



Red Hat AMQ 6.3

Security Guide

Making Red Hat AMQ secure

Red Hat AMQ 6.3 Security Guide

Making Red Hat AMQ secure

JBoss A-MQ Docs Team

Content Services

fuse-docs-support@redhat.com

Legal Notice

Copyright © 2016 Red Hat.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

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, the Red Hat 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 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

This guide describes how to configure the Red Hat AMQ subsystem.

Table of Contents

CHAPTER 1. SECURITY ARCHITECTURE	4
1.1. OSGI CONTAINER SECURITY	4
1.2. APACHE ACTIVEMQ SECURITY	5
CHAPTER 2. SECURING THE CONTAINER	7
2.1. JAAS AUTHENTICATION	7
2.2. ROLE-BASED ACCESS CONTROL	33
2.3. USING ENCRYPTED PROPERTY PLACEHOLDERS	45
2.4. ENABLING REMOTE JMX SSL	49
CHAPTER 3. SECURING THE JETTY HTTP SERVER	54
JETTY SERVER	54
CREATE X.509 CERTIFICATE AND PRIVATE KEY	54
ENABLING SSL/TLS FOR JETTY IN A STANDALONE CONTAINER	54
CUSTOMIZING ALLOWED TLS PROTOCOLS AND CIPHER SUITES	55
CONNECT TO THE SECURE CONSOLE	55
ADVANCED JETTY SECURITY CONFIGURATION	56
ENABLING SSL/TLS FOR JETTY IN A FABRIC	60
REFERENCES	63
CHAPTER 4. SECURING THE MANAGEMENT CONSOLE	64
4.1. CONTROLLING ACCESS TO THE FUSE MANAGEMENT CONSOLE	64
CHAPTER 5. SECURING AN APACHE ACTIVEMQ BROKER	65
5.1. PROGRAMMING CLIENT CREDENTIALS	65
5.2. CONFIGURING CREDENTIALS FOR BROKER COMPONENTS	65
5.3. BROKER-TO-BROKER AUTHENTICATION	67
5.4. TUTORIAL I: JAAS AUTHENTICATION	67
5.5. TUTORIAL II: SSL/TLS SECURITY	69
5.6. SECURITY OPTIONS FOR JMS OBJECTMESSAGE SERIALIZATION	75
CHAPTER 6. SECURING THE CAMEL ACTIVEMQ COMPONENT	79
6.1. SECURE ACTIVEMQ CONNECTION FACTORY	79
6.2. EXAMPLE CAMEL ACTIVEMQ COMPONENT CONFIGURATION	80
CHAPTER 7. SSL/TLS SECURITY	82
7.1. INTRODUCTION TO SSL/TLS	82
7.2. SECURE TRANSPORT PROTOCOLS	83
7.3. JAVA KEYSTORES	84
7.4. HOW TO USE X.509 CERTIFICATES	86
7.5. CONFIGURING JSSE SYSTEM PROPERTIES	89
7.6. SETTING SECURITY CONTEXT FOR THE OPENWIRE/SSL PROTOCOL	92
7.7. SECURING JAVA CLIENTS	93
CHAPTER 8. AUTHORIZATION	95
8.1. SIMPLE AUTHORIZATION PLUG-IN	95
8.2. CACHED LDAP AUTHORIZATION PLUG-IN	98
8.3. LDAP AUTHORIZATION PLUG-IN	101
8.4. PROGRAMMING MESSAGE-LEVEL AUTHORIZATION	106
CHAPTER 9. LDAP AUTHENTICATION TUTORIAL	108
9.1. TUTORIAL OVERVIEW	108
9.2. SET-UP A DIRECTORY SERVER AND CONSOLE	108

9.3. ADD USER ENTRIES TO THE DIRECTORY SERVER	111
9.4. ENABLE LDAP AUTHENTICATION IN THE OSGI CONTAINER	115
9.5. ADD BROKER AUTHORIZATION ENTRIES	122
9.6. ENABLE LDAP AUTHORIZATION IN THE BROKER	127
CHAPTER 10. SECURING THE APACHE ACTIVEMQ STANDARD DISTRIBUTION	133
10.1. APACHE ACTIVEMQ STANDARD DISTRIBUTION	133
10.2. CONFIGURE AND RUN ACTIVE-MQ USING ENCRYPTED PASSWORDS	133
APPENDIX A. MANAGING CERTIFICATES	136
A.1. WHAT IS AN X.509 CERTIFICATE?	136
A.2. CERTIFICATION AUTHORITIES	137
A.3. CERTIFICATE CHAINING	138
A.4. SPECIAL REQUIREMENTS ON HTTPS CERTIFICATES	139
A.5. CREATING YOUR OWN CERTIFICATES	141
APPENDIX B. ASN.1 AND DISTINGUISHED NAMES	148
B.1. ASN.1	148
B.2. DISTINGUISHED NAMES	148
INDEX	151

CHAPTER 1. SECURITY ARCHITECTURE

Abstract

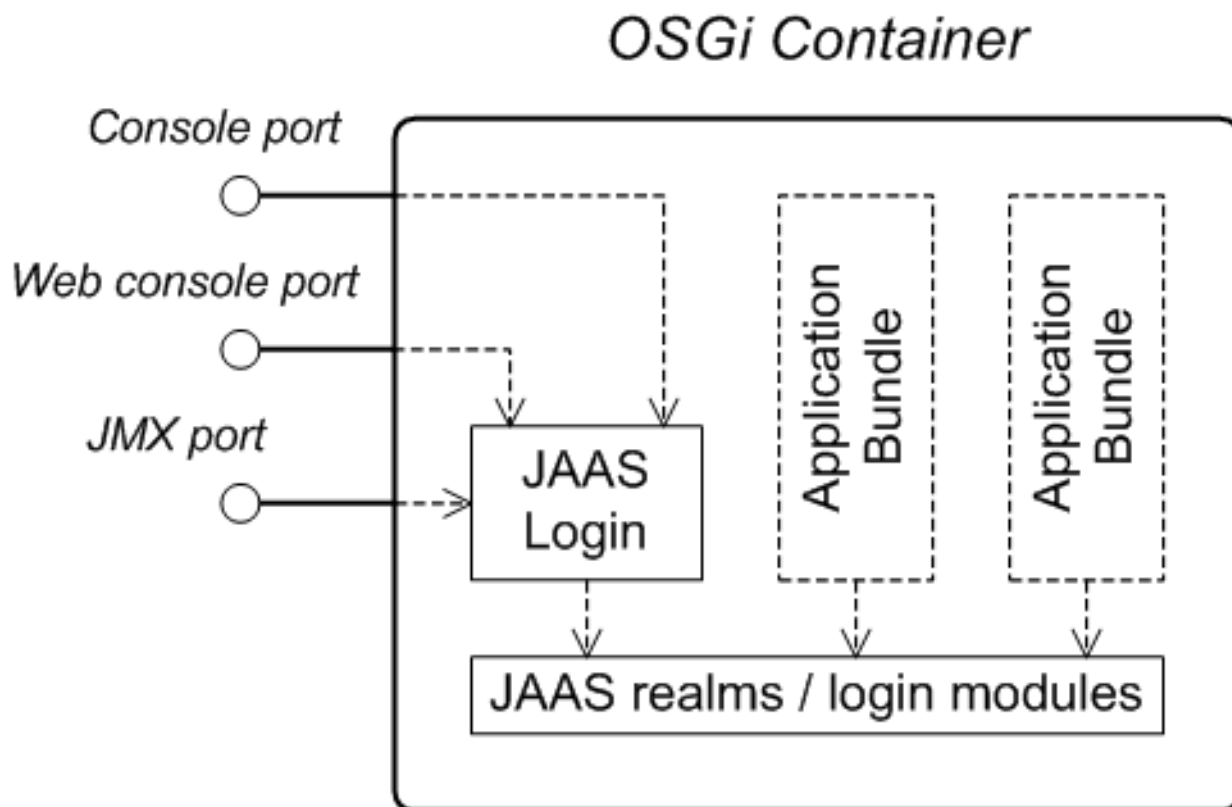
In the OSGi container, it is possible to deploy applications supporting a variety of security features. Currently, only the Java Authentication and Authorization Service (JAAS) is based on a common, container-wide infrastructure. Other security features are provided separately by the individual products and components deployed in the container.

1.1. OSGi CONTAINER SECURITY

Overview

Figure 1.1, “OSGi Container Security Architecture” shows an overview of the security infrastructure that is used across the container and is accessible to all bundles deployed in the container. This common security infrastructure currently consists of a mechanism for making JAAS realms (or login modules) available to all application bundles.

Figure 1.1. OSGi Container Security Architecture



JAAS realms

A JAAS realm or login module is a plug-in module that provides authentication and authorization data to Java applications, as defined by the [Java Authentication and Authorization Service \(JAAS\)](#) specification.

Red Hat AMQ supports a special mechanism for defining JAAS login modules (in either a Spring or a blueprint file), which makes the login module accessible to all bundles in the container. This makes it easy for multiple applications running in the OSGi container to consolidate their security data into a single JAAS realm.

karaf realm

The OSGi container has a predefined JAAS realm, the **karaf** realm. Red Hat AMQ uses the **karaf** realm to provide authentication for remote administration of the OSGi runtime, for the Fuse Management Console, and for JMX management. The **karaf** realm uses a simple file-based repository, where authentication data is stored in the ***InstallDir/etc/users.properties*** file.

You can use the **karaf** realm in your own applications. Simply configure **karaf** as the name of the JAAS realm that you want to use. Your application then performs authentication using the data from the **users.properties** file.

Console port

You can administer the OSGi container remotely either by connecting to the console port with a Karaf client or using the Karaf **ssh:ssh** command. The console port is secured by a JAAS login feature that connects to the **karaf** realm. Users that try to connect to the console port will be prompted to enter a username and password that must match one of the accounts from the **karaf** realm.

JMX port

You can manage the OSGi container by connecting to the JMX port (for example, using Java's JConsole). The JMX port is also secured by a JAAS login feature that connects to the **karaf** realm.

Application bundles and JAAS security

Any application bundles that you deploy into the OSGi container can access the container's JAAS realms. The application bundle simply references one of the existing JAAS realms by name (which corresponds to an instance of a JAAS login module).

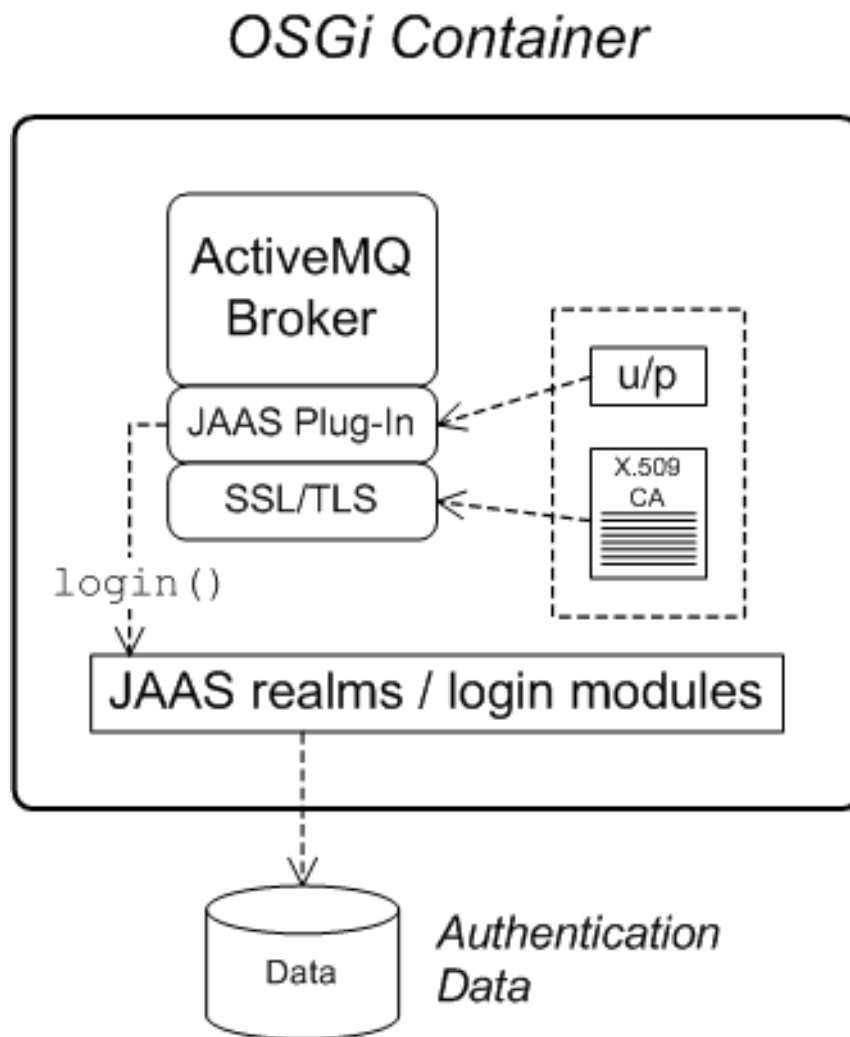
It is essential, however, that the JAAS realms are defined using the OSGi container's own login configuration mechanism—by default, Java provides a simple file-based login configuration implementation, but you *cannot* use this implementation in the context of the OSGi container.

