="WARN"/>
  <param name="ObjectName"
value="jboss.system:service=Logging,type=JMXNotificationAppender"/
>
  <layout class="org.apache.log4j.PatternLayout">
    <param name="ConversionPattern" value="%d %-5p [%c] %m"/>
  </layout>
</appender>
-->

<!-- Security AUDIT Appender -->
<appender name="AUDIT"
class="org.jboss.logging.appender.DailyRollingFileAppender">
  <errorHandler
class="org.jboss.logging.util.OnlyOnceErrorHandler"/>
  <param name="File" value="${jboss.server.log.dir}/audit.log"/>
  <param name="Append" value="true"/>
  <param name="DatePattern" value="'.'yyyy-MM-dd"/>
  <layout class="org.apache.log4j.PatternLayout">
    <param name="ConversionPattern" value="%d %-5p [%c] (%t:%x)
%m%n"/>
```

```
  </layout>
</appender>

   <!-- ================ -->
   <!-- Limit categories -->
   <!-- ================ -->

<!-- Limit the org.apache category to INFO as its DEBUG is verbose
-->
<category name="org.apache">
  <priority value="INFO"/>
</category>
...
```

4. **Save, and restart server**
   You have now activated the auditing service for your implementation, as configured in the **`jboss-log4j.xml`** file.

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

5. **Verify security auditing is functioning correctly**
   Once the audit service is configured and deployed, audit log entries will verify the audit service and EJB invocation success.

   The **`audit.log`** file is located in **`jboss-as/server/$PROFILE/log/`** directory.

   A successful EJB invocation would look similar to the following **`audit.log`** output.

Example 10.3. Successful EJB Invocation log entry

```
2008-12-05 16:08:26,719 TRACE
[org.jboss.security.audit.providers.LogAuditProvider] (http-
127.0.0.1-8080-2:)
[Success]policyRegistration=org.jboss.security.plugins.JBossPolicy
Registration@76ed4518; Resource:=
[org.jboss.security.authorization.resources.EJBResource:contextMap
=
{policyRegistration=org.jboss.security.plugins.JBossPolicyRegistra
tion@76ed4518}:method=public abstract
org.jboss.test.security.interfaces.RunAsServiceRemote
org.jboss.test.security.interfaces.RunAsServiceRemoteHome.create()
throws
java.rmi.RemoteException,javax.ejb.CreateException:ejbMethodInterf
ace=Home:ejbName=RunAs:ejbPrincipal=jduke:MethodRoles=Roles(identi
tySubstitutionCaller,):securityRoleReferences=null:callerSubject=S
ubject:
        Principal: [roles=[identitySubstitutionCaller,
extraRunAsRole],principal=runAsUser]
        Principal:
Roles(members:extraRunAsRole,identitySubstitutionCaller)
:callerRunAs=[roles=[identitySubstitutionCaller,
extraRunAsRole],principal=runAsUser]:callerRunAs=[roles=
[identitySubstitutionCaller,
```

```
extraRunAsRole],principal=runAsUser]:ejbRestrictionEnforcement=fal
se:ejbVersion=null];Source=org.jboss.security.plugins.javaee.EJBAu
thorizationHelper;Exception:=;
```

An unsuccessful EJB invocation would look similar to the following **audit.log** output.

Example 10.4. Unsuccessful EJB Invocation log entry

```
[Error]policyRegistration=org.jboss.security.plugins.JBossPolicyRe
gistration@76ed4518;Resource:=
[org.jboss.security.authorization.resources.EJBResource:contextMap
=
{policyRegistration=org.jboss.security.plugins.JBossPolicyRegistra
tion@76ed4518}:method=public java.security.Principal
org.jboss.test.security.ejb3.SimpleStatelessSessionBean.invokeUnav
ailableMethod():ejbMethodInterface=Remote:ejbName=SimpleStatelessS
essionBean:ejbPrincipal=UserA:MethodRoles=Roles(<NOBODY>,):securit
yRoleReferences=null:callerSubject=Subject:
        Principal: UserA
        Principal: Roles(members:RegularUser,Administrator)
:callerRunAs=null:callerRunAs=null:ejbRestrictionEnforcement=false
:ejbVersion=null];Source=org.jboss.security.plugins.javaee.EJBAuth
orizationHelper;Exception:=Authorization Failed: ;
```

Procedure 10.2. Enable security auditing for Web containers

1. **Enable EJB security auditing**
   You must enable security as described in Procedure 10.1, “Enable the security audit feature”.

2. **Activate auditing in the server realm**
   Web container auditing must first be activated in the server realm of the **server.xml** file.

   The **server.xml** file is located in the ***jboss-as*/server/*$PROFILE*/deploy/jbossweb.sar/** directory.

   The **<Realm>** element must have the ***enableAudit="true"*** attribute set, as per Example 10.5, “server.xml audit activation”.

   Example 10.5. server.xml audit activation

   ```
   <Realm className="org.jboss.web.tomcat.security.JBossWebRealm"
   certificatePrincipal="org.jboss.security.auth.certs.SubjectDNMappi
   ng" allRolesMode="authOnly"
   enableAudit="true"/>
   ```

3. **Specify auditing levels system property**
   The auditing levels for Web applications must be specified using the org.jboss.security.web.audit system property in the **run.sh** (Linux) or **run.bat** (Microsoft Windows) script.

Alternatively, you can specify the system property in the **jboss-as/server/*$PROFILE*/deploy/properties-service.xml** file.

- **Linux**
  Add the system property into the **jboss-as/bin/run.sh** file.

```
## Specify the Security Audit options
#System Property setting to configure the web audit:
#* off = turn it off
#* headers = audit the headers
#* cookies = audit the cookie
#* parameters = audit the parameters
#* attributes = audit the attributes
#* headers,cookies,parameters = audit the headers,cookie and
#                               parameters
#* headers,cookies = audit the headers and cookies
JAVA_OPTS="$JAVA_OPTS -
Dorg.jboss.security.web.audit=headers,cookies,parameter"
```

- **Microsoft Windows**
  Add the system property into the **jboss-as/bin/run.bat** file.

```
rem Specify the Security Audit options
rem System Property setting to configure the web audit
rem * off = turn it off
rem * headers = audit the headers
rem * cookies = audit the cookie
rem * parameters = audit the parameters
rem * attributes = audit the attributes
rem * headers,cookies,parameters = audit the headers,cookie and
rem   parameters
rem * headers,cookies = audit the headers and cookies
set JAVA_OPTS=%JAVA_OPTS% " -
Dorg.jboss.security.web.audit=headers,cookies,parameter"
```

- **properties-service.xml**
  Update the **SystemPropertiesService** class MBean in the **jboss-as/server/*$PROFILE*/deploy/properties-service.xml** file, and declare the java property as an <attribute>. You can uncomment the relevant operating system block in the code sample below.

```
...

<mbean code="org.jboss.varia.property.SystemPropertiesService"
name="jboss:type=Service,name=SystemProperties">

   <!-- Linux Attribute Declaration -->
   <!--
   <attribute name="Properties">JAVA_OPTS="$JAVA_OPTS -
Dorg.jboss.security.web.audit=headers,cookies,parameter"
   </attribute>
   -->

   <!-- Windows Attribute Declaration -->
```

```
   <!--
   <attribute name="Properties">JAVA_OPTS=%JAVA_OPTS% " -
Dorg.jboss.security.web.audit=headers,cookies,parameter"
   </attribute>
   -->

</mbean>

...
```

4. **Verify security auditing is functioning correctly**
   Once the system property is specified in the files, audit log entries will verify Web invocation success.

   The **`audit.log`** file is located in **`jboss-as/server/$PROFILE/log/`** directory.

   A successful Web invocation would look similar to the following **`audit.log`** output.

   Example 10.6. Successful Web Invocation log entry

   ```
   2008-12-05 16:08:38,997 TRACE
   [org.jboss.security.audit.providers.LogAuditProvider] (http-
   127.0.0.1-8080-17:)
   [Success]policyRegistration=org.jboss.security.plugins.JBossPolicy
   Registration@76ed4518;Resource:=
   [org.jboss.security.authorization.resources.WebResource:contextMap
   =
   {policyRegistration=org.jboss.security.plugins.JBossPolicyRegistra
   tion@76ed4518,securityConstraints=
   [Lorg.apache.catalina.deploy.SecurityConstraint;@6feeae6,
   resourcePermissionCheck=true},canonicalRequestURI=/restricted/get-
   only/x,request=[/web-constraints:cookies=null:headers=user-
   agent=Jakarta Commons-
   HttpClient/3.0,authorization=host=localhost:8080,]
   [parameters=],CodeSource=null];securityConstraints=SecurityConstra
   int[RestrictedAccess - Get
   Only];Source=org.jboss.security.plugins.javaee.WebAuthorizationHel
   per;resourcePermissionCheck=true;Exception:=;
   ```

   An unsuccessful EJB invocation would look similar to the following **`audit.log`** output.

   Example 10.7. Unsuccessful Web Invocation log entry

   ```
   2008-12-05 16:08:41,561 TRACE
   [org.jboss.security.audit.providers.LogAuditProvider] (http-
   127.0.0.1-8080-4:)
   [Failure]principal=anil;Source=org.jboss.web.tomcat.security.JBoss
   WebRealm;request=[/jaspi-web-basic:cookies=null:headers=user-
   agent=Jakarta Commons-
   HttpClient/3.0,authorization=host=localhost:8080,][parameters=]
   [attributes=];2008-12-05 16:07:30,129 TRACE
   [org.jboss.security.audit.providers.LogAuditProvider]
   (WorkerThread#1[127.0.0.1:55055]:)
   ```

CHAPTER 11. DEPLOYING SECURITY DOMAINS

Modular Security Domain

A security domain deployment method, where the security domain declaration is included in a *deployment descriptor*. A modular security domain takes the form **`*-jboss-beans.xml`**. It is included in the **`META-INF`** directory of EJB Jars, or the **`WEB-INF`** directory of web application (WAR).

Deployment Descriptor

A declarative XML configuration file that describes the deployment settings of an application. The way an application is deployed can be changed within this file, eliminating the need to make changes to the underlying code of the application.

There are two ways of deploying a security domain in JBoss Enterprise Application Platform:

1. Declare the security domain in the **`jboss-as/server/$PROFILE/conf/login-config.xml`** file.

2. Create and deploy a modular security domain.

Procedure 11.1. Modular Security Domain configuration

Follow this procedure to configure a basic modular security domain deployment descriptor with two domains for EJB and web applications.

Each domain uses the **`UsersRolesLoginModule`** for the authorization policy, however you are not limited to this login module when creating a modular security domain. Refer to Section 12.1, “Using Modules” for additional login modules shipped with JBoss Enterprise Application Platform.

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

   If you have already created a deployment descriptor for your application, you can skip this step and proceed to step 2.

   The filename takes the format **`[domain_name]-jboss-beans.xml`**. While the *domain_name* is arbitrary, you should choose a name that is meaningful to the application ensure the name of the deployment descriptor is unique across the server profile.

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

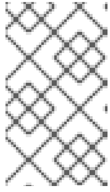
```
<?xml version="1.0" encoding="UTF-8"?>