1.2. APACHE ACTIVEMQ SECURITY

Overview

Figure 1.2, “[Apache ActiveMQ Security Architecture](#)” shows an overview of the Apache ActiveMQ security architecture. The main security features supported by Apache ActiveMQ are the SSL/TLS security layer and the JAAS security layer. The SSL/TLS security layer provides message encryption and identifies the broker to its clients, while the JAAS security layer identifies clients to the broker.

Figure 1.2. Apache ActiveMQ Security Architecture



SSL/TLS security

Apache ActiveMQ supports the use of SSL/TLS to secure client-to-broker and broker-to-broker connections, where the underlying SSL/TLS implementation is provided by the Java Secure Socket Extension (JSSE). When deploying brokers and clients in an OSGi container, you cannot configure SSL/TLS security using JSSE system properties, however. You must either use XML configuration (for example, in a Spring or a blueprint file) or set the security properties by programming.

For more details, see [Chapter 7, SSL/TLS Security](#).

JAAS security

Apache ActiveMQ also supports JAAS security, which typically requires clients to log on to the broker by providing username and password credentials. When deployed in an OSGi container, the broker's JAAS security must be integrated with the container's JAAS security (as described in [Section 1.1, "OSGi Container Security"](#)).

CHAPTER 2. SECURING THE CONTAINER

Abstract

The Red Hat AMQ container is secured using JAAS. By defining JAAS realms, you can configure the mechanism used to retrieve user credentials. You can also refine access to the container's administrative interfaces by changing the default roles.

2.1. JAAS AUTHENTICATION

Abstract

The Java Authentication and Authorization Service (JAAS) provides a general framework for implementing authentication in a Java application. The implementation of authentication is modular, with individual JAAS modules (or plug-ins) providing the authentication implementations.

For background information about JAAS, see the [JAAS Reference Guide](#).

2.1.1. Default JAAS Realm

Overview

This section describes how to manage user data for the default JAAS realm in a standalone container.

Default JAAS realm

The Red Hat AMQ container has a predefined JAAS realm, the **karaf** realm, which is used by default to secure all aspects of the container.

How to integrate an application with JAAS

You can use the **karaf** realm in your own applications. Simply configure **karaf** as the name of the JAAS realm that you want to use.

Default JAAS login modules

When you start AMQ for the first time, the container is configured as a standalone container and uses the **karaf** default realm. In this default configuration, the **karaf** realm deploys four JAAS login modules, which are enabled simultaneously. To see the deployed login modules, enter the **jaas:realms** console command, as follows:

```
JBossFuse:karaf@root> jaas:realms
Index Realm  Module Class
  1 karaf  org.apache.karaf.jaas.modules.properties.PropertiesLoginModule
  2 karaf  org.apache.karaf.jaas.modules.publickey.PublickeyLoginModule
  3 karaf  org.apache.karaf.jaas.modules.audit.FileAuditLoginModule
  4 karaf  org.apache.karaf.jaas.modules.audit.EventAdminAuditLoginModule
```

**IMPORTANT**

In a standalone container, *both* the properties login module and the public key login module are enabled. When JAAS authenticates a user, it tries first of all to authenticate the user with the properties login module. If that fails, it then tries to authenticate the user with the public key login module. If that module also fails, an error is raised.

**NOTE**

The **FileAuditLoginModule** login module and the **EventAdminAuditLoginModule** login module are used to record an audit trail of successful and failed login attempts. These login modules do *not* authenticate users.

**IMPORTANT**

For JAAS login modules that reference properties files, the reload behavior is conditional and disabled by default. To enable the behavior, set the **reload** property to **true** as shown in the example below:

```
<jaas:config name="PropertiesLogin">
  <jaas:module flags="required"
    className="org.apache.activemq.jaas.PropertiesLoginModule">
    reload=true
    org.apache.activemq.jaas.properties.user=users.properties
    org.apache.activemq.jaas.properties.group=groups.properties
  </jaas:module>
</jaas:config>
```

Configuring users in the properties login module

The properties login module is used to store username/password credentials in a flat file format. To create a new user in the properties login module, open the **InstallDir/etc/users.properties** file using a text editor and add a line with the following syntax:

```
Username=Password[,UserGroup|Role][,UserGroup|Role]...
```

For example, to create the **jdoe** user with password, **topsecret**, and role, **Administrator**, you could create an entry like the following:

```
jdoe=topsecret,Administrator
```

Where the **Administrator** role gives full administrative privileges to the **jdoe** user.

Configuring user groups in the properties login module

Instead of (or in addition to) assigning roles directly to users, you also have the option of adding users to *user groups* in the properties login module. To create a user group in the properties login module, open the **InstallDir/etc/users.properties** file using a text editor and add a line with the following syntax:

```
_g_\:GroupName=Role1,Role2,...
```

For example, to create the **adminingroup** user group with the roles, **SuperUser** and **Administrator**, you could create an entry like the following:

```
_g_\:admingroup=SuperUser,Administrator
```

You could then add the **majorclanger** user to the **admingroup**, by creating the following user entry:

```
majorclanger=secretpass,_g_\:admingroup
```

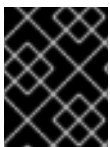
Configuring the public key login module

The public key login module is used to store SSH public key credentials in a flat file format. To create a new user in the public key login module, open the ***InstallDir/etc/keys.properties*** file using a text editor and add a line with the following syntax:

```
Username=PublicKey[,UserGroup|Role][,UserGroup|Role]...
```

For example, you can create the **jdoue** user with the **Administrator** role by adding the following entry to the ***InstallDir/etc/keys.properties*** file (on a single line):

```
jdoue=AAAAB3NzaC1kc3MAAACBAP1/U4EddRIpUt9KnC7s5Of2EbdSPO9EAMMeP4C2USZpRV1AIIH
7WT2NWPq/xfW6MPbLm1Vs14E7
gB00b/JmYLdrmVCIpJ+f6AR7ECLCT7up1/63xhv4O1fnfqimFQ8E+4P208UewwI1VBNaFpEy9nXzrith1y
rv8iIDGZ3RSAHHAAAFQCX
YFCPFSMLzLKSuYKi64QL8Fgc9QAAAnEA9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0Hgmd
RWVeOutRZT+ZxBxCBGLRjFnEj6Ewo
FhO3zwkyjMim4TwWeotifl0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhRkImog9/hWuWfBpKL
Zl6Ae1UIZAFMO/7PSSoAAACB
AKKSU2PFI/qOLxIwmBZPPICjshVe7bVUpFvyl3BbJDow8rXfsl8wO63OzP/qLmcJM0+JbcRU/53Jj7uyk
31drV2qxhIOsLDC9dGCWj4
7Y7TyhPdXh/0dthTRBy6bqGtRPxGa7gJov1xm/UuYYXPIUR/3x9MAZvZ5xvE0kYXO+rx,Administrat
or
```



IMPORTANT

Do not insert the entire contents of an **id_rsa.pub** file here. Insert just the block of symbols which represents the public key itself.

Configuring user groups in the public key login module

Instead of (or in addition to) assigning roles directly to users, you also have the option of adding users to *user groups* in the public key login module. To create a user group in the public key login module, open the ***InstallDir/etc/keys.properties*** file using a text editor and add a line with the following syntax:

```
_g_\:GroupName=Role1,Role2,...
```

For example, to create the **admingroup** user group with the roles, **SuperUser** and **Administrator**, you could create an entry like the following:

```
_g_\:admingroup=SuperUser,Administrator
```

You could then add the **jdoue** user to the **admingroup**, by creating the following user entry:

```
jdoue=AAAAB3NzaC1kc3MAAACBAP1/U4EddRIpUt9KnC7s5Of2EbdSPO9EAMMeP4C2USZpRV1AIIH
7WT2NWPq/xfW6MPbLm1Vs14E7
```

```
gB00b/JmYLdrmVCIpJ+f6AR7ECLCT7up1/63xhv4O1fnfqimFQ8E+4P208Uewwl1VBNaFpEy9nXzrith1y
rv8iIDGZ3RSAHHAAAAFQCX
YFCPFSMLzLKSuYKi64QL8Fgc9QAAAnEA9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0Hgmd
RWVeOutRZT+ZxBxCBGLRJFnEj6Ewo
FhO3zwkyjMim4TwWeotifl0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhRklmog9/hWuWfBpKL
Zl6Ae1UIZAFMO/7PSSoAAACB
AKKSU2PFI/qOLxIwmBZPPIcJshVe7bVUpFvyl3BbJDow8rXfskl8wO63OzP/qLmcJM0+JbcRU/53Jj7uyk
31drV2qxhIOsLDC9dGCWj4
7Y7TyhPdXh/0dthTRBy6bqGtRPxGa7gJov1xm/UuYYXPIUR/3x9MAZvZ5xvE0kYXO+rx,_g_:adming
roup
```

Encrypting the stored passwords

By default, passwords are stored in the `InstallDir/etc/users.properties` file in plaintext format. To protect the passwords in this file, you must set the file permissions of the `users.properties` file so that it can be read only by administrators. To provide additional protection, you can optionally encrypt the stored passwords using a message digest algorithm.

To enable the password encryption feature, edit the `InstallDir/etc/org.apache.karaf.jaas.cfg` file and set the encryption properties as described in the comments. For example, the following settings would enable basic encryption using the MD5 message digest algorithm:

```
encryption.enabled = true
encryption.name = basic
encryption.prefix = {CRYPT}
encryption.suffix = {CRYPT}
encryption.algorithm = MD5
encryption.encoding = hexadecimal
```



NOTE

The encryption settings in the `org.apache.karaf.jaas.cfg` file are applied *only* to the default `karaf` realm in a standalone container. They have no effect on a Fabric container and no effect on a custom realm.

For more details about password encryption, see [Section 2.1.8, “Encrypting Stored Passwords”](#).

Overriding the default realm

If you want to customise the JAAS realm, the most convenient approach to take is to override the default `karaf` realm by defining a higher ranking `karaf` realm. This ensures that all of the Red Hat AMQ security components switch to use your custom realm. For details of how to define and deploy custom JAAS realms, see [Section 2.1.2, “Defining JAAS Realms”](#).

2.1.2. Defining JAAS Realms

Overview

When defining a JAAS realm in the OSGi container, you *cannot* put the definitions in a conventional JAAS [login configuration](#) file. Instead, the OSGi container uses a special `jaas:config` element for defining JAAS realms in a blueprint configuration file. The JAAS realms defined in this way are made available to *all* of the application bundles deployed in the container, making it possible to share the JAAS security infrastructure across the whole container.

Namespace

The `jaas:config` element is defined in the `http://karaf.apache.org/xmlns/jaas/v1.0.0` namespace. When defining a JAAS realm you will need to include the line shown in [Example 2.1, “JAAS Blueprint Namespace”](#).

Example 2.1. JAAS Blueprint Namespace

```
xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
```

Configuring a JAAS realm

The syntax for the `jaas:config` element is shown in [Example 2.2, “Defining a JAAS Realm in Blueprint XML”](#).

Example 2.2. Defining a JAAS Realm in Blueprint XML

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0">

  <jaas:config name="JaasRealmName"
    [rank="IntegerRank"]>
    <jaas:module className="LoginModuleClassName"
      [flags="[required|requisite|sufficient|optional]"]>
      Property=Value
      ...
    </jaas:module>
    ...
    <!-- Can optionally define multiple modules -->
    ...
  </jaas:config>

</blueprint>
```

The elements are used as follows:

`jaas:config`

Defines the JAAS realm. It has the following attributes:

- **name**—specifies the name of the JAAS realm.
- **rank**—specifies an optional rank for resolving naming conflicts between JAAS realms . When two or more JAAS realms are registered under the same name, the OSGi container always picks the realm instance with the highest rank. If you decide to override the default realm, **karaf**, you should specify a **rank** of **100** or more, so that it overrides all of the previously installed **karaf** realms (in the context of Fabric, you need to override the default **ZookeeperLoginModule**, which has a rank of **99**).

`jaas:module`

Defines a JAAS login module in the current realm. `jaas:module` has the following attributes:

- **className**—the fully-qualified class name of a JAAS login module. The specified class must be available from the bundle classloader.
- **flags**—determines what happens upon success or failure of the login operation. [Table 2.1, “Flags for Defining a JAAS Module”](#) describes the valid values.

Table 2.1. Flags for Defining a JAAS Module

Value	Description
required	Authentication of this login module must succeed. Always proceed to the next login module in this entry, irrespective of success or failure.
requisite	Authentication of this login module must succeed. If success, proceed to the next login module; if failure, return immediately without processing the remaining login modules.
sufficient	Authentication of this login module is not required to succeed. If success, return immediately without processing the remaining login modules; if failure, proceed to the next login module.
optional	Authentication of this login module is not required to succeed. Always proceed to the next login module in this entry, irrespective of success or failure.

The contents of a **jaas:module** element is a space separated list of property settings, which are used to initialize the JAAS login module instance. The specific properties are determined by the JAAS login module and must be put into the proper format.

**NOTE**

You can define multiple login modules in a realm.

Converting standard JAAS login properties to XML

Red Hat AMQ uses the same properties as a standard Java login configuration file, however Red Hat AMQ requires that they are specified slightly differently. To see how the Red Hat AMQ approach to defining JAAS realms compares with the standard Java login configuration file approach, consider how to convert the login configuration shown in [Example 2.3, “Standard JAAS Properties”](#), which defines the **PropertiesLogin** realm using the Red Hat AMQ properties login module class, **PropertiesLoginModule**:

Example 2.3. Standard JAAS Properties

```
PropertiesLogin {
    org.apache.activemq.jaas.PropertiesLoginModule required
```



```

    org.apache.activemq.jaas.properties.user="users.properties"
    org.apache.activemq.jaas.properties.group="groups.properties";
};

```

The equivalent JAAS realm definition, using the `jaas:config` element in a blueprint file, is shown in [Example 2.4, "Blueprint JAAS Properties"](#).

Example 2.4. Blueprint JAAS Properties

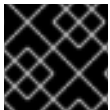
```

<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0">

  <jaas:config name="PropertiesLogin">
    <jaas:module flags="required"
      className="org.apache.activemq.jaas.PropertiesLoginModule">
      org.apache.activemq.jaas.properties.user=users.properties
      org.apache.activemq.jaas.properties.group=groups.properties
    </jaas:module>
  </jaas:config>

</blueprint>

```



IMPORTANT

You **do not** use double quotes for JAAS properties in the blueprint configuration.

Example

Red Hat AMQ also provides an adapter that enables you to store JAAS authentication data in an X.500 server. [Example 2.5, "Configuring a JAAS Realm"](#) defines the **LDAPLogin** realm to use Red Hat AMQ's **LDAPLoginModule** class, which connects to the LDAP server located at `ldap://localhost:10389`.

Example 2.5. Configuring a JAAS Realm

```

<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0">

  <jaas:config name="LDAPLogin" rank="200">
    <jaas:module flags="required"
      className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule">
      initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
      connection.username=uid=admin,ou=system
      connection.password=secret
      connection.protocol=
      connection.url = ldap://localhost:10389
      user.base.dn = ou=users,ou=system
      user.filter = (uid=%u)
      user.search.subtree = true
    </jaas:module>
  </jaas:config>

```

```
role.base.dn = ou=users,ou=system
role.filter = (uid=%u)
role.name.attribute = ou
role.search.subtree = true
authentication = simple
</jaas:module>
</jaas:config>
</blueprint>
```

For a detailed description and example of using the LDAP login module, see [Section 2.1.7, "JAAS LDAP Login Module"](#).

2.1.3. JAAS Properties Login Module

Overview

The JAAS properties login module stores user data in a flat file format (where the stored passwords can optionally be encrypted using a message digest algorithm). The user data can either be edited directly, using a simple text editor, or managed using the **jaas:*** console commands.

For example, a standalone container uses the JAAS properties login module by default and stores the associated user data in the ***InstallDir/etc/users.properties*** file.

Supported credentials

The JAAS properties login module authenticates username/password credentials, returning the list of roles associated with the authenticated user.

Implementation classes

The following classes implement the JAAS properties login module:

org.apache.karaf.jaas.modules.properties.PropertiesLoginModule

Implements the JAAS login module.

org.apache.karaf.jaas.modules.properties.PropertiesBackingEngineFactory

Must be exposed as an OSGi service. This service makes it possible for you to manage the user data using the **jaas:*** console commands from the Apache Karaf shell (see [chapter "JAAS Console Commands" in "Console Reference"](#)).

Options

The JAAS properties login module supports the following options:

users

Location of the user properties file.

Format of the user properties file

The user properties file is used to store username, password, and role data for the properties login module. Each user is represented by a single line in the user properties file, where a line has the following form:

```
Username=Password[,UserGroup|Role][,UserGroup|Role]...
```

User groups can also be defined in this file, where each user group is represented by a single line in the following format:

```
_g_\:GroupName=Role1[,Role2]...
```

For example, you can define the users, **bigcheese** and **guest**, and the user groups, **admingroup** and **guestgroup**, as follows:

```
# Users
bigcheese=cheesepass,_g_:admingroup
guest=guestpass,_g_:guestgroup

# Groups
_g_\:admingroup=SuperUser,Administrator
_g_\:guestgroup=Monitor
```

Sample Blueprint configuration

The following Blueprint configuration shows how to define a new **karaf** realm using the properties login module, where the default **karaf** realm is overridden by setting the **rank** attribute to **200**:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
  xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.1.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0">

  <type-converters>
    <bean class="org.apache.karaf.jaas.modules.properties.PropertiesConverter"/>
  </type-converters>

  <!--Allow usage of System properties, especially the karaf.base property-->
  <ext:property-placeholder
    placeholder-prefix="$[" placeholder-suffix="]"/>

  <jaas:config name="karaf" rank="200">
    <jaas:module flags="required"
      className="org.apache.karaf.jaas.modules.properties.PropertiesLoginModule">
      users= $[karaf.base]/etc/users.properties
    </jaas:module>
  </jaas:config>

  <!-- The Backing Engine Factory Service for the PropertiesLoginModule -->
  <service interface="org.apache.karaf.jaas.modules.BackingEngineFactory">
    <bean class="org.apache.karaf.jaas.modules.properties.PropertiesBackingEngineFactory"/>
  </service>

</blueprint>
```

Remember to export the **BackingEngineFactory** bean as an OSGi service, so that the **jaas:*** console commands can manage the user data.

2.1.4. JAAS OSGi Config Login Module

Overview

The JAAS OSGi config login modules leverages the *OSGi Config Admin Service* to store user data. This login module is fairly similar to the JAAS properties login module (for example, the syntax of the user entries is the same), but the mechanism for retrieving user data is based on the OSGi Config Admin Service.

The user data can be edited directly by creating a corresponding OSGi configuration file, **etc/PersistentID.cfg** or using any method of configuration that is supported by the OSGi Config Admin Service. The **jaas:*** console commands are not supported, however.

Supported credentials

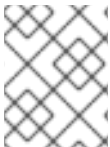
The JAAS OSGi config login module authenticates username/password credentials, returning the list of roles associated with the authenticated user.

Implementation classes

The following classes implement the JAAS OSGi config login module:

org.apache.karaf.jaas.modules.osgi.OsgiConfigLoginModule

Implements the JAAS login module.



NOTE

There is no backing engine factory for the OSGi config login module, which means that this module cannot be managed using the **jaas:*** console commands.

Options

The JAAS OSGi config login module supports the following options:

pid

The *persistent ID* of the OSGi configuration containing the user data. In the OSGi Config Admin standard, a persistent ID references a set of related configuration properties.

Location of the configuration file

The location of the configuration file follows the usual convention where the configuration for the persistent ID, **PersistentID**, is stored in the following file:

```
InstallDir/etc/PersistentID.cfg
```

Format of the configuration file

The **PersistentID.cfg** configuration file is used to store username, password, and role data for the OSGi config login module. Each user is represented by a single line in the configuration file, where a line has the following form:

```
Username=Password[,Role][,Role]...
```



NOTE

User groups are *not* supported in the JAAS OSGi config login module.

Sample Blueprint configuration

The following Blueprint configuration shows how to define a new **karaf** realm using the OSGi config login module, where the default **karaf** realm is overridden by setting the **rank** attribute to **200**:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
  xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.1.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0">

  <jaas:config name="karaf" rank="200">
    <jaas:module flags="required"
      className="org.apache.karaf.jaas.modules.osgi.OsgiConfigLoginModule">
      pid = org.jboss.example.osgiconfigloginmodule
    </jaas:module>
  </jaas:config>

</blueprint>
```

In this example, the user data will be stored in the file, **InstallDir/etc/org.jboss.example.osgiconfigloginmodule.cfg**, and it is not possible to edit the configuration using the **jaas:*** console commands.

2.1.5. JAAS Public Key Login Module

Overview

The JAAS public key login module stores user data in a flat file format, which can be edited directly using a simple text editor. The **jaas:*** console commands are not supported, however.

For example, a standalone container uses the JAAS public key login module by default and stores the associated user data in the **InstallDir/etc/keys.properties** file.

Supported credentials

The JAAS public key login module authenticates SSH key credentials. When a user tries to log in, the SSH protocol uses the stored public key to challenge the user. The user must possess the corresponding private key in order to answer the challenge. If login is successful, the login module returns the list of roles associated with the user.

Implementation classes

The following classes implement the JAAS public key login module:

org.apache.karaf.jaas.modules.publickey.PublickeyLoginModule

Implements the JAAS login module.



NOTE

There is no backing engine factory for the public key login module, which means that this module cannot be managed using the **jaas:*** console commands.

Options

The JAAS public key login module supports the following options:

users

Location of the user properties file for the public key login module.

Format of the keys properties file

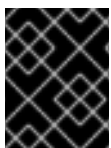
The **keys.properties** file is used to store username, public key, and role data for the public key login module. Each user is represented by a single line in the keys properties file, where a line has the following form:

```
Username=PublicKey[,UserGroup|Role][,UserGroup|Role]...
```

Where the *PublicKey* is the public key part of an SSH key pair (typically found in a user's home directory in `~/.ssh/id_rsa.pub` in a UNIX system).

For example, to create the user **jdoe** with the **Administrator** role, you would create an entry like the following:

```
jdoe=AAAAB3NzaC1kc3MAAACBAP1/U4EddRIpUt9KnC7s5Of2EbdSPO9EAMMeP4C2USZpRV1AIIH
7WT2NWPq/xfW6MPbLm1Vs14E7
gB00b/JmYldrmVCIpJ+f6AR7ECLCT7up1/63xhv4O1fnfqimFQ8E+4P208Uewwl1VBNaFpEy9nXzrith1y
rv8iIDGZ3RSAHHAAAAFQCX
YFCPFSMLzLKSuYKi64QL8Fgc9QAAAnEA9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0Hgmd
RWVeOutRZT+ZxBxCBGLRjFnEj6Ewo
FhO3zwkyjMim4TwwEotifl0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhRklmog9/hWuWfBpKL
ZI6Ae1UIZAFMO/7PSSoAAACB
AKKSU2PFI/qOLxIwmBZPPicJshVe7bVUpFvyl3BbJDow8rXfskl8wO63OzP/qLmcJM0+JbcRU/53Jj7uyk
31drV2qxhIOsLDC9dGCWj4
7Y7TyhPdXh/0dthTRBy6bqGtRPxGa7gJov1xm/UuYYXPIUR/3x9MAZvZ5xvE0kYXO+rx,Administrat
or
```



IMPORTANT

Do not insert the entire contents of the **id_rsa.pub** file here. Insert just the block of symbols which represents the public key itself.

User groups can also be defined in this file, where each user group is represented by a single line in the following format:

```
_g_\:GroupName=Role1[,Role2]...
```

Sample Blueprint configuration

The following Blueprint configuration shows how to define a new **karaf** realm using the public key login module, where the default **karaf** realm is overridden by setting the **rank** attribute to **200**:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
  xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.1.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0">

<!--Allow usage of System properties, especially the karaf.base property-->
<ext:property-placeholder
  placeholder-prefix="$[" placeholder-suffix="]"/>

<jaas:config name="karaf" rank="200">
  <jaas:module flags="required"
className="org.apache.karaf.jaas.modules.publickey.PublickeyLoginModule">
    users = ${karaf.base}/etc/keys.properties
  </jaas:module>
</jaas:config>

</blueprint>
```

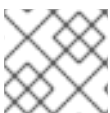
In this example, the user data will be stored in the file, **InstallDir/etc/keys.properties**, and it is not possible to edit the configuration using the **jaas:*** console commands.