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


</deployment>
```

2. **Define application policies**
   Individual security domains are defined within the <deployment> element.

In the example below, two security domains are specified. Each authentication 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 12.1, “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 **jboss-as/server/*$PROFILE*/deploy** directory of the required server profile in your installation.

   If you are distributing your application to a wider audience, package the deployment descriptor in the **META-INF** directory of the EJB Jar, or the **WEB-INF** directory of your web application (WAR).

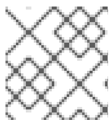
CHAPTER 12. LOGIN MODULES

12.1. USING MODULES

JBoss Enterprise Application Platform includes several bundled login modules suitable for most user management needs. JBoss Enterprise Application Platform can read user information from a relational database, an Lightweight Directory Access Protocol (LDAP) server or flat files. In addition to these core login modules, JBoss provides other login modules that provide user information for very customized needs in JBoss.

12.1.1. LdapLoginModule

`LdapLoginModule` is a **`LoginModule`** implementation that authenticates against a Lightweight Directory Access Protocol (LDAP) server. Use the **`LdapLoginModule`** if your user name and credentials are stored in an LDAP server that is accessible using a Java Naming and Directory Interface (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 protocol level to use. The available values include **none**, **`simple`**, and **`strong`**. If the property is undefined, the behavior is determined by the service provider.

java.naming.security.protocol
: Transport protocol to use for secure access. Set this configuration option to the type of service provider (for example, SSL). If the property is undefined, the behavior is determined by the service provider.

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

java.naming.security.credentials
: Specifies the credentials of the Principal for authenticating the caller to the service. Credentials can take the form of a hashed password, a clear-text password, a key, or a certificate. If the property is undefined, the behavior is determined by the service provider.

The supported login module configuration options include the following:

principalDNPrefix

Prefix added to the user name to form the user *distinguished name*. See **principalDNSuffix** for more info.

principalDNSuffix

Suffix added to the user name when forming the user distinguished name. This is useful if you prompt a user for a user name 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 for searching 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.

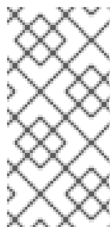
roleAttributeID

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 Distinguished Names (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 user ID. 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 user name is used as the match value against the uidAttributeName attribute. The default value is **false**.

unauthenticatedIdentity

Principal name to assign to requests containing no authentication information. This behavior is inherited from the **UsernamePasswordLoginModule** superclass.

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 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 12.1. LDAP Login Module Authentication Policy

This authentication policy describes how you use the parameters in a security domain authentication policy

```
<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>
```

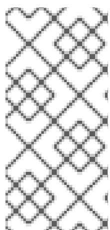
The ***java.naming.factory.initial***, ***java.naming.factory.url*** and ***java.naming.security*** options in the testLDAP <login-module> configuration indicate the following conditions:

- 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 user name of the user and the principalDNSuffix as described above. In Example 12.2, “LDIF File Example”, the user name **jsmith** would map to **uid=jsmith,ou=People,dc=jboss,dc=org**.

Distinguished Name (DN)

In Lightweight Directory Access Protocol (LDAP), the distinguished name uniquely identifies an object in a directory. Each distinguished name must have a unique name and location from all other objects, which is achieved using a number of attribute-value pairs (AVPs). The AVPs define information such as common names, organization unit, among others. The combination of these values results in a unique string required by the LDAP.



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 **jsmith** 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 do not always map well onto LDAP groups, and LDAP administrators are often hesitant to allow external application-specific data in central LDAP servers. 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.

An LDAP Data Interchange Format (LDIF) file representing the structure of the directory this data operates against is shown in Example 12.2, “LDIF File Example”.

LDAP Data Interchange Format (LDIF)

Plain text data interchange format used to represent LDAP directory content and update requests. Directory content is represented as one record for each object or update request. Content consists of add, modify, delete, and rename requests.

Example 12.2. 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=jsmith,ou=People,dc=jboss,dc=org
objectclass: top
objectclass: uidObject
objectclass: person
uid: jsmith
cn: John
sn: Smith
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=jsmith,ou=People,dc=jboss,dc=org
description: the JBossAdmin group
```

12.1.2. Password Stacking

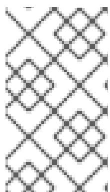
Multiple login modules can be chained together in a stack, with each login module providing both the authentication and authorization components. This works for many use cases, but sometimes authentication and authorization are split across multiple user management stores.

Section 12.1.1, “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 user name 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 12.3. 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>
```

12.1.3. 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 12.4. 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.

12.1.4. 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.

12.1.5. UsersRolesLoginModule

UsersRolesLoginModule is a simple login module that supports multiple users and user roles loaded from Java properties files. The user name-to-password mapping file is called **users.properties** and the user name-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 user name to password mappings. This defaults to **<filename_prefix>-users.properties**.

rolesProperties

Name of the properties resource (file) containing the user name 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 12.5. 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 12.5, “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 12.5, “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,...
```

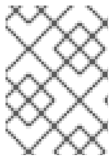
The user name.XXX property name pattern present in **ejb3-sampleapp-roles.properties** is used to assign the user name roles to a particular named group of roles where the **XXX** portion of the property name is the group name. The user name=... form is an abbreviation for user name.Roles=..., where 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** user name:

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

12.1.6. 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 user name, 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.

principalClass

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

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>
```

12.1.7. 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 12.1. 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>
```

Procedure 12.1, "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 12.6, "Certificate Example".

Example 12.6. 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
```

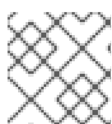
The **localhost.keystore** would need the certificate in Example 12.6, "Certificate Example" stored

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
```

12.1.8. 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>
```

12.1.9. 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.

12.1.10. RunAsIdentity Creation

In order for JBoss Enterprise Application Platform to secure access to EJB methods, the user's identity must be known at the time the method call is made.

A user's identity in the server is represented either by a **javax.security.auth.Subject** instance or an **org.jboss.security.RunAsIdentity** instance. Both these classes store one or more principals that represent the identity and a list of roles that the identity possesses. In the case of the **javax.security.auth.Subject** a list of credentials is also stored.

In the <assembly-descriptor> section of the **ejb-jar.xml** deployment descriptor, you specify one or more roles that a user must have to access the various EJB methods. A comparison of these lists reveals whether the user has one of the roles necessary to access the EJB method.

Example 12.7. org.jboss.security.RunAsIdentity Creation

In the **ejb-jar.xml** file, you specify a <security-identity> element with a <run-as> role defined as a child of the <session> element.

```
<session>
   ...
   <security-identity>
      <run-as>
         <role-name>Admin</role-name>
      </run-as>
   </security-identity>
   ...
</session>
```

This declaration signifies that an "Admin" RunAsIdentity role must be created.

To name a principal for the Admin role, you define a **<run-as-principal>** element in the **jboss-web.xml** file.

```
<session>
   ...
   <security-identity>
      <run-as-principal>John</run-as-principal>
   </security-identity>
```

```
    ...
</session>
```

The <security-identity> element in both the ejb-jar.xml and jboss-web.xml files are parsed at deployment time. The <run-as> role name and the <run-as-principal> name are then stored in the org.jboss.metadata.SecurityIdentityMetaData class.

Example 12.8. Assigning multiple roles to a RunAsIdentity

You can assign more roles to RunAsIdentity by mapping roles to principals in the jboss-web.xml deployment descriptor <assembly-descriptor> element group.

```
<assembly-descriptor>
   ...
   <security-role>
      <role-name>Support</role-name>
      <principal-name>John</principal-name>
      <principal-name>Jill</principal-name>
      <principal-name>Tony</principal-name>
   </security-role>
   ...
</assembly-descriptor>
```

In Example 12.7, “ org.jboss.security.RunAsIdentity Creation” , the <run-as-principal> of "Mark" was created. The configuration in this example extends the "Admin" role, by adding the "Support" role. The new role contains extra principals, including the originally defined principal "John".

The <security-role> element in both the ejb-jar.xml and jboss.xml files are parsed at deployment time. The <role-name> and the <principal-name> data is stored in the org.jboss.metadata.SecurityIdentityMetaData class.

12.1.11. 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 org.jboss.security.SecurityAssociation.principal to the value of the NameCallback filled in by the callbackhandler, and the org.jboss.security.SecurityAssociation.credential to the value of the PasswordCallback filled in by the callbackhandler.

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**.

password-stacking

Activates client-side authentication of clients using other login modules such as the **LdapLoginModule**. When **password-stacking** option is set to **useFirstPass**, the module first looks for a shared user name and password using **javax.security.auth.login.name** and **javax.security.auth.login.password** respectively in the login module shared state map. This allows a module configured prior to this one to establish a valid JBoss user name and password.

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**.

12.2. 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; user name, social security number, employee ID) are stored as **java.security.Principal** objects in the **SubjectPrincipals** 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 as operational environment identity.

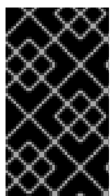
12.2.1. Subject Usage Pattern Support

To simplify correct implementation of the **Subject** usage patterns described in Section 12.2, "Custom Modules", JBossSX includes login modules that populate the authenticated **Subject** with a a template pattern that enforces correct **Subject** usage.

AbstractServerLoginModule

The most generic of the two is the **org.jboss.security.auth.spi.AbstractServerLoginModule** class.

It provides an implementation of the **javax.security.auth.spi.LoginModule** interface and offers abstract methods for the key tasks specific to an operation environment security infrastructure. The key details of the class are highlighted in Example 12.9, "AbstractServerLoginModule Class Fragment". The JavaDoc comments detail the responsibilities of subclasses.



IMPORTANT

The **loginOk** instance variable is pivotal. This must be set to **true** if the login succeeds, or **false** by any subclasses that override the login method. If this variable is incorrectly set, the commit method will not correctly update the subject.

Tracking the login phase outcomes allows login modules to be chained together with control flags. These control flags do not require the login modules to succeed as part of the authentication process.

Example 12.9. AbstractServerLoginModule Class Fragment

```
package org.jboss.security.auth.spi;
/**
 *  This class implements the common functionality required for a JAAS
 *  server-side LoginModule and implements the JBossSX standard
 *  Subject usage pattern of storing identities and roles. Subclass
```

```
 *  this module to create your own custom LoginModule and override the
 *  login(), getRoleSets(), and getIdentity() methods.
 */
public abstract class AbstractServerLoginModule
    implements javax.security.auth.spi.LoginModule
{
    protected Subject subject;
    protected CallbackHandler callbackHandler;
    protected Map sharedState;
    protected Map options;
    protected Logger log;

    /** Flag indicating if the shared credential should be used */
    protected boolean useFirstPass;
    /**
     * Flag indicating if the login phase succeeded. Subclasses that
     * override the login method must set this to true on successful
     * completion of login
     */
    protected boolean loginOk;