2.1.6. JAAS JDBC Login Module

Overview

The JAAS JDBC login module enables you to store user data in a database back-end, using Java Database Connectivity (JDBC) to connect to the database. Hence, you can use any database that supports JDBC to store your user data. To manage the user data, you can use either the native database client tools or the **jaas:*** console commands (where the backing engine uses configured SQL queries to perform the relevant database updates).

You can combine multiple login modules with each login module providing both the authentication and authorization components. For example, you can combine default **PropertiesLoginModule** with **JDBCLoginModule** to ensure access to the system.



NOTE

User groups are *not* supported in the JAAS JDBC login module.

Supported credentials

The JAAS JDBC Login Module authenticates username/password credentials, returning the list of roles associated with the authenticated user.

Implementation classes

The following classes implement the JAAS JDBC Login Module:

org.apache.karaf.jaas.modules.jdbc.JDBCLoginModule

Implements the JAAS login module.

org.apache.karaf.jaas.modules.jdbc.JDBCBackingEngineFactory

Must be exposed as an OSGi service. This service makes it possible for you to manage the user data using the **jaas:*** console commands from the Apache Karaf shell (see [chapter "JAAS Console Commands" in "Console Reference"](#)).

Options

The JAAS JDBC login module supports the following options:

datasource

The JDBC data source, specified either as an OSGi service or as a JNDI name. You can specify a data source's OSGi service using the following syntax:

```
osgi:ServiceInterfaceName[/ServicePropertiesFilter]
```

The *ServiceInterfaceName* is the interface or class that is exported by the data source's OSGi service (usually **javax.sql.DataSource**).

Because multiple data sources can be exported as OSGi services in a container, it is usually necessary to specify a filter, *ServicePropertiesFilter*, to select the particular data source that you want. Filters on OSGi services are applied to the service property settings and follow a syntax that is borrowed from LDAP filter syntax.

query.password

The SQL query that retrieves the user's password. The query can contain a single question mark character, **?**, which is substituted by the username at run time.

query.role

The SQL query that retrieves the user's roles. The query can contain a single question mark character, **?**, which is substituted by the username at run time.

insert.user

The SQL query that creates a new user entry. The query can contain two question marks, **?**, characters: the first question mark is substituted by the username and the second question mark is substituted by the password at run time.

insert.role

The SQL query that adds a role to a user entry. The query can contain two question marks, **?**, characters: the first question mark is substituted by the username and the second question mark is substituted by the role at run time.

delete.user

The SQL query that deletes a user entry. The query can contain a single question mark character, **?**, which is substituted by the username at run time.

delete.role

The SQL query that deletes a role from a user entry. The query can contain two question marks, **?**, characters: the first question mark is substituted by the username and the second question mark is substituted by the role at run time.

delete.roles

The SQL query that deletes multiple roles from a user entry. The query can contain a single question mark character, **?**, which is substituted by the username at run time.

Example of setting up a JDBC login module

To set up a JDBC login module, perform the following main steps:

1. [the section called "Create the database tables"](#)
2. [the section called "Create the data source"](#)
3. [the section called "Specify the data source as an OSGi service"](#)

Create the database tables

Before you can set up the JDBC login module, you must set up a users table and a roles table in the backing database to store the user data. For example, the following SQL commands show how to create a suitable **users** table and **roles** table:

```
CREATE TABLE users (
  username VARCHAR(255) NOT NULL,
  password VARCHAR(255) NOT NULL,
  PRIMARY KEY (username)
);
CREATE TABLE roles (
  username VARCHAR(255) NOT NULL,
  role VARCHAR(255) NOT NULL,
  PRIMARY KEY (username,role)
);
```

The **users** table stores username/password data and the **roles** table associates a username with one or more roles.

Create the data source

To use a JDBC datasource with the JDBC login module, the correct approach to take is to create a data source instance and export the data source as an OSGi service. The JDBC login module can then access the data source by referencing the exported OSGi service. For example, you could create a MySQL data source instance and expose it as an OSGi service (of **javax.sql.DataSource** type) using code like the following in a Blueprint file:

```
<blueprint xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
  <bean id="mysqlDatasource"
    class="com.mysql.jdbc.jdbc2.optional.MysqlDataSource">
    <property name="serverName" value="localhost"></property>
    <property name="databaseName" value="DBName"></property>
```

```

<property name="port" value="3306"></property>
<property name="user" value="DBUser"></property>
<property name="password" value="DBPassword"></property>
</bean>

<service id="mysqlDS" interface="javax.sql.DataSource"
  ref="mysqlDatasource">
  <service-properties>
    <entry key="osgi.jndi.service.name" value="jdbc/karafdb"/>
  </service-properties>
</service>
</blueprint>

```

The preceding Blueprint configuration should be packaged and installed in the container as an OSGi bundle.

Specify the data source as an OSGi service

After the data source has been instantiated and exported as an OSGi service, you are ready to configure the JDBC login module. In particular, the **datasource** option of the JDBC login module can reference the data source's OSGi service using the following syntax:

```
osgi:javax.sql.DataSource/(osgi.jndi.service.name=jdbc/karafdb)
```

Where **javax.sql.DataSource** is the interface type of the exported OSGi service and the filter, **(osgi.jndi.service.name=jdbc/karafdb)**, selects the particular **javax.sql.DataSource** instance whose **osgi.jndi.service.name** service property has the value, **jdbc/karafdb**.

For example, you can use the following Blueprint configuration to override the **karaf** realm with a JDBC login module that references the sample MySQL data source:

```

<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
  xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.1.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0">

  <!--Allow usage of System properties, especially the karaf.base property-->
  <ext:property-placeholder
    placeholder-prefix="$[" placeholder-suffix="]"/>

  <jaas:config name="karaf" rank="200">
    <jaas:module flags="required"
      className="org.apache.karaf.jaas.modules.jdbc.JDBCLoginModule">
      datasource = osgi:javax.sql.DataSource/(osgi.jndi.service.name=jdbc/karafdb)
      query.password = SELECT password FROM users WHERE username=?
      query.role = SELECT role FROM roles WHERE username=?
      insert.user = INSERT INTO users VALUES(?,?)
      insert.role = INSERT INTO roles VALUES(?,?)
      delete.user = DELETE FROM users WHERE username=?
      delete.role = DELETE FROM roles WHERE username=? AND role=?
      delete.roles = DELETE FROM roles WHERE username=?
    </jaas:module>
  </jaas:config>

```

```

<!-- The Backing Engine Factory Service for the JDBCLoginModule -->
<service interface="org.apache.karaf.jaas.modules.BackingEngineFactory">
  <bean class="org.apache.karaf.jaas.modules.jdbc.JDBCBackingEngineFactory"/>
</service>

</blueprint>

```



NOTE

The SQL statements shown in the preceding configuration are in fact the default values of these options. Hence, if you create user and role tables consistent with these SQL statements, you could omit the options settings and rely on the defaults.

In addition to creating a `JDBCLoginModule`, the preceding Blueprint configuration also instantiates and exports a **JDBCBackingEngineFactory** instance, which enables you to manage the user data using the `jaas:*` console commands.

2.1.7. JAAS LDAP Login Module

Overview

The JAAS LDAP login module enables you to store user data in an LDAP database. To manage the stored user data, use a standard LDAP client tool. The `jaas:*` console commands are *not* supported.

For more details about using LDAP with Red Hat AMQ see [Chapter 9, LDAP Authentication Tutorial](#).



NOTE

User groups are *not* supported in the JAAS LDAP login module.



IMPORTANT

In a Fuse Fabric, the Zookeeper login module must always be enabled. Hence, if you want to enable the LDAP login module in a Fabric, both the Zookeeper login module and the LDAP login module must be enabled. See [Section 9.4, "Enable LDAP Authentication in the OSGi Container"](#) for details.

Supported credentials

The JAAS LDAP Login Module authenticates username/password credentials, returning the list of roles associated with the authenticated user.

Implementation classes

The following classes implement the JAAS LDAP Login Module:

org.apache.karaf.jaas.modules.Idap.LDAPLoginModule

Implements the JAAS login module. It is preloaded in the container, so you do not need to install its bundle.

**NOTE**

There is no backing engine factory for the LDAP Login Module, which means that this module cannot be managed using the **jaas:*** console commands.

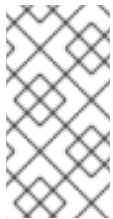
Options

The JAAS LDAP login module supports the following options:

authentication

Specifies the authentication method used when binding to the LDAP server. Valid values are

- **simple**—bind with user name and password authentication, requiring you to set the **connection.username** and **connection.password** properties.
- **none**—bind anonymously. In this case the **connection.username** and **connection.password** properties can be left unassigned.

**NOTE**

The connection to the directory server is used only for performing searches. In this case, an anonymous bind is often preferred, because it is faster than an authenticated bind (but you would also need to ensure that the directory server is sufficiently protected, for example by deploying it behind a firewall).

connection.url

Specifies specify the location of the directory server using an ldap URL, `ldap://Host:Port`. You can optionally qualify this URL, by adding a forward slash, `/`, followed by the DN of a particular node in the directory tree. To enable SSL security on the connection, you need to specify the **ldaps:** scheme in the URL—for example, `ldaps://Host:Port`. You can also specify multiple URLs, as a space-separated list, for example:

```
connection.url=ldap://10.0.0.153:2389 ldap://10.10.178.20:389
```

connection.username

Specifies the DN of the user that opens the connection to the directory server. For example, **uid=admin,ou=system**.

connection.password

Specifies the password that matches the DN from `connection.username`. In the directory server, the password is normally stored as a **userPassword** attribute in the corresponding directory entry.

context.com.sun.jndi.ldap.connect.pool

If **true**, enables connection pooling for LDAP connections. Default is **false**.

context.com.sun.jndi.ldap.connect.timeout

Specifies the timeout for creating a TCP connection to the LDAP server, in units of milliseconds. We recommend that you set this property explicitly, because the default value is infinite, which can result in a hung connection attempt.

context.com.sun.jndi.ldap.read.timeout

Specifies the read timeout for an LDAP operation, in units of milliseconds. We recommend that you set this property explicitly, because the default value is infinite.

context.java.naming.referral

An *LDAP referral* is a form of indirection supported by some LDAP servers. The LDAP referral is an entry in the LDAP server which contains one or more URLs (usually referencing a node or nodes in another LDAP server). The **context.java.naming.referral** property can be used to enable or disable referral following. It can be set to one of the following values:

- **follow** to follow the referrals (assuming it is supported by the LDAP server),
- **ignore** to silently ignore all referrals,
- **throw** to throw a **PartialResultException** whenever a referral is encountered.

disableCache

The user and role caches can be disabled by setting this property to **true**. Default is **false**.

initial.context.factory

Specifies the class of the context factory used to connect to the LDAP server. This must always be set to **com.sun.jndi.ldap.LdapCtxFactory**.

role.base.dn

Specifies the DN of the subtree of the DIT to search for role entries. For example, **ou=groups,ou=system**.

role.filter

Specifies the LDAP search filter used to locate roles. It is applied to the subtree selected by **role.base.dn**. For example, **(member=uid=%u)**. Before being passed to the LDAP search operation, the value is subjected to string substitution, as follows:

- **%u** is replaced by the user name extracted from the incoming credentials, and
- **%dn** is replaced by the RDN of the corresponding user in the LDAP server (which was found by matching against the **user.filter** filter).
- **%fqdn** is replaced by the DN of the corresponding user in the LDAP server (which was found by matching against the **user.filter** filter).

role.mapping

Specifies the mapping between LDAP groups and JAAS roles. If no mapping is specified, the default mapping is for each LDAP group to map to the corresponding JAAS role of the same name. The role mapping is specified with the following syntax:

```
ldap-group=jaas-role(,jaas-role)*(;ldap-group=jaas-role(,jaas-role)*)*
```

Where each LDAP group, **ldap-group**, is specified by its Common Name (CN). Note that the **role.mapping** option *must* be set to a non-empty value.