    // ...
    /**
     * Initialize the login module. This stores the subject,
     * callbackHandler and sharedState and options for the login
     * session. Subclasses should override if they need to process
     * their own options. A call to super.initialize(...)  must be
     * made in the case of an override.
     *
     * <p>
     * The options are checked for the  <em>password-stacking</em>
parameter.
     * If this is set to "useFirstPass", the login identity will be
taken from the
     * <code>javax.security.auth.login.name</code> value of the
sharedState map,
     * and the proof of identity from the
     * <code>javax.security.auth.login.password</code> value of the
sharedState map.
     *
     * @param subject the Subject to update after a successful login.
     * @param callbackHandler the CallbackHandler that will be used to
obtain the
     * the user identity and credentials.
     * @param sharedState a Map shared between all configured login
module instances
     * @param options the parameters passed to the login module.
     */
    public void initialize(Subject subject,
                           CallbackHandler callbackHandler,
                           Map sharedState,
                           Map options)
    {
        // ...
    }
```

```
    /**
     *  Looks for javax.security.auth.login.name and
     *  javax.security.auth.login.password values in the sharedState
     *  map if the useFirstPass option was true and returns true if
     *  they exist. If they do not or are null this method returns
     *  false.
     *  Note that subclasses that override the login method
     *  must set the loginOk var to true if the login succeeds in
     *  order for the commit phase to populate the Subject. This
     *  implementation sets loginOk to true if the login() method
     *  returns true, otherwise, it sets loginOk to false.
     */
    public boolean login()
        throws LoginException
    {
        // ...
    }

    /**
     *  Overridden by subclasses to return the Principal that
     *  corresponds to the user primary identity.
     */
    abstract protected Principal getIdentity();

    /**
     *  Overridden by subclasses to return the Groups that correspond
     *  to the role sets assigned to the user. Subclasses should
     *  create at least a Group named "Roles" that contains the roles
     *  assigned to the user.  A second common group is
     *  "CallerPrincipal," which provides the application identity of
     *  the user rather than the security domain identity.
     *
     *  @return Group[] containing the sets of roles
     */
    abstract protected Group[] getRoleSets() throws LoginException;
}
```

UsernamePasswordLoginModule

The second abstract base login module suitable for custom login modules is the **`org.jboss.security.auth.spi.UsernamePasswordLoginModule`**.

This login module further simplifies custom login module implementation by enforcing a string-based user name as the user identity and a **`char[]`** password as the authentication credentials. It also supports the mapping of anonymous users (indicated by a null user name and password) to a principal with no roles. The key details of the class are highlighted in the following class fragment. The JavaDoc comments detail the responsibilities of subclasses.

Example 12.10. UsernamePasswordLoginModule Class Fragment

```
package org.jboss.security.auth.spi;

/**
 *  An abstract subclass of AbstractServerLoginModule that imposes a
```

```
 *  an identity == String username, credentials == String password
 *  view on the login process. Subclasses override the
 *  getUsersPassword() and getUsersRoles() methods to return the
 *  expected password and roles for the user.
 */
public abstract class UsernamePasswordLoginModule
    extends AbstractServerLoginModule
{
    /** The login identity */
    private Principal identity;
    /** The proof of login identity */
    private char[] credential;
    /** The principal to use when a null username and password are seen
*/
    private Principal unauthenticatedIdentity;

    /**
     * The message digest algorithm used to hash passwords. If null then
     * plain passwords will be used. */
    private String hashAlgorithm = null;

    /**
     *  The name of the charset/encoding to use when converting the
     * password String to a byte array. Default is the platform's
     * default encoding.
     */
     private String hashCharset = null;

    /** The string encoding format to use. Defaults to base64. */
    private String hashEncoding = null;

    // ...

    /**
     *  Override the superclass method to look for an
     *  unauthenticatedIdentity property. This method first invokes
     *  the super version.
     *
     *  @param options,
     *  @option unauthenticatedIdentity: the name of the principal to
     *  assign and authenticate when a null username and password are
     *  seen.
     */
    public void initialize(Subject subject,
                           CallbackHandler callbackHandler,
                           Map sharedState,
                           Map options)
    {
        super.initialize(subject, callbackHandler, sharedState,
                         options);
        // Check for unauthenticatedIdentity option.
        Object option = options.get("unauthenticatedIdentity");
        String name = (String) option;
        if (name != null) {
            unauthenticatedIdentity = new SimplePrincipal(name);
        }
```

```
    }

    // ...

    /**
     *  A hook that allows subclasses to change the validation of the
     *  input password against the expected password. This version
     *  checks that neither inputPassword or expectedPassword are null
     *  and that inputPassword.equals(expectedPassword) is true;
     *
     *  @return true if the inputPassword is valid, false otherwise.
     */
    protected boolean validatePassword(String inputPassword,
                                       String expectedPassword)
    {
        if (inputPassword == null || expectedPassword == null) {
            return false;
        }
        return inputPassword.equals(expectedPassword);
    }

    /**
     *  Get the expected password for the current username available
     * via the getUsername() method. This is called from within the
     * login() method after the CallbackHandler has returned the
     * username and candidate password.
     *
     * @return the valid password String
     */
    abstract protected String getUsersPassword()
        throws LoginException;
}
```

Subclassing Login Modules

The choice of sub-classing the **`AbstractServerLoginModule`** versus **`UsernamePasswordLoginModule`** is simply based on whether a string-based user name and credentials are usable for the authentication technology you are writing the login module for. If the string-based semantic is valid, then subclass **`UsernamePasswordLoginModule`**, otherwise subclass **`AbstractServerLoginModule`**.

Subclassing Steps

The steps your custom login module must execute depend on which base login module class you choose. When writing a custom login module that integrates with your security infrastructure, you should start by sub-classing **`AbstractServerLoginModule`** or **`UsernamePasswordLoginModule`** to ensure that your login module provides the authenticated **`Principal`** information in the form expected by the JBossSX security manager.

When sub-classing the **`AbstractServerLoginModule`**, you must override the following:

- **`void initialize(Subject, CallbackHandler, Map, Map)`**: if you have custom options to parse.

- **boolean login()**: to perform the authentication activity. Be sure to set the **loginOk** instance variable to true if login succeeds, false if it fails.
- **Principal getIdentity()**: to return the **Principal** object for the user authenticated by the **log()** step.
- **Group[] getRoleSets()**: to return at least one **Group** named **Roles** that contains the roles assigned to the **Principal** authenticated during **login()**. A second common **Group** is named **CallerPrincipal** and provides the user's application identity rather than the security domain identity.

When sub-classing the **UsernamePasswordLoginModule**, you must override the following:

- **void initialize(Subject, CallbackHandler, Map, Map)**: if you have custom options to parse.
- **Group[] getRoleSets()**: to return at least one **Group** named **Roles** that contains the roles assigned to the **Principal** authenticated during **login()**. A second common **Group** is named **CallerPrincipal** and provides the user's application identity rather than the security domain identity.
- **String getUsersPassword()**: to return the expected password for the current user name available via the **getUsername()** method. The **getUsersPassword()** method is called from within **login()** after the **callbackhandler** returns the user name and candidate password.

12.2.2. 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 12.11, “ 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 12.11. 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 user name 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 user name=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

CHAPTER 13. 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, they 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 13.1, “The JBossSX components of the SRP client-server framework.” describes the key components involved in the JBossSX implementation of the SRP client/server framework.

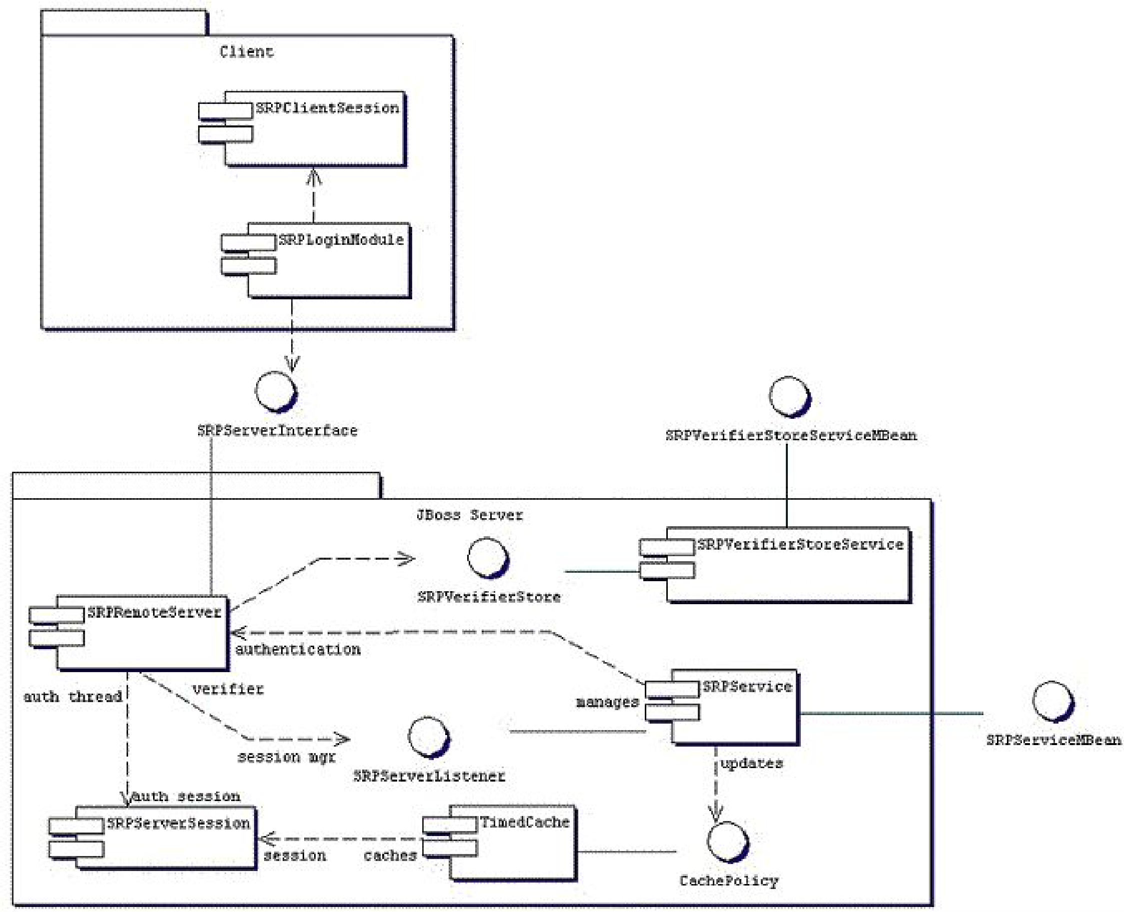


Figure 13.1. The JBossSX components of the SRP client-server framework.

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 13.1, “Login Configuration Entry” .

Example 13.1. 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`** .

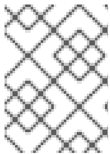
The **`SRPVerifierStoreService`** MBean also supports ***`addUser`*** and ***`delUser`*** operations for addition and deletion of users. The signatures are:

```
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 13.2, “The SRPVerifierStore interface” .

13.1. 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 user name 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 user name.

3. The client side **`SRPLoginModule`** begins an SRP session by creating an **`SRPClientSession`** object using the login user name, 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 user name 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.

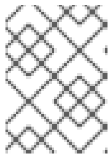
5. The client side **SRPLoginModule** saves the login user name and **M1** challenge into the **LoginModule** sharedState map. This is used as the Principal name and credentials by the standard JBoss **ClientLoginModule**. The **M1** challenge is used in place of the password as proof of identity on 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 user name. 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 user name 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 13.2, “SRPCacheLoginModule with SRP Session Cache” illustrates the operation of the SRPCacheLoginModule.login method implementation.

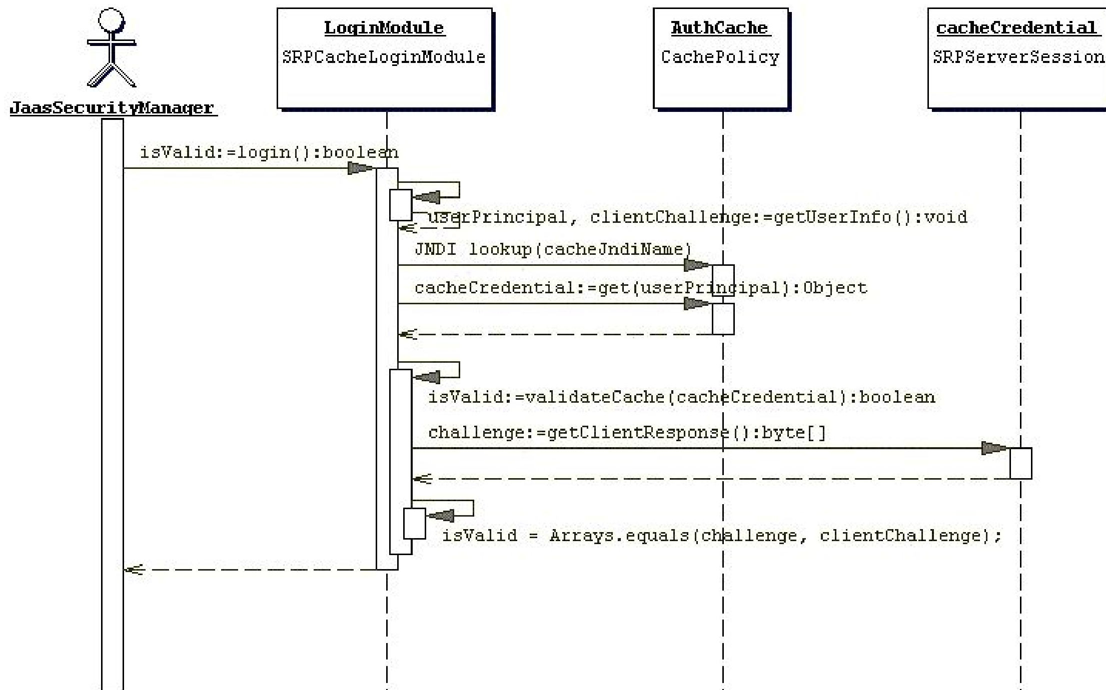
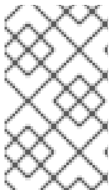


Figure 13.2. SRPCacheLoginModule with SRP Session Cache

13.2. 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 13.2, "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 13.2. 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 user name. 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:

user name

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 user name 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 13.1. 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 13.3, “Secure Remote Password Example”).

13.3. 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 13.3, “The security-ex3.sar jboss-service.xml Descriptor” .

The example client side and server side login module configurations are described in Example 13.4, “The client side standard JAAS configuration” and Example 13.5, “The server side XMLLoginConfig configuration” give .

Example 13.3. 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 13.4. 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 13.5. 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 13.3, “Secure Remote Password Example” you must set the cache timeout correctly, or handle re-authentication on failure.

CHAPTER 14. JAVA SECURITY MANAGER

Java Security Manager

The Java Security Manager is a class that manages the external boundary of the Java Virtual Machine (JVM) sandbox, controlling how code executing within the JVM can interact with resources outside the JVM. When the Java Security Manager is activated the Java API checks with the security manager for approval before executing a wide range of potentially unsafe operations.

The Security Manager uses a security policy to determine whether a given action will be permitted or denied.

Security Policy

A set of defined permissions for different classes of code. The Java Security Manager compares actions requested by applications against the security policy. If an action is allowed by the policy, the Security Manager will permit that action to take place. If the action is not allowed by the policy, the Security Manager will deny that action. The security policy can define permissions based on the location of code or on the code's signature.

The Security Manager and the security policy used are configured using the Java Virtual Machine options `java.security.manager` and `java.security.policy`.

Security Manager-related options

java.security.manager

Use a security manager, optionally specifying which security manager to use. If no argument is supplied with this option the default JDK security manager, `java.lang.SecurityManager`, is used. To use another security manager implementation, supply the fully qualified classname of a subclass of `java.lang.SecurityManager` with this option.

java.security.policy

Specifies a policy file to augment or replace the default security policy for the VM. This option takes two forms:

`java.security.policy=policyFileURL`

The policy file referenced by *policyFileURL* will *augment* the default security policy configured by the VM.

`java.security.policy==policyFileURL`

The policy file referenced by *policyFileURL* will *replace* the default security policy configured by the VM.

The *policyFileURL* value can be a URL or a file path.

JBoss Enterprise Application Platform does not activate the Java Security Manager by default. To configure the Platform to use the Security Manager, refer to Section 14.1, “Using the Security Manager”.

14.1. USING THE SECURITY MANAGER

JBoss Enterprise Application Platform can use the JDK default Security Manager or a custom security manager. For details on selecting a custom Security Manager, refer to Security Manager-related options .

When the Platform is configured to use a security manager, a security policy file must be specified. A security policy file, **jboss-as/bin/server.policy.cert** is included as a starting point.

For information on writing security policy, refer to Section 14.3, “Writing Security Policy for JBoss Enterprise Application Platform” .

Configuration File

The file **run.conf** (Linux) or **run.conf.bat** (Windows) is used to configure the Security Manager and security policy. This file is found in the **jboss-as/bin** directory.

This file is used to configure server-level options, and applies to all server profiles. Configuring the Security Manager and security policy involves profile-specific configuration. You may elect to copy the global **run.conf** or **run.conf.bat** file from **jboss-as/bin/** to the server profile (for example: **jboss-as/server/production/run.conf**), and make the configuration changes there. A configuration file in the server profile takes precedence over the global **run.conf** / **run.conf.bat** file when the server profile is started.

Procedure 14.1. Activate the Security Manager

This procedure configures JBoss Enterprise Application Platform to start with the Java Security Manager activated.

The file editing actions in this procedure refer to the file **run.conf** (Linux), or **run.conf.bat** (Windows) in the server profile directory, if one exists there, or in **jboss-as/bin** . Refer to Configuration File for details on the location of this file.

1. **Specify the JBoss home directory**
   Edit the file **run.conf** (Linux), or **run.conf.bat** (Windows). Add the **jboss.home.dir** option, specifying the path to the **jboss-as** directory of your installation.

   Linux

   ```
   JAVA_OPTS="$JAVA_OPTS -Djboss.home.dir=/path/to/jboss-eap-5.1/jboss-as"
   ```

   Windows

   ```
   JAVA_OPTS="%JAVA_OPTS% -Djboss.home.dir=c:\path\jboss-eap-5.1\jboss-as"
   ```

2. **Specify the server home directory**
   Add the **jboss.server.home.dir** option, specifying the path to your server profile.

   Linux

   ```
   JAVA_OPTS="$JAVA_OPTS -Djboss.server.home.dir=path/to/jboss-eap-5.1/jboss-as/server/production"
   ```

   Windows

```
JAVA_OPTS="%JAVA_OPTS% -Djboss.server.home.dir=c:\path\to\jboss-eap-
5.1\jboss-as\server\production"
```

3. **Specify the Protocol Handler**
   Add the **java.protocol.handler.pkgs** option, specifying the JBoss stub handler.

   Linux

   ```
   JAVA_OPTS="$JAVA_OPTS -
   Djava.protocol.handler.pkgs=org.jboss.handlers.stub"
   ```

   Windows

   ```
   JAVA_OPTS="%JAVA_OPTS% -
   Djava.protocol.handler.pkgs=org.jboss.handlers.stub"
   ```

4. **Specify the security policy to use**
   Add the **$POLICY** variable, specifying the security policy to use. Add the variable definition before the line that activates the Security Manager.

   Example 14.1. Use the Platform's included security policy

   ```
   POLICY="server.policy.cert"
   ```

5. **Activate the Security Manager**
   Uncomment the following line by removing the initial # :

   Linux

   ```
   #JAVA_OPTS="$JAVA_OPTS -Djava.security.manager -
   Djava.security.policy=$POLICY"
   ```

   Windows

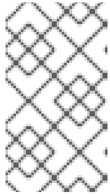
   ```
   #JAVA_OPTS="%JAVA_OPTS% -Djava.security.manager -
   Djava.security.policy=%POLICY%"
   ```

   Result:

   JBoss Enterprise Application Platform is now configured to start with the Security Manager activated.

6. **Optional: Import Red Hat's JBoss signing key**
   The included security policy grants permissions to JBoss-signed code. If you use the included policy you must import the JBoss signing key to the JDK **cacerts** key store.

   The following command assumes that the environment variable **JAVA_HOME** is set to the location of a JDK supported by JBoss Enterprise Application Platform 5. You configure **JAVA_HOME** when you first install JBoss Enterprise Application Platform 5. Refer to the *Installation Guide* for further information.



NOTE

To ensure the correct JVM is selected, you can use the **alternatives** command to select from JDKs installed on your Linux system. Refer to Appendix A, *Setting the default JDK with the* ***/usr/sbin/alternatives*** *Utility*.

Execute the following command in a terminal:

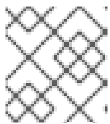
Linux

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

Windows

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

Although broken across two lines in this documentation, the commands above should be entered on one single line in a terminal.



NOTE

The default password for the cacerts key store is **changeit** .

Result:

The key used to the sign the JBoss Enterprise Application Platform code is now installed.

14.2. DEBUGGING SECURITY POLICY ISSUES

You can enable debugging information to help you troubleshoot security policy-related issues. The **java.security.debug** option configures the level of security-related information reported.

The command **java -Djava.security.debug=help** will produce help output with the full range of debugging options. Setting the debug level to **all** is useful when troubleshooting a security-related failure whose cause is completely unknown, but for general use it will produce too much information. A sensible general default is **access:failure**.

Procedure 14.2. Enable general debugging

This procedure will enable a sensible general level of security-related debug information.