For example, given the LDAP groups, **admin**, **devop**, and **tester**, you could map them to JAAS roles, as follows:

```
role.mapping=admin=Administrator;devop=Administrator,Deployer;tester=Monitor
```

role.name.attribute

Specifies the attribute type of the role entry that contains the name of the role/group. If you omit this option, the role search feature is effectively disabled. For example, **cn**.

role.search.subtree

Specifies whether the role entry search scope includes the subtrees of the tree selected by **role.base.dn**. If **true**, the role lookup is recursive (**SUBTREE**). If **false**, the role lookup is performed only at the first level (**ONELEVEL**).

ssl

Specifies whether the connection to the LDAP server is secured using SSL. If `connection.url` starts with `ldaps://` SSL is used regardless of this property.

ssl.provider

Specifies the SSL provider to use for the LDAP connection. If not specified, the default SSL provider is used.

ssl.protocol

Specifies the protocol to use for the SSL connection. You *must* set this property to **TLSv1**, in order to prevent the SSLv3 protocol from being used (POODLE vulnerability).

ssl.algorithm

Specifies the algorithm used by the trust store manager. For example, **PKIX**.

ssl.keystore

The ID of the keystore that stores the LDAP client's own X.509 certificate (required only if SSL client authentication is enabled on the LDAP server). The keystore must be deployed using a **jaas:keystore** element (see [the section called "Sample configuration for Apache DS"](#)).

ssl.keyalias

The keystore alias of the LDAP client's own X.509 certificate (required only if there is more than one certificate stored in the keystore specified by **ssl.keystore**).

ssl.truststore

The ID of the keystore that stores trusted CA certificates, which are used to verify the LDAP server's certificate (the LDAP server's certificate chain must be signed by one of the certificates in the truststore). The keystore must be deployed using a **jaas:keystore** element.

user.base.dn

Specifies the DN of the subtree of the DIT to search for user entries. For example, **ou=users,ou=system**.

user.filter

Specifies the LDAP search filter used to locate user credentials. It is applied to the subtree selected by **user.base.dn**. For example, **(uid=%u)**. Before being passed to the LDAP search operation, the value is subjected to string substitution, as follows:

- `%u` is replaced by the user name extracted from the incoming credentials.

user.search.subtree

Specifies whether the user entry search scope includes the subtrees of the tree selected by **user.base.dn**. If **true**, the user lookup is recursive (**SUBTREE**). If **false**, the user lookup is performed only at the first level (**ONELEVEL**).

Sample configuration for Apache DS

The following Blueprint configuration shows how to define a new **karaf** realm using the LDAP login module, where the default **karaf** realm is overridden by setting the **rank** attribute to **200**, and the LDAP login module connects to an Apache Directory Server:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
  xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.1.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0">

  <jaas:config name="karaf" rank="100">

    <jaas:module className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule"
      flags="sufficient">
      debug=true

      <!-- LDAP Configuration -->
      initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
      <!-- multiple LDAP servers can be specified as a space separated list of URLs -->
      connection.url=ldap://10.0.0.153:2389 ldap://10.10.178.20:389

      <!-- authentication=none -->
      authentication=simple
      connection.username=cn=Directory Manager
      connection.password=directory

      <!-- User Info -->
      user.base.dn=dc=redhat,dc=com
      user.filter=(&objectClass=inetOrgPerson)(uid=%u)
      user.search.subtree=true

      <!-- Role/Group Info-->
      role.base.dn=dc=redhat,dc=com
      role.name.attribute=cn
      <!--
      The 'dc=redhat,dc=com' used in the role.filter
      below is the user.base.dn.
      -->
      <!-- role.filter=(uniquemember=%dn,dc=redhat,dc=com) -->
      role.filter=(&objectClass=GroupOfUniqueNames)(UniqueMember=%fqdn)
      role.search.subtree=true

      <!-- role mappings - a ';' separated list -->
      role.mapping=JBossAdmin=admin;JBossMonitor=Monitor,viewer
```

```

<!-- LDAP context properties -->
context.com.sun.jndi.ldap.connect.timeout=5000
context.com.sun.jndi.ldap.read.timeout=5000

<!-- LDAP connection pooling -->
<!-- http://docs.oracle.com/javase/jndi/tutorial/ldap/connect/pool.html -->
<!-- http://docs.oracle.com/javase/jndi/tutorial/ldap/connect/config.html -->
context.com.sun.jndi.ldap.connect.pool=true

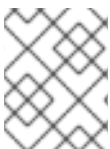
<!-- How are LDAP referrals handled?

Can be `follow`, `ignore` or `throw`. Configuring `follow` may not work on all LDAP servers,
`ignore` will
silently ignore all referrals, while `throw` will throw a partial results exception if there is a referral.
-->
context.java.naming.referral=ignore

<!-- SSL configuration -->
ssl=false
ssl.protocol=SSL
<!-- matches the keystore/truststore configured below -->
ssl.truststore=ks
ssl.algorithm=PKIX
<!-- The User and Role caches can be disabled - 6.3.0 179 and later -->
disableCache=true
</jaas:module>
</jaas:config>

<!-- Location of the SSL truststore/keystore
<jaas:keystore name="ks" path="file:///${karaf.home}/etc/ldap.truststore"
keystorePassword="XXXXXX" />
-->
</blueprint>

```



NOTE

In order to enable SSL, you must remember to use the **ldaps** scheme in the **connection.url** setting.



IMPORTANT

You must set **ssl.protocol** to **TLSv1**, in order to protect against the [Poodle vulnerability \(CVE-2014-3566\)](#)

Filter settings for different directory servers

The most significant differences between directory servers arise in connection with setting the filter options in the LDAP login module. The precise settings depend ultimately on the organisation of your DIT, but the following table gives an idea of the typical role filter settings required for different directory servers:

Directory Server	Typical Filter Settings
389-DS Red Hat DS	<pre>user.filter=(& (objectClass=InetOrgPerson)(uid=%u)) role.filter=(uniquemember=%fqdn)</pre>
MS Active Directory	<pre>user.filter=(&(objectCategory=person) (samAccountName=%u)) role.filter=(uniquemember=%fqdn)</pre>
Apache DS	<pre>user.filter=(uid=%u) role.filter=(member=uid=%u)</pre>
OpenLDAP	<pre>user.filter=(uid=%u) role.filter=(member:=uid=%u)</pre>



NOTE

In the preceding table, the **&** symbol (representing the logical *And* operator) is escaped as **&**, because the option settings will be embedded in a Blueprint XML file.

2.1.8. Encrypting Stored Passwords

Overview

By default, the JAAS login modules store passwords in plaintext format. Although you can (and should) protect such data by setting file permissions appropriately, you can provide additional protection to passwords by storing them in an obscured format (using a *message digest* algorithm).

Red Hat AMQ provides a set of options for enabling password encryption, which can be combined with any of the JAAS login modules (except the public key login module, where it is not needed).



IMPORTANT

Although message digest algorithms are difficult to crack, they are not invulnerable to attack (for example, see the [Wikipedia article on cryptographic hash functions](#)). Always use file permissions to protect files containing passwords, in addition to using password encryption.

Options

You can optionally enable password encryption for JAAS login modules by setting the following login module properties. To do so, either edit the `InstallDir/etc/org.apache.karaf.jaas.cfg` file or deploy your own blueprint file as described in [the section called “Example of a login module with Jasypt encryption”](#).

encryption.enabled

Set to **true**, to enable password encryption.

encryption.name

Name of the encryption service, which has been registered as an OSGi service.

encryption.prefix

Prefix for encrypted passwords.

encryption.suffix

Suffix for encrypted passwords.

encryption.algorithm

Specifies the name of the encryption algorithm—for example, **MD5** or **SHA-1**. You can specify one of the following encryption algorithms:

- **MD2**
- **MD5**
- **SHA-1**
- **SHA-256**
- **SHA-384**
- **SHA-512**

encryption.encoding

Encrypted passwords encoding: **hexadecimal** or **base64**.

encryption.providerName (*Jasypt only*)

Name of the **java.security.Provider** instance that is to provide the digest algorithm.

encryption.providerClassName (*Jasypt only*)

Class name of the security provider that is to provide the digest algorithm

encryption.iterations (*Jasypt only*)

Number of times to apply the hash function recursively.

encryption.saltSizeBytes (*Jasypt only*)

Size of the salt used to compute the digest.

encryption.saltGeneratorClassName (*Jasypt only*)

Class name of the salt generator.

role.policy

Specifies the policy for identifying role principals. Can have the values, **prefix** or **group**.

role.discriminator

Specifies the discriminator value to be used by the role policy.

Encryption services

There are two encryption services provided by AMQ:

- **encryption.name = basic**, described in [the section called “Basic encryption service”](#),
- **encryption.name = jasypt**, described in [the section called “Jasypt encryption”](#).

You can also create your own encryption service. To do so, you need to:

- implement interface **org.apache.karaf.jaas.modules.EncryptionService**
- and expose your implementation as OSGi service.

Following listing shows, how jasypt encryption service is exposed to OSGi container.

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
  <service interface="org.apache.karaf.jaas.modules.EncryptionService">
    <service-properties>
      <entry key="name" value="jasypt" />
    </service-properties>
    <bean class="org.apache.karaf.jaas.jasypt.impl.JasyptEncryptionService"/>
  </service>
  ...
</blueprint>
```

Basic encryption service

The basic encryption service is installed in the standalone container by default and you can reference it by setting the **encryption.name** property to the value, **basic**. In the basic encryption service, the message digest algorithms are provided by the [SUN](#) security provider (the default security provider in the Oracle JDK).

Jasypt encryption

By default, the Jasypt encryption service is installed on standalone JBoss Fuse, but not on standalone JBoss A-MQ. To install it on JBoss A-MQ, install the **jasypt-encryption** feature, using the following console command:

```
JBossA-MQ:karaf@root> features:install jasypt-encryption
```

This command installs the requisite Jasypt bundles and exports Jasypt encryption as an OSGi service, so that it is available for use by JAAS login modules. To access the Jasypt encryption service, set the **encryption.name** property to the value, **jasypt**.

For more information about Jasypt encryption, see the [Jasypt documentation](#).

Example of a login module with Jasypt encryption

Assuming that you have already installed the **jasypt-encryption** feature, you could deploy a properties login module with Jasypt encryption using the following Blueprint configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
  xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.1.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0">

  <type-converters>
    <bean class="org.apache.karaf.jaas.modules.properties.PropertiesConverter"/>
  </type-converters>

  <!--Allow usage of System properties, especially the karaf.base property-->
  <ext:property-placeholder
    placeholder-prefix="$[" placeholder-suffix="]"/>

  <jaas:config name="karaf" rank="200">
    <jaas:module flags="required"
      className="org.apache.karaf.jaas.modules.properties.PropertiesLoginModule">
      users = ${karaf.base}/etc/users.properties
      encryption.enabled = true
      encryption.name = jasypt
      encryption.algorithm = SHA-256
      encryption.encoding = base64
      encryption.iterations = 100000
      encryption.saltSizeBytes = 16
      encryption.prefix = {CRYPT}
      encryption.suffix = {CRYPT}
    </jaas:module>
  </jaas:config>

  <!-- The Backing Engine Factory Service for the PropertiesLoginModule -->
  <service interface="org.apache.karaf.jaas.modules.BackingEngineFactory">
    <bean class="org.apache.karaf.jaas.modules.properties.PropertiesBackingEngineFactory"/>
  </service>

  <!-- Enable automatic encryption of all user passwords
  in InstallDir/etc/users.properties file.
  No login required to activate.
  Encrypted passwords appear in the
  InstallDir/etc/users.properties file as values enclosed
  by {CRYPT}...{CRYPT} prefix/suffix pairs -->

  <bean init-method="init" destroy-method="destroy"
  class="org.apache.karaf.jaas.modules.properties.AutoEncryptionSupport">
    <argument>
      <map>
        <entry key="org.osgi.framework.BundleContext"
          value-ref="blueprintBundleContext"/>
        <entry key="users" value="${karaf.base}/etc/users.properties"/>
        <entry key="encryption.name" value="jasypt"/>
        <entry key="encryption.enabled" value="true"/>
        <entry key="encryption.prefix" value="{CRYPT}"/>
        <entry key="encryption.suffix" value="{CRYPT}"/>
        <entry key="encryption.algorithm" value="SHA-256"/>
      </map>
    </argument>
  </bean>

```

```

    <entry key="encryption.encoding" value="base64"/>
    <entry key="encryption.iterations" value="100000"/>
    <entry key="encryption.saltSizeBytes" value="16"/>
  </map>
</argument>
</bean>

</blueprint>

```

2.2. ROLE-BASED ACCESS CONTROL

Abstract

This section describes the role-based access control (RBAC) feature, which is enabled by default in the AMQ container. You can immediately start taking advantage of the RBAC feature, simply by adding one of the standard roles (such as **Deployer** or **Administrator**) to a user's credentials. For more advanced usage, you have the option of customizing the access control lists, in order to control exactly what each role can do. Finally, you have the option of applying custom ACLs to your own OSGi services.

2.2.1. Overview of Role-Based Access Control

Overview

By default, the AMQ role-based access control protects access through the Fuse Management Console, JMX connections, and the Karaf command console. To use the default levels of access control, simply add any of the standard roles to your user authentication data (for example, by editing the **etc/users.properties** file). You also have the option of customizing access control, by editing the relevant Access Control List (ACL) files.

Mechanisms

Role-based access control in AMQ is based on the following mechanisms:

JMX Guard

The AMQ container is configured with a JMX guard, which intercepts every incoming JMX invocation and filters the invocation through the configured JMX access control lists. The JMX guard is configured at the JVM level, so it intercepts every JMX invocation, without exception.

OSGi Service Guard

For any OSGi service, it is possible to configure an OSGi service guard. The OSGi service guard is implemented as a proxy object, which interposes itself between the client and the original OSGi service. An OSGi service guard must be explicitly configured for each OSGi service: it is not installed by default (except for the OSGi services that represent Karaf console commands, which are preconfigured for you).



NOTE

If you change the configuration of RBAC, the changes propagation to Hawtio may take up to 10 minutes due to cache update.

Types of protection

The AMQ implementation of role-based access control is capable of providing the following types of protection:

Fuse Management Console (Hawtio)

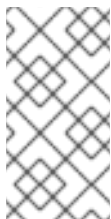
Container access through the Fuse Management Console (Hawtio) is controlled by the JMX ACL files. The REST/HTTP service that provides the Fuse Management Console is implemented using Jolokia technology, which is layered above JMX. Hence, ultimately, all Fuse Management Console invocations pass through JMX and are regulated by JMX ACLs.

JMX

Direct access to the container's JMX port is regulated by the JMX ACLs. Moreover, any additional JMX ports opened by an application running in the container would also be regulated by the JMX ACLs, because the JMX guard is set at the JVM level.

Karaf command console

Access to the Karaf command console is regulated by the command console ACL files. Access control is applied no matter how the Karaf console is accessed. Whether accessing the command console through the Fuse Management Console or through the SSH protocol, access control is applied in both cases.



NOTE

In the special case where you start up the container directly at the command line (for example, using the `./bin/fuse` script) and no user authentication is performed, you automatically get the roles specified by the `karaf.local.roles` property in the `etc/system.properties` file.

OSGi services

For any OSGi service deployed in the container, you can optionally enable an ACL file, which restricts method invocations to specific roles.

Adding roles to users

In the system of role-based access control, you can give users permissions by adding roles to their user authentication data. For example, the following entry in the `etc/users.properties` file defines the `admin` user and grants the `Administrator` and `SuperUser` roles.

```
admin = secretpass,Administrator,SuperUser
```

You also have the option of defining user groups and then assigning users to a particular user group. For example, you could define and use an `admingroup` user group as follows:

```
admin = secretpass, _g_:admingroup
_g_\:admingroup = Administrator, SuperUser
```



NOTE

User groups are not supported by every type of JAAS login module.

Standard roles

Table 2.2, “Standard Roles for Access Control” lists and describes the standard roles that are used throughout the JMX ACLs and the command console ACLs.

Table 2.2. Standard Roles for Access Control

Roles	Description
Monitor, Operator, Maintainer	Grants read-only access to the container.
Deployer, Auditor	Grants read-write access at the appropriate level for ordinary users, who want to deploy and run applications. But blocks access to sensitive container configuration settings.
Administrator, SuperUser	Grants unrestricted access to the container.

ACL files

The standard set of ACL files are located under the **etc/auth/** directory of the AMQ installation, as follows:

etc/auth/jmx.acl[.*].cfg

JMX ACL files.

etc/auth/org.apache.karaf.command.acl.*.cfg

Command console ACL files.

Customizing role-based access control

A complete set of JMX ACL files and command console ACL files are provided by default. You are free to customize these ACLs as required to suit the requirements of your system.

You can create custom roles by editing the ACL files that are located under the **etc/auth/** directory of the JBoss Fuse installation. For more information see [Customizing the JMX ACLs](#) and [Customizing the Command Console ACLs](#)

Customizing ACLs in a fabric environment

In a standalone environment, you can assign the custom roles by editing the ACL files. This process does not work in a fabric environment as the ACL files in **/etc/auth** are over-written by the content stored in profiles. Hence, to assign custom roles in a fabric environment, you can add ACL assignments to the `acl` profile or `jboss-fuse-full` profile. For example,

```
fabric:profile-edit --pid org.apache.karaf.command.acl.fabric/container-start="Deployer, Auditor, Administrator, SuperUser, admin, MyCustomRoleForStartingContainer" acls
```

Additional properties for controlling access

The **system.properties** file under the **etc** directory provides the following additional properties for controlling access through the Karaf command console and the Fuse Management Console (Hawtio):

karaf.local.roles

Specifies the roles that apply when a user starts up the container console *locally* (for example, by running the `./bin/amq` script).

hawtio.roles

Specifies the roles that are allowed to access the container through the Fuse Management Console. This constraint is applied *in addition to* the access control defined by the JMX ACL files.

karaf.secured.command.compulsory.roles

Specifies the default roles required to invoke a Karaf console command, in case the console command is not configured explicitly by a command ACL file, **etc/auth/org.apache.karaf.command.acl.*.cfg**. A user must be configured with at least one of the roles from the list in order to invoke the command. The value is specified as a comma-separated list of roles.

2.2.2. Customizing the JMX ACLs

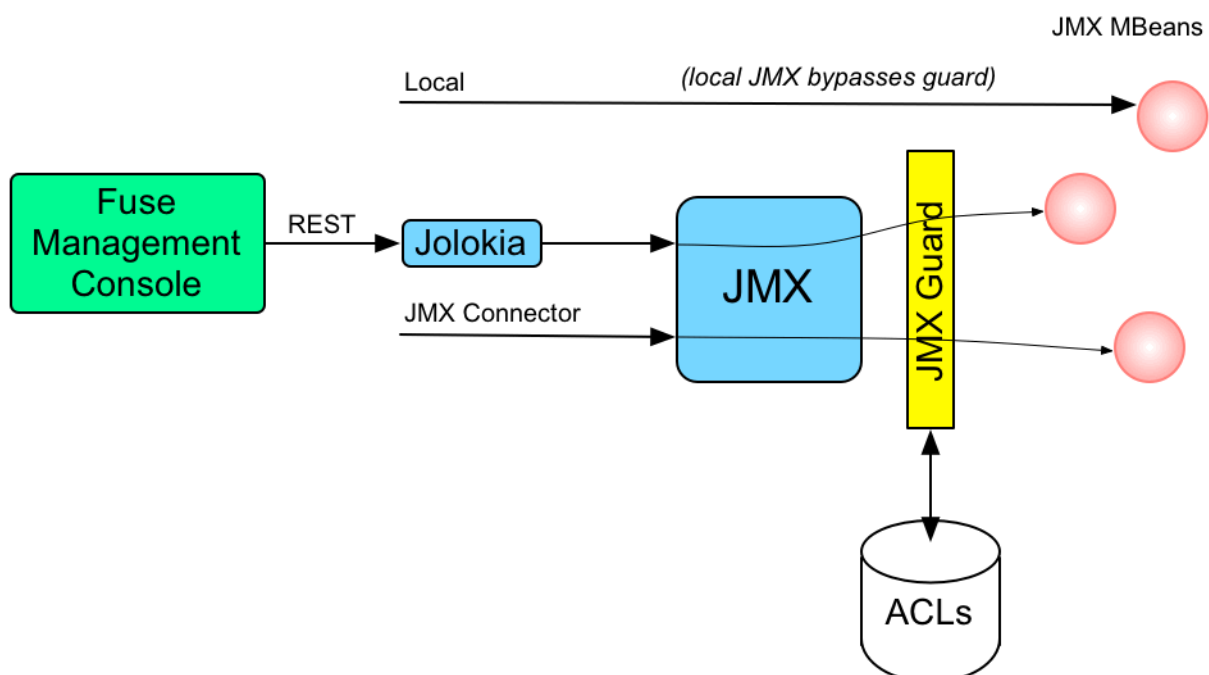
Overview

The JMX ACLs are stored in the OSGi Config Admin Service and are normally accessible as the files, **etc/auth/jmx.acl.*.cfg**. This section explains how you can customize the JMX ACLs by editing these files yourself.

Architecture

Figure 2.1, “Access Control Mechanism for JMX” shows an overview of the role-based access control mechanism for JMX connections to the AMQ container.

Figure 2.1. Access Control Mechanism for JMX



How it works

JMX access control works by inserting a *JMX Guard*, which is configured through a JVM-wide **MBeanServerBuilder** object. The Apache Karaf launching scripts have been modified to include the following setting:

```
-Djavax.management.builder.initial=org.apache.karaf.management.boot.KarafMBeanServerBuilder
```

JMX access control is now applied as follows:

1. For every non-local JMX invocation, the JVM-wide **MBeanServerBuilder** calls into an OSGi bundle that contains the JMX Guard.
2. The JMX Guard looks up the relevant ACL for the MBean the user is trying to access (where the ACLs are stored in the OSGi Config Admin service).
3. The ACL returns the list of roles that are allowed to make this particular invocation on the MBean.
4. The JMX Guard checks the list of roles against the current security subject (the user that is making the JMX invocation), to see whether the current user has any of the required roles.
5. If no matching role is found, the JMX invocation is blocked and a **java.lang.SecurityException** is raised.

Location of JMX ACL files

The JMX ACL files are located in the *InstallDir/etc/auth* directory, where the ACL file names obey the following convention:

```
etc/auth/jmx.acl[.*].cfg
```

Technically, the ACLs are mapped to OSGi persistent IDs (PIDs), matching the pattern, **jmx.acl[.*]**. It just so happens that the standalone container stores OSGi PIDs as files, *PID.cfg*, under the *etc/* directory by default.

Mapping MBeans to ACL file names

The JMX Guard applies access control to every MBean class that is accessed through JMX (including any MBeans you define in your own application code). The ACL file for a specific MBean class is derived from the MBean's Object Name, by prefixing it with **jmx.acl**. For example, given the MBean whose Object Name is given by **org.apache.activemq:type=Broker**, the corresponding PID would be:

```
jmx.acl.org.apache.activemq.Broker
```

In the case of a standalone container, the OSGi Config Admin service stores this PID data in the following file:

```
etc/auth/jmx.acl.org.apache.activemq.Broker.cfg
```

ACL file format

Each line of a JMX ACL file is an entry in the following format:

```
Pattern = Role1[,Role2][,Role3]...
```

Where **Pattern** is a pattern that matches a method invocation on an MBean, and the right-hand side of the equals sign is a comma-separated list of roles that give a user permission to make that invocation. In the simplest cases, the **Pattern** is simply a method name. For example, as in the following settings for the **org.apache.activemq.Broker** MBean (from the **jmx.acl.org.apache.activemq.Broker.cfg** file):

```
addConnector = Deployer, Auditor, Administrator, SuperUser
removeConnector = Deployer, Auditor, Administrator, SuperUser
enableStatistics = Deployer, Auditor, Administrator, SuperUser
addNetworkConnector = Deployer, Auditor, Administrator, SuperUser
```

It is also possible to use the wildcard character, *****, to match multiple method names. For example, the following entry gives permission to invoke all method names starting with **set**:

```
set* = Deployer, Auditor, Administrator, SuperUser
```

But the ACL syntax is also capable of defining much more fine-grained control of method invocations. You can define patterns to match methods invoked with specific arguments or even arguments that match a regular expression. For example, the ACL for the **org.apache.karaf.config** MBean package exploits this capability to prevent ordinary users from modifying sensitive configuration settings. The **create** method from this package is restricted, as follows:

```
create(java.lang.String)[jmx[.]acl.*] = Administrator, SuperUser
create(java.lang.String)[org[.]apache[.]karaf[.]command[.]acl.+] = Administrator, SuperUser
create(java.lang.String)[org[.]apache[.]karaf[.]service[.]acl.+] = Administrator, SuperUser
create(java.lang.String) = Deployer, Auditor, Administrator, SuperUser
```

In this case, the **Deployer** and **Auditor** roles generally have permission to invoke the **create** method, but only the **Administrator** and **SuperUser** roles have permission to invoke **create** with a PID argument matching **jmx.acl.***, **org.apache.karaf.command.acl.***, or **org.apache.karaf.service.***.

For complete details of the ACL file format, please see the comments in the **etc/auth/jmx.acl.cfg** file.

ACL file hierarchy

Because it is often impractical to provide an ACL file for every single MBean, you have the option of specifying an ACL file at the level of a Java package, which provides default settings for *all* of the MBeans in that package. For example, the **org.apache.activemq.Broker** MBean could be affected by ACL settings at *any* of the following PID levels:

```
jmx.acl.org.apache.activemq.Broker
jmx.acl.org.apache.activemq
jmx.acl.org.apache
jmx.acl.org
jmx.acl
```

Where the most specific PID (top of the list) takes precedence over the least specific PID (bottom of the list).

Root ACL definitions

The root ACL file, **jmx.acl.cfg**, is a special case, because it supplies the default ACL settings for *all* MBeans. The root ACL has the following settings by default:

```
list* = viewer, Monitor, Operator, Maintainer, Deployer, Auditor, Administrator, SuperUser
get* = viewer, Monitor, Operator, Maintainer, Deployer, Auditor, Administrator, SuperUser
is* = viewer, Monitor, Operator, Maintainer, Deployer, Auditor, Administrator, SuperUser
set* = admin, Administrator, SuperUser
* = admin, Administrator, SuperUser
```

This implies that the typical *read* method patterns (**list***, **get***, **is***) are accessible to all standard roles, but the typical *write* method patterns and other methods (**set*** and *****) are accessible only to the administrator roles, **admin**, **Administrator**, **SuperUser**.

Package ACL definitions

Many of the standard JMX ACL files provided in **etc/auth/jmx.acl[.*].cfg** apply to MBean packages. For example, the ACL for the **org.apache.camel.endpoints** MBean package is defined with the following permissions:

```
is* = Monitor, Operator, Maintainer, Deployer, Auditor, Administrator, SuperUser
get* = Monitor, Operator, Maintainer, Deployer, Auditor, Administrator, SuperUser
set* = Deployer, Auditor, Administrator, SuperUser
```

ACL for custom MBeans

If you define custom MBeans in your own application, these custom MBeans are automatically integrated with the ACL mechanism and protected by the JMX Guard when you deploy them into the container. By default, however, your MBeans are typically protected only by the default root ACL file, **jmx.acl.cfg**. If you want to define a more fine-grained ACL for your MBean, create a new ACL file under **etc/auth**, using the standard JMX ACL file naming convention.

For example, if your custom MBean class has the JMX Object Name, **org.example:type=MyMBean**, create a new ACL file under the **etc/auth** directory called:

```
jmx.acl.org.example.MyMBean.cfg
```

Dynamic configuration at run time

Because the OSGi Config Admin service is dynamic, you can change ACL settings while the system is running, and even while a particular user is logged on. Hence, if you discover a security breach while the system is running, you can immediately restrict access to certain parts of the system by editing the relevant ACL file, without having to restart the container.

2.2.3. Customizing the Command Console ACLs

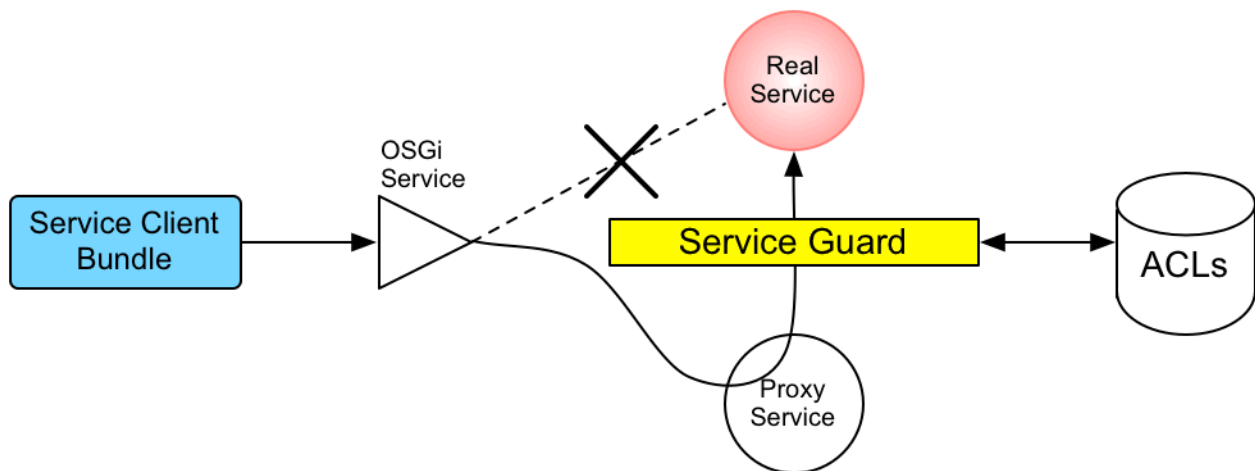
Overview

The command console ACLs are stored in the OSGi Config Admin Service and are normally accessible as the files, **etc/auth/org.apache.karaf.command.acl.*.cfg**. This section explains how you can customize the command console ACLs by editing these files yourself.

Architecture

Figure 2.2, “Access Control Mechanism for OSGi Services” shows an overview of the role-based access control mechanism for OSGi services in the AMQ container.

Figure 2.2. Access Control Mechanism for OSGi Services



How it works

The mechanism for command console access control is, in fact, based on the generic access control mechanism for OSGi services. It so happens that console commands are implemented and exposed as OSGi services. The Karaf console itself discovers the available commands through the OSGi service registry and accesses the commands as OSGi services. Hence, the access control mechanism for OSGi services can be used to control access to console commands.

The mechanism for securing OSGi services is based on OSGi Service Registry Hooks. This is an advanced OSGi feature that makes it possible to hide OSGi services from certain consumers and to replace an OSGi service with a proxy service.

When a service guard is in place for a particular OSGi service, a client invocation on the OSGi service proceeds as follows:

1. The invocation does *not* go directly to the requested OSGi service. Instead, the request is routed to a replacement proxy service, which has the same service properties as the original service (and some extra ones).
2. The service guard looks up the relevant ACL for the target OSGi service (where the ACLs are stored in the OSGi Config Admin service).
3. The ACL returns the list of roles that are allowed to make this particular method invocation on the service.
4. If no ACL is found for this command, the service guard defaults to the list of roles specified in the **karaf.secured.command.compulsory.roles** property in the **etc/system.properties** file.
5. The service guard checks the list of roles against the current security subject (the user that is making the method invocation), to see whether the current user has any of the required roles.
6. If no matching role is found, the method invocation is blocked and a **java.lang.SecurityException** is raised.
7. Alternatively, if a matching role is found, the method invocation is delegated to the original OSGi service.

Configuring default security roles

For any commands that do not have a corresponding ACL file, you specify a default list of security roles by setting the **karaf.secured.command.compulsory.roles** property in the **etc/system.properties** file (specified as a comma-separated list of roles).

Location of command console ACL files

The command console ACL files are located in the **InstallDir/etc/auth** directory, with the prefix, **org.apache.karaf.command.acl**.

Mapping command scopes to ACL file names

The command console ACL file names obey the following convention:

```
etc/auth/org.apache.karaf.command.acl.CommandScope.cfg
```

Where the **CommandScope** corresponds to the prefix for a particular group of Karaf console commands. For example, the **features:install** and **features:uninstall** commands belong to the **features** command scope, which has the corresponding ACL file, **org.apache.karaf.command.acl.features.cfg**.

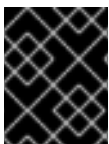
ACL file format

Each line of a command console ACL file is an entry in the following format:

```
Pattern = Role1[,Role2][,Role3]...
```

Where **Pattern** is a pattern that matches a Karaf console command from the current command scope, and the right-hand side of the equals sign is a comma-separated list of roles that give a user permission to make that invocation. In the simplest cases, the **Pattern** is simply an unscoped command name. For example, the **org.apache.karaf.command.acl.features.cfg** ACL file includes the following rules for the **features** commands:

```
list = Monitor, Operator, Maintainer, Deployer, Auditor, Administrator, SuperUser
listRepositories = Monitor, Operator, Maintainer, Deployer, Auditor, Administrator, SuperUser
listUrl = Monitor, Operator, Maintainer, Deployer, Auditor, Administrator, SuperUser
info = Monitor, Operator, Maintainer, Deployer, Auditor, Administrator, SuperUser
install = Administrator, SuperUser
uninstall = Administrator, SuperUser
```



IMPORTANT

If no match is found for a specific command name, it is assumed that no role is required for this command and it can be invoked by any user.

You can also define patterns to match commands invoked with specific arguments or even arguments that match a regular expression. For example, the **org.apache.karaf.command.acl.osgi.cfg** ACL file exploits this capability to prevent ordinary users from invoking the **osgi:start** and **osgi:stop** commands with the **-f** (force) flag (which must be specified to manage system bundles). This restriction is coded as follows in the ACL file:

```
start[/.*[-][f].*/] = Administrator, SuperUser
start = Deployer, Auditor, Administrator, SuperUser
```

```
stop[/. *[-][f]. */] = Administrator, SuperUser
stop = Deployer, Auditor, Administrator, SuperUser
```

In this case, the **Deployer** and **Auditor** roles generally have permission to invoke the **osgi:start** and **osgi:stop** commands, but only the **Administrator** and **SuperUser** roles have permission to invoke these commands with the force option, **-f**.

For complete details of the ACL file format, please see the comments in the **etc/auth/org.apache.karaf.command.acl.osgi.cfg** file.

Dynamic configuration at run time

The command console ACL settings are fully dynamic, which means you can change the ACL settings while the system is running and the changes will take effect within a few seconds, even for users that are already logged on.

2.2.4. Defining ACLs for OSGi Services

Overview

It is possible to define a custom ACL for any OSGi service (whether system level or application level). By default, OSGi services do not have access control enabled (with the exception of the OSGi services that expose Karaf console commands, which are pre-configured with command console ACL files). This section explains how to define a custom ACL for an OSGi service and how to invoke methods on that service using a specified role.

ACL file format

An OSGi service ACL file has one special entry, which identifies the OSGi service to which this ACL applies, as follows:

```
service.guard = (objectClass=InterfaceName)
```

Where the value of **service.guard** is an LDAP search filter that is applied to the registry of OSGi service properties in order to pick out the matching OSGi service. The simplest type of filter, **(objectClass=*InterfaceName*)**, picks out an OSGi service with the specified Java interface name, ***InterfaceName***.

The remaining entries in the ACL file are of the following form:

```
Pattern = Role1[,Role2][,Role3]...
```

Where ***Pattern*** is a pattern that matches a service method, and the right-hand side of the equals sign is a comma-separated list of roles that give a user permission to make that invocation. The syntax of these entries is essentially the same as the entries in a JMX ACL file—see [the section called “ACL file format”](#).

How to define an ACL for a custom OSGi service

To define an ACL for a custom OSGi service, perform the following steps:

1. It is customary to define an OSGi service using a Java interface (you could use a regular Java class, but this is not recommended). For example, consider the Java interface, **MyService**, which we intend to expose as an OSGi service:

-

```
package org.example;
```

```
public interface MyService {
    void doit(String s);
}
```

- To expose the Java interface as an OSGi service, you would typically add a **service** element to an OSGi Blueprint XML file (where the Blueprint XML file is typically stored under the **src/main/resources/OSGI-INF/blueprint** directory in a Maven project). For example, assuming that **MyServiceImpl** is the class that implements the **MyService** interface, you could expose the **MyService** OSGi service as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  default-activation="lazy">

  <bean id="myserviceimpl" class="org.example.MyServiceImpl"/>

  <service id="myservice" ref="myserviceimpl" interface="org.example.MyService"/>

</blueprint>
```

- To define an ACL for the the OSGi service, you must create an OSGi Config Admin PID with the prefix, **org.apache.karaf.service.acl**.

For example, in the case of a standalone container (where the OSGi Config Admin PIDs are stored as **.cfg** files under the **etc/auth/** directory), you can create the following ACL file for the **MyService** OSGi service:

```
etc/auth/org.apache.karaf.service.acl.myservice.cfg
```



NOTE

It does not matter exactly how you name this file, as long as it starts with the required prefix, **org.apache.karaf.service.acl**. The corresponding OSGi service for this ACL file is actually specified by a property setting in this file (as you will see in the next step).