- Add the following line to the file **run.conf** (Linux), or **run.conf.bat** (Windows):

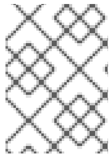
  Linux

  ```
  JAVA_OPTS="$JAVA_OPTS -Djava.security.debug=access:failure"
  ```

  Windows

```
JAVA_OPTS="%JAVA_OPTS% -Djava.security.debug=access:failure"
```

14.2.1. Debugging Security Manager



NOTE

The Debugging Security Manager was introduced with JBoss Enterprise Application Platform 5.1

The Debugging Security Manager **org.jboss.system.security.DebuggingJavaSecurityManager** prints out the protection domain corresponding to a failing permission. This additional information is very useful information when debugging permissions problems.

Procedure 14.3. Enable the Debugging Security Manager

This procedure will enable the Debugging Security Manager.

1. Add the following option to **$JBOSS_HOME/bin/run.conf** (Linux) or **$JBOSS_HOME/bin/run.conf.bat**. See Configuration File for the location of this file.

   Linux

   ```
   JAVA_OPTS="$JAVA_OPTS -
   Djava.security.manager=org.jboss.system.security.DebuggingJavaSecuri
   tyManager"
   ```

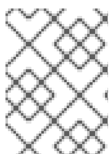
   Windows

   ```
   JAVA_OPTS="%JAVA_OPTS% -
   Djava.security.manager=org.jboss.system.security.DebuggingJavaSecuri
   tyManager"
   ```

2. Comment out all other **java.security.manager** references in the file.

3. Ensure that the file still contains a **java.security.policy** option specifying the policy file to use

4. Enable general debugging following the instruction in Procedure 14.2, “Enable general debugging”.



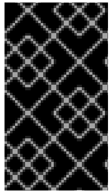
NOTE

The Debugging Security Manager has a significance performance cost. Do not use it in general production.

14.3. WRITING SECURITY POLICY FOR JBOSS ENTERPRISE APPLICATION PLATFORM

The included file **jboss-as/bin/server.policy.cert** is an example security policy for JBoss Enterprise Application Platform. You can use this file as the basis for your own security policy.

The **policytool** application, included with the JDK, provides a graphical tool for editing and writing security policy.



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.

For a general treatment of security policy files and Java permissions refer to the official Java documentation at http://download-llnw.oracle.com/javase/6/docs/technotes/guides/security/PolicyFiles.html . JBoss-specific **java.lang.RuntimePermissions** are described below.

JBoss-specific Runtime Permissions

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 multi-thread 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 does not 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.

CHAPTER 15. SECURING THE EJB RMI TRANSPORT LAYER

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.

Transport options for EJB3 via SSL

This chapter describes configuration of two different arrangements for Remote Method Invocation of EJB3s over an encrypted transport: RMI + SSL and RMI via HTTPS. HTTPS is an option as the transport for RMI when firewall configuration prevents use of the RMI ports.

Generating a key pair for SSL is covered in Section 15.2, “Generate encryption keys and certificate” .

Configuring RMI + SSL for EJB3 is covered in Section 15.3, “EJB3 RMI + SSL Configuration” .

Configuring RMI via HTTPS for EJB3 is covered in Section 15.4, “EJB3 RMI via HTTPS Configuration” .

Configuring RMI + SLL for EJB2 is covered in Section 15.5, “EJB2 RMI + SSL Configuration” .

15.1. SSL ENCRYPTION OVERVIEW

15.1.1. 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 are 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 connects 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 15.2.1, “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.

15.2. GENERATE ENCRYPTION KEYS AND CERTIFICATE

15.2.1. Generate a self-signed certificate with keytool

15.2.1.1. Generate a key pair

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 Create a secure remoting connector for RMI .

Procedure 15.1. Generate a new key pair and add it to the key store "localhost.keystore" in the JBoss server conf directory.

This procedure generates a new key pair for SSL encryption.

- The following command will create a key pair for use with SSL encryption:

```
keytool -genkey -alias ejb-ssl -keystore localhost.keystore -
storepass KEYSTORE_PASSWORD
  -keypass EJB-SSL_KEYPAIR_PASSWORD
  -dname "CN=SERVER_NAME,OU=QE,O=example.com,L=Brno,C=CZ"
```

Result:

A key pair will be added to the key store `localhost.keystore` under the alias `ejb-ssl`.

The parameters for this command are explained in `keytool` parameters

`keytool` parameters

alias

An alphanumeric token used to identify the key pair within the key store. A key store can contain multiple keys. The alias provides a means to uniquely identify a key pair within a key store. The alias for a key pair must by unique within a key store.

keystore

The key store that will be used to store the key pair. This can be a relative or absolute file path.

storepass

The password for key store. If the key store already exists, this must be the existing password for the key store. If the key store specified does not already exist, it will be created and this password will be the new password. This password is needed to access the key store to retrieve or store keys and certificates.

keypass

The password for the new key pair. This password must be supplied to use the key pair in the future.

dname

The identifying details of the certificate.

CN

Common Name: the name of the server. This must match the server name as returned to clients in a JNDI lookup. If a client attempts to make an SSL connection to the server using one name from JNDI, and receives a certificate with a different name, the connection will fail.

OU

Organizational Unit: the name of the organizational unit that is responsible for the server.

O

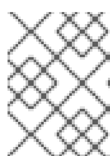
Organization: The name of the organization, sometimes expressed as a URL.

L

Location: the location of the server.

C

Country: two letter country code



NOTE

For best security practice, store key store files on a secure file system, readable only by the owner of the JBoss Application Server process.

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

15.2.1.2. Export a self-signed certificate

Once a key pair has been generated for the server to use, a certificate must be created. Procedure 15.2, "Export a certificate" details the steps to export the **ejb-ssl** key from the key store named **localhost.keystore**.

Procedure 15.2. Export a certificate

This procedure exports a certificate from a key store into a file.

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 is exported to the file **mycert.cer**.

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

To make remote method invocations over SSL, a client needs to trust the certificate of the server. The certificate we generated is self-signed and does not have a chain of trust to a known certificate authority. With a self-signed certificate the client must be explicitly configured to trust the certificate; otherwise the connection fails. To configure a client to trust a self-signed certificate, import 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 are accepted as valid. If your server uses a self-signed certificate then any client that makes remote method calls over SSL requires that certificate in its trust store. Export your public key as a certificate, and then import that certificate to the trust store on those clients.

The certificate created in Section 15.2.1.2, "Export a self-signed certificate" must be copied to the client in order to perform the steps detailed in Procedure 15.3, "Import the certificate to the trust store "localhost.truststore"".

Procedure 15.3. Import the certificate to the trust store "localhost.truststore"

This procedure imports a certificate that was previously exported on a server to the trust store on a client.

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 a new trust store.

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

   Result:

   The certificate is imported to the trust store, and a secure connection can now be established with a server that uses this certificate.

As with the key store, if the trust store specified does not already exist, it is created. However, in contrast with the key store, there is no default trust store and the command fails if one is not 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. Do this 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 15.1, “Invoking the com.acme.Runclient application with a specific trust store” is an example command that invokes the application **com.acme.RunClient**, a hypothetical application that makes remote method calls to an EJB on a JBoss Application Server. This command is run from the root of the application's package directory (the directory containing **com** directory in the file path `com/acme/RunClient.class`).

Example 15.1. Invoking the com.acme.Runclient application with a specific trust store

```
java -cp $JBOSS_HOME/client/jbossall-client.jar:. -
Djavax.net.ssl.trustStore=${resources}/localhost.truststore \
    -Djavax.net.ssl.trustStorePassword=TRUSTSTORE_PASSWORD
com.acme.RunClient
```

15.3. EJB3 RMI + SSL CONFIGURATION

Procedure 15.4. Configure RMI + SSL for EJB3 Overview

This procedure configures SSL encryption of Remote Method Invocation traffic between EJB3 beans on the server and a fat client running on another machine on the network.

1. Generate encryption keys and certificate
2. Configure a secure remote connector for RMI
3. Annotate EJB3 beans to use the secure RMI connector

Generating encryption keys and certificates is covered in Section 15.2, “Generate encryption keys and certificate”.

Create a secure remoting connector for RMI

The file `ejb3-connectors-jboss-beans.xml` in a JBoss Application Server profile `deploy` directory contains JBoss Remoting connector definitions for EJB3 remote method invocation.

Example 15.2. Sample Secure EJB3 Connector

The beans described in the code sample are appended to the `ejb3-connectors-jboss-beans.xml` file. Both beans are required to configure a secure connector for EJB3 using the key pair created in Procedure 15.1, “ Generate a new key pair and add it to the key store

"localhost.keystore" in the JBoss server conf directory. ".

The **keyPassword** property in the sample configuration is the key pair password specified when the key pair was created.

The sample configuration creates a connector that listens for SSL connections on port 3843. This port needs to be open on the server firewall for access by clients.

```
<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>
```

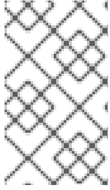
Configure EJB3 Beans for SSL Transport

All EJB3 beans use the unsecured RMI connector by default. To enable remote invocation of a bean via SSL, annotate the bean with **@org.jboss.annotation.ejb.RemoteBinding**.

Example 15.3. EJB3 bean annotation to enable secure remote invocation

The annotation binds 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, communicates with the server via SSL.

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



NOTE

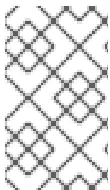
In Example 15.3, “EJB3 bean annotation to enable secure remote invocation” the IP address is specified as 0.0.0.0, meaning "all interfaces". Change this to the value of the ${jboss.bind.address} system property.

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 15.4, “EJB3 Bean annotation for secure and unsecured invocation” demonstrates the annotations to do this.

Example 15.4. EJB3 Bean annotation for secure and unsecured invocation

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



NOTE

In Example 15.4, “EJB3 Bean annotation for secure and unsecured invocation” , the IP address is specified as `0.0.0.0`, meaning "all interfaces". Change this to the value of the ${jboss.bind.address} system property.

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

15.4. EJB3 RMI VIA HTTPS CONFIGURATION

Procedure 15.5. Configure EJB3 RMI via HTTPS Overview

This procedure configures tunneling of Remote Method Invocation traffic over SSL-encrypted HTTP. This has the dual effect of encrypting the traffic and allowing it to traverse firewalls that block the RMI port.

1. Generate encryption keys and certificates.
2. Configure RMI via HTTPS web connector.
3. Configure Servlets.

4. Configure secure remoting connector for RMI via HTTPS.

5. Configure EJB3 beans for HTTPS transport.

6. Configure clients for RMI via HTTPS.

Generating encryption keys and certificates is covered in Section 15.2, “Generate encryption keys and certificate” .

Procedure 15.6. Configure RMI via HTTPS web connector

This procedure creates a web connector that listens on port 8443 and accepts SSL connections from clients.

- Edit the file **jboss-as/server/*$PROFILE*/deploy/jbossweb.sar/server.xml** and uncomment the HTTPS connector.

```
<!-- SSL/TLS Connector configuration using the admin devl guide
keystore -->
<Connector protocol="HTTP/1.1" SSLEnabled="true"
   port="8443" address="${jboss.bind.address}"
   scheme="https" secure="true" clientAuth="false"
   keystoreFile="${jboss.server.home.dir}/conf/localhost.keystore"
   keystorePass="KEYSTORE_PASSWORD" sslProtocol = "TLS" />
```

Result:

You create a web connector to accept SSL connections.

Procedure 15.7. Configure Servlets

This procedure configures a servlet that passes requests from the web connector to the **ServletServerInvoker** .

1. Create a directory named **servlet-invoker.war** in **jboss-as/server/*$PROFILE*/deploy/**.

2. Create a **WEB-INF** directory in the **servlet-invoker.war** directory.

3. Create a file named **web.xml** in that **WEB-INF** directory, with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC
   "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
   "http://java.sun.com/dtd/web-app_2_3.dtd">


<web-app>
    <servlet>
        <servlet-name>ServerInvokerServlet</servlet-name>
        <description>The ServerInvokerServlet receives requests via
HTTP
           protocol from within a web container and passes it onto
the
           ServletServerInvoker for processing.
        </description>
```

```
        <servlet-
class>org.jboss.remoting.transport.servlet.web.ServerInvokerServlet<
/servlet-class>

        <init-param>
            <param-name>locatorUrl</param-name>
            <param-
value>servlet://${jboss.bind.address}:8080/servlet-
invoker/ServerInvokerServlet</param-value>
            <description>The servlet server invoker</description>
        </init-param>

        <load-on-startup>1</load-on-startup>
    </servlet>


    <servlet>
        <servlet-name>SSLServerInvokerServlet</servlet-name>
        <description>The ServerInvokerServlet receives requests via
HTTPS
           protocol from within a web container and passes it onto
the
           ServletServerInvoker for processing.
        </description>
        <servlet-
class>org.jboss.remoting.transport.servlet.web.ServerInvokerServlet<
/servlet-class>

        <init-param>
            <param-name>locatorUrl</param-name>
            <param-
value>sslservlet://${jboss.bind.address}:8443/servlet-
invoker/SSLServerInvokerServlet</param-value>
            <description>The servlet server invoker</description>
        </init-param>

        <load-on-startup>2</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>ServerInvokerServlet</servlet-name>
        <url-pattern>/ServerInvokerServlet/*</url-pattern>
    </servlet-mapping>

    <servlet-mapping>
        <servlet-name>SSLServerInvokerServlet</servlet-name>
        <url-pattern>/SSLServerInvokerServlet/*</url-pattern>
    </servlet-mapping>

</web-app>
```

Result:

You create a servlet to forward SSL requests from the web container to a server invoker.

The `locatorUrl` is used to connect the servlet to the remoting connector through the " `InvokerLocator` attribute of the remoting connector we define in Procedure 15.8, “Configure secure

remoting connector for RMI via HTTPS”.

Procedure 15.8. Configure secure remoting connector for RMI via HTTPS

This procedure creates the Server Invoker that implements RMI.

- Create a file named **servlet-invoker-service.xml** in **jboss-as/server/*$PROFILE*/deploy/**, with the following content:

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

<server>
   <mbean code="org.jboss.remoting.transport.Connector"
name="jboss.remoting:service=connector,transport=servlet"
    display-name="Servlet transport Connector">
      <attribute
name="InvokerLocator">servlet://${jboss.bind.address}:8080/servlet-
invoker/ServerInvokerServlet</attribute>
      <attribute name="Configuration">
         <handlers>
            <handler
subsystem="AOP">org.jboss.aspects.remoting.AOPRemotingInvocationHand
ler</handler>
         </handlers>
      </attribute>
   </mbean>

   <mbean code="org.jboss.remoting.transport.Connector"
name="jboss.remoting:service=connector,transport=sslservlet"
    display-name="Servlet transport Connector">
      <attribute
name="InvokerLocator">sslservlet://${jboss.bind.address}:8443/servle
t-invoker/SSLServerInvokerServlet</attribute>
      <attribute name="Configuration">
         <handlers>
            <handler
subsystem="AOP">org.jboss.aspects.remoting.AOPRemotingInvocationHand
ler</handler>
         </handlers>
      </attribute>
   </mbean>
</server>
```

Result:

You create a remoting connector that can receive requests from a servlet, and invoke methods of an EJB3.

Procedure 15.9. Configure EJB3 beans for HTTPS transport

This procedure configures the EJB3 to bind to the HTTPS transport.

- Annotate the bean for RMI via HTTPS:

Example 15.5. Annotating an EJB3 for RMI via HTTPS

```
// RMI tunneled over HTTPS
@Stateless
@RemoteBinding(clientBindUrl = "https://0.0.0.0:8443/servlet-
invoker/SSLServerInvokerServlet")
@Remote(Calculator.class)
@SecurityDomain("other")
public class CalculatorHttpsBean implements Calculator
{
....
```

Result:

The EJB3 is now available for remote invocation via HTTPS.

Annotating a bean for RMI via HTTP

Optionally, you can annotate the bean for invocation via RMI via HTTP. This can be useful for testing, as it allows you to tunnel RMI calls through firewalls that block RMI ports, but removes the extra layer of the security configuration.

Example 15.6. Annotating a bean for RMI via HTTP

```
// RMI tunneled over HTTP
@Stateless
@RemoteBinding(clientBindUrl = "http://0.0.0.0:8080/servlet-
invoker/ServerInvokerServlet")
@Remote(Calculator.class)
@SecurityDomain("other")
public class CalculatorHttpBean extends CalculatorImpl
{
....
```

Configure clients for RMI via HTTPS

The EJB client should use the following properties for the JNDI lookup when looking up beans:

Client access to RMI via HTTP(S)

HTTPS

```
Properties props = new Properties();
props.put("java.naming.factory.initial",
"org.jboss.naming.HttpNamingContextFactory");
props.put("java.naming.provider.url",
"https://localhost:8443/invoker/JNDIFactory");
props.put("java.naming.factory.url.pkgs", "org.jboss.naming");
Context ctx = new InitialContext(props);
props.put(Context.SECURITY_PRINCIPAL, username);
props.put(Context.SECURITY_CREDENTIALS, password);
Calculator calculator = (Calculator) ctx.lookup(jndiName);
// use the bean to do any operations
```

HTTP

```
Properties props = new Properties();
props.put("java.naming.factory.initial",
"org.jboss.naming.HttpNamingContextFactory");
props.put("java.naming.provider.url",
"http://localhost:8080/invoker/JNDIFactory");
props.put("java.naming.factory.url.pkgs", "org.jboss.naming");
Context ctx = new InitialContext(props);
props.put(Context.SECURITY_PRINCIPAL, username);
props.put(Context.SECURITY_CREDENTIALS, password);
Calculator calculator = (Calculator) ctx.lookup(jndiName);
// use the bean to do any operations
```

In Client access to RMI via HTTP(S) , the *user name* and *password* values correspond to a valid user name and password for the security domain that is used to secure the http-invoker. This security domain is set in **jboss-as/*$PROFILE*/deploy/http-invoker.sar/invoker.war/WEB-INF/jboss-web.xml** .

15.5. EJB2 RMI + SSL CONFIGURATION

Procedure 15.10. Configure SSL for EJB2 Overview

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

Generating encryption keys and certificates is covered in Section 15.2, “Generate encryption keys and certificate” .

Configured Unified Invoker for SSL

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 **$JBOSS_HOME/server/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 15.7. 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>
```

Now customize the SSLSocketBuilder, by adding the following to the **$JBOSS_HOME/server/*$PROFILE*/conf/jboss-service.xml** file of a JBoss Application Server profile:

Example 15.8. SSLSocketBuilder configuration

```
<!-- This section is for custom (SSL) server socket factory  -->
   <mbean code="org.jboss.remoting.security.SSLSocketBuilder"
      name="jboss.remoting:service=SocketBuilder,type=SSL"
      display-name="SSL Server Socket Factory Builder">
      <!-- IMPORTANT - If making ANY customizations, this MUST be set
to false. -->
      <!-- Otherwise, will used default settings and the following
attributes will be ignored. -->
      <attribute name="UseSSLServerSocketFactory">false</attribute>
      <!-- This is the url string to the key store to use -->
      <attribute name="KeyStoreURL">localhost.keystore</attribute>
      <!-- The password for the key store -->
      <attribute name="KeyStorePassword">sslsocket</attribute>
      <!-- The password for the keys (will use KeystorePassword if this
is not set explicitly. -->
      <attribute name="KeyPassword">sslsocket</attribute>
      <!-- The protocol for the SSLContext.  Default is TLS. -->
      <attribute name="SecureSocketProtocol">TLS</attribute>
      <!-- The algorithm for the key manager factory.  Default is
SunX509. -->
      <attribute name="KeyManagementAlgorithm">SunX509</attribute>
      <!-- The type to be used for the key store. -->
      <!-- Defaults to JKS.  Some acceptable values are JKS (Java
Keystore - Sun's keystore format), -->
      <!-- JCEKS (Java Cryptography Extension keystore - More secure
version of JKS), and -->
      <!-- PKCS12 (Public-Key Cryptography Standards #12
                 keystore - RSA's Personal Information Exchange Syntax
Standard). -->
      <!-- These are not case sensitive. -->
      <attribute name="KeyStoreType">JKS</attribute>
   </mbean>

  <mbean
code="org.jboss.remoting.security.SSLServerSocketFactoryService"
     name="jboss.remoting:service=ServerSocketFactory,type=SSL"
     display-name="SSL Server Socket Factory">
     <depends optional-attribute-name="SSLSocketBuilder"
        proxy-
type="attribute">jboss.remoting:service=SocketBuilder,type=SSL</depends>
  </mbean>
```

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 15.9. SSL Transport for Beans

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

<annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX(name="jboss.re
moting:service=Connector,transport=socket",
exposedInterface=org.jboss.remoting.transport.ConnectorMBean.class,regis
terDirectly=true)
  </annotation>
  <property name="serverConfiguration"><inject
bean="UnifiedInvokerConfiguration"/></property>
  <property name="serverSocketFactory"><inject
bean="sslServerSocketFactoryEJB2"/></property>
  <!-- add this to configure the SSL socket for the UnifiedInvoker -->
   </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>
   ...
```

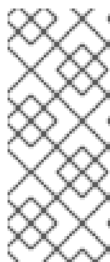
CHAPTER 16. MASKING PASSWORDS IN XML CONFIGURATION

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

16.1. PASSWORD MASKING OVERVIEW

Passwords are secret authentication tokens that are used to limit access to resources to authorized parties only. 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 through inclusion in configuration files.

All JBoss Enterprise Application Platform 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 17, *Encrypting Data Source Passwords*for instructions on encrypting Data Source passwords; to Chapter 18, *Encrypting the Keystore Password in a Tomcat Connector*for instructions on encrypting the key store password in Tomcat; and to Chapter 19, *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 unauthorized 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 16.1. 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.

16.2. GENERATE A KEY STORE AND A MASKED PASSWORD

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 16.6, “Changing the password masking defaults” for instructions.

Procedure 16.2. 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 15.1, “SSL Encryption overview”.

16.3. 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 16.3. 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 Procedure 16.2, "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.

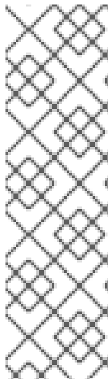
Note:

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.

16.4. 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 16.4. Create password masks

Prerequisites:

- Procedure 16.2, "Generate a key pair and key store for password masking" .
- Procedure 16.3, "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 '`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 Procedure 16.2, "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 16.6, "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 16.6, "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`'

16.5. 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 Procedure 16.4, “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 16.1. General form of password mask annotation

```
<annotation>@org.jboss.security.integration.password.Password(securityDo
main=MASK_NAME, methodName=setPROPERTY_NAME)</annotation>
```

As a concrete example, the JBoss Messaging password is stored in the server profile in the file **`deploy/messaging/messaging-jboss-beans.xml`**. If you create a password mask named "messaging", then the before and after snippet of the configuration file looks like this:

Example 16.2. JBoss Messaging Microcontainer Bean Configuration Before

```
<property name="suckerPassword">CHANGE ME!!</property>
```

Example 16.3. JBoss Messaging Microcontainer Bean Configuration After

```
<annotation>@org.jboss.security.integration.password.Password(securityDo
main=messaging,
methodName=setSuckerPassword)</annotation>
```

16.6. CHANGING THE PASSWORD MASKING DEFAULTS

JBoss Enterprise Application Platform 5 ships with server profiles configured 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 16.4. 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</proper
ty>
         <property
name="keyStorePasswordEncryptedFileName">password/jboss_keystore_pass.da
t</property>
   </bean>
```

CHAPTER 17. ENCRYPTING DATA SOURCE PASSWORDS

Database connections for the JBoss Enterprise Application Platform 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 data source files with encrypted passwords.

This chapter presents two different methods for encrypting data source passwords.

Secured Identity using the module **SecureIdentityLoginModule** is described in Section 17.1, “Secured Identity”.

*Configured Identity with Password Based Encryption*using the module **JaasSecurityDomainIdentityLoginModule** is described in Section 17.1, “Secured Identity”.

17.1. 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 17.1. Overview: Using SecureIdentityLoginModule to encrypt a data source 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.

17.1.1. 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 17.2. Encrypt a data source password - Platform versions 5.0 and 5.0.1

This procedure encrypts a data source password on JBoss Enterprise Application Platform versions 5.0 and 5.0.1

1. Change directory to the **jboss-as** directory
2. Invoke the SecureIdentityLoginModule with the following command, supplying the clear text password as *PASSWORD*:

   Linux command

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

   Windows command:

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

-

Result:

The command will return an encrypted password.

Procedure 17.3. Encrypt a data source password - Platform version 5.1 and later

This procedure encrypts a data source password on 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.

17.1.2. 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 an application authentication policy for your encrypted password, add a new <application-policy> element to the <policy> element.

Example 17.1, “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 17.1. 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-option
name="managedConnectionFactoryName">jboss.jca:name=PostgresDS,service=Lo
calTxCM</module-option>
            </login-module>
```

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

SecureIdentityLoginModule module options

user name

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

password

Provide the encrypted password generated in Section 17.1.1, “Encrypt the data source password” .

managedConnectionFactoryName

jboss.jca:name

Nominate a Java Naming and Directory Interface (JNDI) name for this data source.

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

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

At run-time the application policy is bound to JNDI under the application policy name, and is made available as a security domain.

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 17.1.2, “Create an application authentication policy with the encrypted password”.

Using the example name from Section 17.1.2, “Create an application authentication policy with the encrypted password”, "EncryptDBPassword", will result in a data source file that looks something like Example 17.2, “Example data source file using secured identity” .

Example 17.2. 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>
```

17.2. CONFIGURED IDENTITY WITH PASSWORD BASED ENCRYPTION (PBE)

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

Procedure 17.4. Encrypt password with PBEUtils - Platforms version 5.0 and 5.0.1

This procedure encrypts a password on JBoss Enterprise Application Platform versions 5.0 and 5.0.1.

- Execute the command:

  ```
  java -cp jboss-as/common/lib/jbosssx.jar
  org.jboss.security.plugins.PBEUtils \
    salt count domain-password data-source-password
  ```

  Result:

  The encrypted password is displayed

Procedure 17.5. Encrypt password with PBEUtils - Platform version 5.1

This procedure encrypts a password on JBoss Enterprise Application Platform versions 5.1 and later.

- Execute the command:

```
java -cp jboss-as/lib/jbosssx.jar
org.jboss.security.plugins.PBEUtils \
 salt count domain-password data-source-password
```

Result:

The encrypted password is displayed

The parameters for the **PBEUtils** are:

salt

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

count

The IterationCount attribute from the JaasSecurity domain.

domain-password

The plaintext password that maps to the KeyStorePass attribute from the JaasSecurityDomain.

data-source-password

The plaintext password for the data source that should be encrypted with the JaasSecurityDomain password.

Example 17.3, “PBEUtils command example” provides an example of the command with its output.

Example 17.3. PBEUtils command example

```
 java -cp jbosssx.jar org.jboss.security.plugins.PBEUtils abcdefgh 13
master password
 Encoded password: 3zbEkBDfpQAASa3H39pIyP
```

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="jaasSecurityDomain">

jboss.security:service=JaasSecurityDomain,domain=ServerMasterPassword
         </module-option>
      </login-module>
   </authentication>
</application-policy>
```

-

The **$JBOSS_HOME/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>
   ...


   <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="jaasSecurityDomain">

jboss.security:service=JaasSecurityDomain,domain=ServerMasterPassword
            </module-option>
         </login-module>
      </authentication>
   </application-policy>

   <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
name="KeyStorePass">{CLASS}org.jboss.security.plugins.FilePassword:${jboss
.server.home.dir}/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>
```

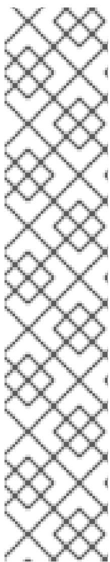
```
    </mbean>

    ...

</datasources>
```

WARNING

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



NOTE

When starting a service that depends on an encrypted data source, the error **`java.security.InvalidAlgorithmParameterException: Parameters missing`** is raised when the following MBean is not yet started as a service:

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

Include the following element 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=Serve
rMasterPassword</depends>
```

CHAPTER 18. ENCRYPTING THE KEYSTORE PASSWORD IN A TOMCAT CONNECTOR

JBoss Web is based on Apache Tomcat.

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** file.

A working understanding of the JaasSecurityDomain that supports keystores, truststores, and password based encryption is advised.

Refer to Chapter 13, *Secure Remote Password Protocol* and Chapter 17, *Encrypting Data Source Passwords* for supporting information and related procedures.

Procedure 18.1. Encrypt Tomcat Container Keystore Password

1. **Append connector element**
   Add a connector element in **server.xml** in
   $JBOSS_HOME/server/*$PROFILE*/deploy/jbossweb.sar

   ```
   <!-- SSL/TLS Connector with encrypted keystore password
   configuration  -->
   <Connector port="8443" address="${jboss.bind.address}"
      maxThreads="100" minSpareThreads="5" maxSpareThreads="15"
      scheme="https" secure="true" clientAuth="true"
      sslProtocol="TLS"
      securityDomain="java:/jaas/encrypt-keystore-password"
      SSLImplementation="org.jboss.net.ssl.JBossImplementation" >
   </Connector>
   ```

   .

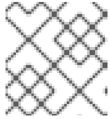
2. **Configure JaasSecurityDomain MBean**
   Set the JaasSecurityDomain MBean in a
   $JBOSS_HOME/server/*$PROFILE*/deploy/security-service.xml file.

   If the file does not exist, you must create it. The code sample describes the content required when the file does not exist. If you already have a **security-service.xml**, append the <mbean> element block to the file.

   ```
   <server>
      <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
   name="KeyStorePass">{CLASS}org.jboss.security.plugins.FilePassword:$
   {jboss.server.home.dir}/conf/keystore.password</attribute>
         <attribute name="Salt">welcometojboss</attribute>
   ```

```
        <attribute name="IterationCount">13</attribute>
    </mbean>
</server>
```

The Salt and IterationCount are the variables that define the strength of your encrypted password, so you can vary it from what is shown. Ensure you record the new values, and use when generating the encrypted password.



NOTE

The Salt must be at least eight characters long.

3. **Generate encrypted password**
   The <mbean> configuration specifies that the keystore is stored in the **jboss-as/server/*$PROFILE*/conf/localhost.keystore** file. The <mbean> also specifies the encrypted password file is stored in **jboss-as/server/*$PROFILE*/conf/keystore.password** file.

   You must create the **localhost.keystore** file.

   Execute the following command in the **jboss-as/server/*$PROFILE*/conf** directory.

   ```
   [conf]$ java -cp $JBOSS_HOME/lib/jbosssx.jar
   \org.jboss.security.plugins.FilePassword welcometojboss 13 unit-
   tests-server keystore.password
   ```

   This command uses jbosssx.jar as the classpath (**-cp**) and the FilePassword security plugin to create a **keystore.password** file with the password set as **unit-tests-server**. To verify you have permission to create a **keystore.password** file, you supply the salt and iteration parameters configured in the <mbean> <attribute> elements of the JaasSecurityDomain.

   You execute this command in the **/conf** directory so the **keystore.password** file is saved to this directory.

4. **Update the Tomcat service MBean**
   Navigate to **$JBOSS_HOME/server/*$PROFILE*/deploy/jbossweb.sar/META-INF**.

   Open **jboss-service.xml** and append the following <depends> tag toward the end of the file. Adding the <depends> tag specifies that Tomcat must start after **jboss.security:service=PBESecurityDomain**.

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

Example 18.1. JaasSecurityDomain definition for pkcs12 keystores

Based on Procedure 18.1, “Encrypt Tomcat Container Keystore Password”, pkcs12 keystore containers referenced by the Tomcat Connector would look similar to this 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="KeyStoreType">pkcs12</attribute>
    <attribute
name="KeyStoreURL">resource:localhost.keystore</attribute>
    <attribute
name="KeyStorePass">{CLASS}org.jboss.security.plugins.FilePassword:${jbo
ss.server.home.dir}/conf/keystore.password</attribute>
    <attribute name="Salt">welcometojboss</attribute>
    <attribute name="IterationCount">13</attribute>
</mbean>
```

18.1. 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 18.2. Predefined JaasSecurityDomain

1. **Update jboss-service.xml to add a connector**
   Navigate to **$JBOSS_HOME/server/ *$PROFILE* /deploy/jbossweb.sar/META-INF**, and add the following code block to the **jboss-service.xml** file.

   ```
   <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** .

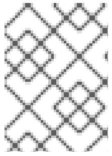
   Open **server.xml** and append the following <depends> element toward the end of the file:

   ```
   <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>
```



NOTE

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

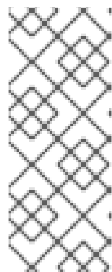
CHAPTER 19. USING LDAPEXTLOGINMODULE WITH JAASSECURITYDOMAIN

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.

Procedure 19.1.

1. **Define JaasSecurityDomain MBean**
   Define the JaasSecurityDomain MBean used to decrypt the encrypted version of the password. You can add the MBean to **$JBOSS_HOME/server/$PROFILE/conf/jboss-service.xml**, or to a ***-service.xml** deployment descriptor in the **$JBOSS_HOME/server/$PROFILE/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>
```



NOTE

The default cipher algorithm used by the JaasSecurityDomain implementation is **PBEwithMD5andDES**. Other cipher algorithms include **DES**, **TripleDES**, **Blowfish**, and **PBEWithMD5AndTripleDES**. All algorithms are symmetric algorithms. You specify a cipher algorithm by appending an <attribute> element with the ***CypherElement*** attribute set to one of these values.

2. **Adjust password, salt, and iteration count**
   Step One contains a simple configuration where the required password, Salt, and Iteration Count used for the encryption or decryption are contained within the MBean definition.

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

3.

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
name="jaasSecurityDomain">jboss.security:service=JaasSecurityDomain,domain
=jmx-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.

CHAPTER 20. FIREWALLS

JBoss Enterprise Application Platform ships with many socket-based services that require open firewall ports. Table 20.1, "The ports found in the default configuration" lists services that listen on ports that must be activated when accessing JBoss behind a firewall. Table 20.2, "Additional ports in the all configuration" lists additional ports that exist in the all profile.

Table 20.1. 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`
4446	TCP	`org.jboss.invocation.unified.server.UnifiedInvoker`
4457	TCP	JBoss Messaging 1.x socket
4712	TCP	JBossTS Recovery Manager socket
4713	TCP	JBossTS Transaction Status Manager
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 20.2. 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`

Port	Type	Service
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`
4448	TCP	`org.jboss.invocation.pooled.server.PooledInvokerHA`
4448	TCP	`org.jboss.invocation.pooled.server.PooledInvokerHA`
7900	TCP	
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.

CHAPTER 21. CONSOLES AND INVOKERS

JBoss Enterprise Application Platform 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.

21.1. JMX CONSOLE

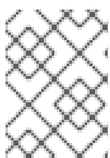
The **jmx-console.war** found in the **deploy** directory provides an HTML view into the JMX Microkernel. As such, it provides access to administrative actions like shutting down the server, stopping services, deploying new services, etc. It should either be secured like any other web application, or removed.

21.2. 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*

21.3. 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.

21.4. 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.

21.5. 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 21.1, “The main components in the detached invoker architecture”.

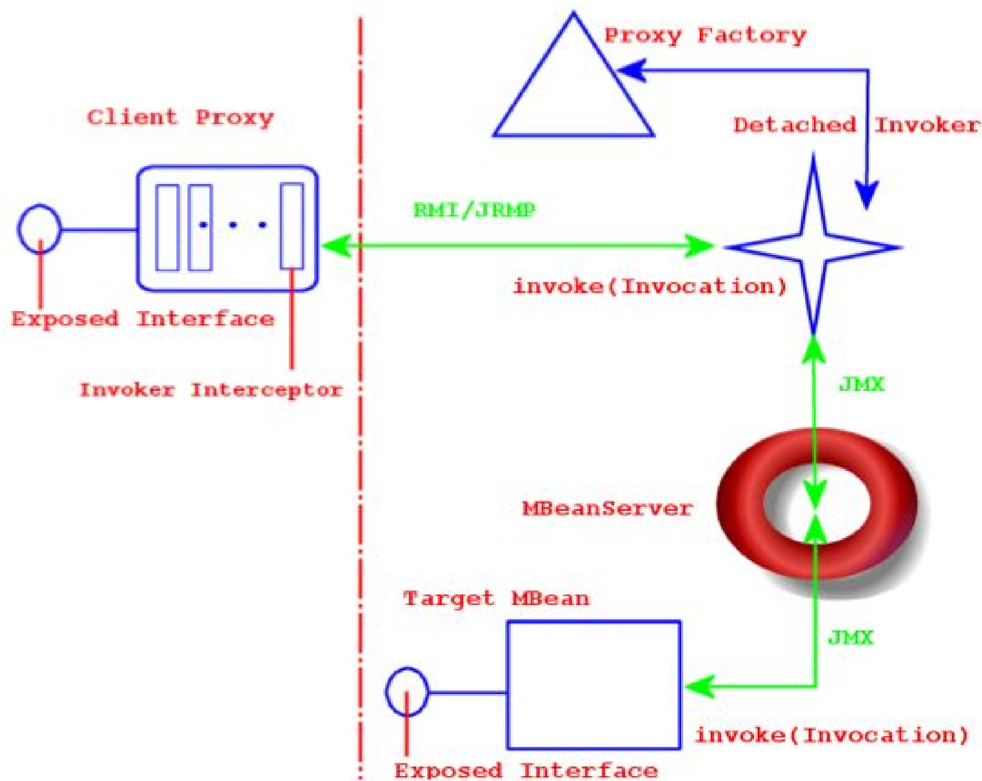


Figure 21.1. The main components in the detached invoker architecture

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 21.1, "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 21.1, "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 acts as a transport gateway that accepts invocations represented as the **org.jboss.invocation.Invocation** object over its specific transport. The invoker service unmarshalls the invocation, forwards the invocation onto the destination MBean service represented by the *Target MBean* in Figure 21.1, "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 21.1, "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 21.1, "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.

21.5.1. 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 21.1. 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 operates.

Example 21.2. 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 21.3. 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 **ObjectNames** 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 21.4. 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 21.5. 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 21.3, "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.

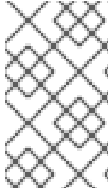
Example 21.6. Block Five

```
...
                // 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 21.7, "Default jmx-invoker-adaptor-server.sar deployment descriptor" shows the **`jboss-service.xml`** descriptor for this deployment.

Example 21.7. 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=proxyFac
tory">
        <!-- 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>
```

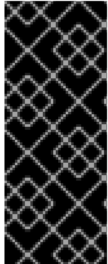
```
        </attribute>
        <depends>jboss:service=invoker,type=jrmp</depends>
    </mbean>
    <!-- This is the service that handles the RMIAdaptor invocations by
routing
         them to the MBeanServer the service is deployed under. -->
    <mbean code="org.jboss.jmx.connector.invoker.InvokerAdaptorService"
           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 21.7, "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.

APPENDIX A. SETTING THE DEFAULT JDK WITH THE /USR/SBIN/ALTERNATIVES UTILITY

/usr/sbin/alternatives is a tool for managing different software packages that provide the same functionality. **Red Hat Enterprise Linux** uses **/usr/sbin/alternatives** to ensure that only one Java Development Kit is set as the system default at one time.



IMPORTANT

Installing a Java Development Kit from the Red Hat Network will normally result in an automatically configured system. However, if multiple JDKs are installed, it is possible that **/usr/sbin/alternatives** may contain conflicting configurations. Refer to Procedure A.1, “ Using /usr/sbin/alternatives to Set the Default JDK ” for syntax of the **/usr/sbin/alternatives** command.

Procedure A.1. Using /usr/sbin/alternatives to Set the Default JDK

1. **Become the root user.**
   /usr/sbin/alternatives needs to be run with root privileges. Use the **su** command or other mechanism to gain these privileges.

2. **Set java.**
   Input this command: **/usr/sbin/alternatives --config java**

   Next, follow the on-screen directions to ensure that the correct version of **java** is selected. Table A.1, “java alternative commands” shows the relevant command settings for each of the different JDKs.

   Table A.1. java alternative commands

JDK	alternative command
OpenJDK 1.6	/usr/lib/jvm/jre-1.6.0-openjdk/bin/java
Sun Microsystems JDK 1.6	/usr/lib/jvm/jre-1.6.0-sun/bin/java

3. **Set javac.**
   Enter this command: **/usr/sbin/alternatives --config javac**

   Follow the on-screen directions to ensure that the correct version of **javac** is selected. Table A.2, “javac alternative commands” shows the appropriate command settings for the different JDKs.

   Table A.2. javac alternative commands

JDK	alternative command
OpenJDK 1.6	/usr/lib/jvm/java-1.6.0-openjdk/bin/javac
Sun Microsystems JDK 1.6	/usr/lib/jvm/java-1.6.0-sun/bin/javac

4. **Extra Step: Set `java_sdk_1.6.0`.**
   The **Sun Microsystems JDK 1.6** requires an additional command be run:

   ```
   /usr/sbin/alternatives --config java_sdk_1.6.0
   ```

   Follow the on-screen directions to ensure that the correct **`java_sdk`** is selected. It is **`/usr/lib/jvm/java-1.6.0-sun`**.

APPENDIX B. REVISION HISTORY

Revision 5.1.0-113.400	**2013-10-31**	**Rüdiger Landmann**

Rebuild with publican 4.0.0

Revision 5.1.0-113	**2012-07-18**	**Anthony Towns**

Rebuild for Publican 3.0

Revision 5.1.0-111	**Wed Sep 15 2010**	**Jared Morgan**

Contains defects and enhancements relating to the Common Criteria Certification for JBoss Enterprise Application Platform v5.1, and other related issues raised by customers and the community.

Rewrote chapters on using SSL to secure Remote Method Invocation of EJBs, and Masking Passwords.

JBOSSCC-47 - Issues raised during Common Criteria feedback rounds.

JBOSSCC-53 - Added Section 1.7, “Enabling Form-based Authentication”.

JBOSSCC-54 - Added Tomcat security mechanism information to Figure 4.2, “Secured EJB Home Method Authentication and Authorization Invocation Steps.”.

JBOSSCC-55 - Clarified the location of Administration Console security information.

JBOSSCC-56 - Duplicate of JBOSSCC-55.

JBOSSCC-57 - Clarified book containing security configuration for legacy invokers.

JBOSSCC-58 - Added Section 21.5, “Remote Access to Services, Detached Invokers”

JBOSSCC-59 - Updated firewall ports that must be enabled depending on which profile is used (see Chapter 20, *Firewalls*.

JBOSSCC-62 - Updated Chapter 3, *JBoss Security Model*

JBOSSCC-63 - Added Chapter 16, *Masking Passwords in XML Configuration*.

JBOSSCC-65 - Added EJB3 Firewall Port information to Chapter 20, *Firewalls*

JBPAPP-3298 - Updated grant statement in Chapter 14, *Java Security Manager* to specify the JBoss

JBPAPP-4942 - Added Chapter 17, *Encrypting Data Source Passwords*.

JBPAPP-4973 - Final review comments from Common Criteria QE.