- Specify the contents of the ACL file in a format like the following:

```
service.guard = (objectClass=InterfaceName)
Pattern = Role1[,Role2][,Role3]...
```

The **service.guard** setting specifies the **InterfaceName** of the OSGi service (using the syntax of an LDAP search filter, which is applied to the OSGi service properties). The other entries in the ACL file consist of a method **Pattern**, which associates a matching method to the specified roles. For example, you could define a simple ACL for the **MyService** OSGi service with the following settings in the **org.apache.karaf.service.acl.myservice.cfg** file:

```
service.guard = (objectClass=org.example.MyService)
doit = Deployer, Auditor, Administrator, SuperUser
```

- Finally, in order to enable the ACL for this OSGi service, you must edit the **karaf.secured.services** property in the **etc/system.properties** file. The value of the **karaf.secured.services** property has the syntax of an LDAP search filter (which gets applied to the OSGi service properties). In general, to enable ACLs for an OSGi service, **ServiceInterface**, you must modify this property as follows:

```
karaf.secured.services=((objectClass=ServiceInterface)(...ExistingPropValue...))
```

For example, to enable the **MyService** OSGi service:

```
karaf.secured.services=((objectClass=org.example.MyService)&(osgi.command.scope=*)
(osgi.command.function=*)))
```

CAUTION

The initial value of the **karaf.secured.services** property has the settings to enable the command console ACLs. If you delete or corrupt these entries, the command console ACLs might stop working.

How to invoke an OSGi service secured with RBAC

If you are writing Java code to invoke methods on a custom OSGi service (that is, implementing a client of the OSGi service), you must use the Java security API to specify the role you are using to invoke the service. For example, to invoke the **MyService** OSGi service using the **Deployer** role, you could use code like the following:

```
// Java
import javax.security.auth.Subject;
import org.apache.karaf.jaas.boot.principal.RolePrincipal;
// ...
Subject s = new Subject();
s.getPrincipals().add(new RolePrincipal("Deployer"));
Subject.doAs(s, new PrivilegedAction() {
    public Object run() {
        svc.doit("foo"); // invoke the service
    }
})
```

NOTE

This example uses the Karaf role type, **org.apache.karaf.jaas.boot.principal.RolePrincipal**. If necessary, you could use your own custom role class instead, but in that case you would have to specify your roles using the syntax **className:roleName** in the OSGi service's ACL file.

How to discover the roles required by an OSGi service

When you are writing code against an OSGi service secured by an ACL, it can sometimes be useful to check what roles are allowed to invoke the service. For this purpose, the proxy service exports an additional OSGi property, **org.apache.karaf.service.guard.roles**. The value of this property is a **java.util.Collection** object, which contains a list of all the roles that could possibly invoke a method on that service.

2.3. USING ENCRYPTED PROPERTY PLACEHOLDERS

Overview

When securing a container it is undesirable to use plain text passwords in configuration files. They create easy to target security holes. One way to avoid this problem is to use encrypted property placeholders when ever possible. This feature is supported both in Blueprint XML files and in Spring XML files.

How to use encrypted property placeholders

To use encrypted property placeholders in a Blueprint XML file or in a Spring XML file, perform the following steps:

1. [Download and install Jasypt](#), to gain access to the Jasypt **listAlgorithms.sh**, **encrypt.sh** and **decrypt.sh** command-line tools.



NOTE

When installing the Jasypt command-line tools, don't forget to enable execute permissions on the script files, by running **chmod u+x ScriptName.sh**.

2. Choose a master password and an encryption algorithm. To discover which algorithms are supported in your current Java environment, run the **listAlgorithms.sh** Jasypt command-line tool, as follows:

```
./listAlgorithms.sh
DIGEST ALGORITHMS: [MD2, MD5, SHA, SHA-256, SHA-384, SHA-512]

PBE ALGORITHMS: [PBEWITHMD5ANDDES, PBEWITHMD5ANDTRIPLEDES,
PBEWITHSHA1ANDDESEDE, PBEWITHSHA1ANDRC2_40]
```

On Windows platforms, the script is **listAlgorithms.bat**. AMQ uses **PBEWithMD5AndDES** by default.

3. Use the Jasypt **encrypt** command-line tool to encrypt your sensitive configuration values (for example, passwords for use in configuration files). For example, the following command encrypts the **PlaintextVal** value, using the specified algorithm and master password **MasterPass**:

```
./encrypt.sh input="PlaintextVal" algorithm=PBEWithMD5AndDES password=MasterPass
```

4. Create a properties file with encrypted values. For example, suppose you wanted to store some LDAP credentials. You could create a file, **etc/ldap.properties**, with the following contents:

Example 2.6. Property File with an Encrypted Property

```
#ldap.properties
ldap.password=ENC(amlsvdqno9iSwnd7kAILYQ==)
ldap.url=ldap://192.168.1.74:10389
```

The encrypted property values (as generated in the previous step) are identified by wrapping in the **ENC()** function.

5. (*Blueprint XML only*) Add the requisite namespaces to your Blueprint XML file:

- Aries extensions—<http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0>
- Apache Karaf Jasypt—<http://karaf.apache.org/xmlns/jasypt/v1.0.0>

Example 2.7, “Encrypted Property Namespaces” shows a Blueprint file with the requisite namespaces.

Example 2.7. Encrypted Property Namespaces

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0"
  xmlns:enc="http://karaf.apache.org/xmlns/jasypt/v1.0.0">
...
</blueprint>
```

6. Configure the location of the properties file for the property placeholder and configure the Jasypt encryption algorithm .

- **Blueprint XML**

Example 2.8, “Jasypt Blueprint Configuration” shows how to configure the **ext:property-placeholder** element to read properties from the **etc/ldap.properties** file. The **enc:property-placeholder** element configures Jasypt to use the **PBEWithMD5AndDES** encryption algorithm and to read the master password from the **JASYPT_ENCRYPTION_PASSWORD** environment variable.

Example 2.8. Jasypt Blueprint Configuration

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0"
  xmlns:enc="http://karaf.apache.org/xmlns/jasypt/v1.0.0">

  <ext:property-placeholder>
    <location>file:etc/ldap.properties</location>
  </ext:property-placeholder>

  <enc:property-placeholder>
    <enc:encryptor class="org.jasypt.encryption.pbe.StandardPBEStrngEncryptor">
      <property name="config">
        <bean class="org.jasypt.encryption.pbe.config.EnvironmentStringPBEConfig">
          <property name="algorithm" value="PBEWithMD5AndDES" />
          <property name="passwordEnvName"
value="JASYPT_ENCRYPTION_PASSWORD" />
        </bean>
      </property>
    </enc:encryptor>
  </enc:property-placeholder>

...
</blueprint>
```

- Spring XML

Example 2.9, “Jasypt Spring Configuration” shows how to configure Jasypt to use the **PBEWithMD5AndDES** encryption algorithm and to read the master password from the **JASYPT_ENCRYPTION_PASSWORD** environment variable.

The **EncryptablePropertyPlaceholderConfigurer** bean is configured to read properties from the **etc/ldap.properties** file and to read properties from the **io.fabric8.mq.fabric.ConfigurationProperties** class (which defines the **karaf.base** property, for example).

Example 2.9. Jasypt Spring Configuration

```
<bean id="environmentVariablesConfiguration"
class="org.jasypt.encryption.pbe.config.EnvironmentStringPBEConfig">
  <property name="algorithm" value="PBEWithMD5AndDES" />
  <property name="passwordEnvName"
value="JASYPT_ENCRYPTION_PASSWORD" />
</bean>

<bean id="configurationEncryptor"
class="org.jasypt.encryption.pbe.StandardPBEStringEncryptor">
  <property name="config" ref="environmentVariablesConfiguration" />
</bean>

<bean id="propertyConfigurer"
class="org.jasypt.spring31.properties.EncryptablePropertyPlaceholderConfigurer">
  <constructor-arg ref="configurationEncryptor" />
  <property name="location" value="file:${karaf.base}/etc/ldap.properties"/>
  <property name="properties">
    <bean class="io.fabric8.mq.fabric.ConfigurationProperties"/>
  </property>
</bean>
```

7. Use the placeholders in your configuration file. The placeholders you use for encrypted properties are the same as you use for regular properties. Use the syntax **`${prop.name}`**.
8. Make sure that the **jasypt-encryption** feature is installed in the container. If necessary, install the **jasypt-encryption** feature with the following console command:

```
JBossFuse:karaf@root> features:install jasypt-encryption
```

9. Shut down the container, by entering the following command:

```
JBossFuse:karaf@root> shutdown
```

10. Carefully restart the container and deploy your secure application, as follows:
 1. Open a command window (first command window) and enter the following commands to start the AMQ container in the background:

```
export JASYPT_ENCRYPTION_PASSWORD="your super secret master pass phrase"
./bin/start
```

2. Open a second command window and start the client utility, to connect to the container running in the background:

```
./bin/client -u Username -p Password
```

Where ***Username*** and ***Password*** are valid JAAS user credentials for logging on to the container console.

3. In the second command window, use the console to install your secure application that uses encrypted property placeholders. Check that the application has launched successfully (for example, using the **osgi:list** command to check its status).
4. After the secure application has started up, go back to the first command window and unset the **JASYPT_ENCRYPTION_PASSWORD** environment variable.



IMPORTANT

Unsetting the **JASYPT_ENCRYPTION_PASSWORD** environment variable ensures there will be minimum risk of exposing the master password. The Jasypt library retains the master password in encrypted form in memory.

Blueprint XML example

Example 2.10, “[Jasypt Example in Blueprint XML](#)” shows an example of an LDAP JAAS realm configured in Blueprint XML, using Jasypt encrypted property placeholders.

Example 2.10. Jasypt Example in Blueprint XML

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0"
  xmlns:enc="http://karaf.apache.org/xmlns/jasypt/v1.0.0">

  <ext:property-placeholder>
    <location>file:etc/ldap.properties</location>
  </ext:property-placeholder>

  <enc:property-placeholder>
    <enc:encryptor class="org.jasypt.encryption.pbe.StandardPBEStrngEncryptor">
      <property name="config">
        <bean class="org.jasypt.encryption.pbe.config.EnvironmentStringPBESConfig">
          <property name="algorithm" value="PBEWithMD5AndDES" />
          <property name="passwordEnvName" value="JASYPT_ENCRYPTION_PASSWORD" />
        </bean>
      </property>
    </enc:encryptor>
  </enc:property-placeholder>

  <jaas:config name="karaf" rank="200">
    <jaas:module className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule"
      flags="required">
      initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
```

```

debug=true
connectionURL=${ldap.url}
connectionUsername=cn=mqbroker,ou=Services,ou=system,dc=jbossfuse,dc=com
connectionPassword=${ldap.password}
connectionProtocol=
authentication=simple
userRoleName=cn
userBase = ou=User,ou=ActiveMQ,ou=system,dc=jbossfuse,dc=com
userSearchMatching=(uid={0})
userSearchSubtree=true
roleBase = ou=Group,ou=ActiveMQ,ou=system,dc=jbossfuse,dc=com
roleName=cn
roleSearchMatching= (member:=uid={1})
roleSearchSubtree=true
</jaas:module>
</jaas:config>

</blueprint>

```

The `${ldap.password}` placeholder is replaced with the decrypted value of the `ldap.password` property from the `etc/ldap.properties` properties file.

2.4. ENABLING REMOTE JMX SSL

Overview

Red Hat JBoss Fuse provides a JMX port that allows remote monitoring and management of Fuse containers using MBeans. By default, however, the credentials that you send over the JMX connection are unencrypted and vulnerable to snooping. To encrypt the JMX connection and protect against password snooping, you need to secure JMX communications by configuring JMX over SSL.

To configure JMX over SSL, perform the following steps:

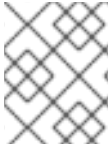
1. [Create the jbossweb.keystore file](#)
2. [Create and deploy the keystore.xml file](#)
3. [Add the required properties to org.apache.karaf.management.cfg](#)
4. Restart the container

After you have configured JMX over SSL access, you should test the connection.



WARNING

If you are planning to enable SSL/TLS security, you must ensure that you explicitly disable the SSLv3 protocol, in order to safeguard against the [Poodle vulnerability \(CVE-2014-3566\)](#). For more details, see [Disabling SSLv3 in JBoss Fuse 6.x and JBoss A-MQ 6.x](#).

**NOTE**

If you configure JMX over SSL while Red Hat JBoss Fuse is running, you will need to restart it.

Prerequisites

If you haven't already done so, you need to:

- Set your **JAVA_HOME** environment variable
- Configure a JBoss Fuse user with the **Administrator** role

Edit the `<installDir>/jboss-fuse-6.3.0.redhat-187/etc/users.properties` file and add the following entry, on a single line:

```
admin=YourPassword,Administrator
```

This creates a new user with username, **admin**, password, **YourPassword**, and the **Administrator** role.

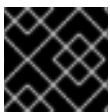
Create the jbossweb.keystore file

Open a command prompt and make sure you are in the **etc/** directory of your AMQ installation:

```
cd <installDir>/jboss-fuse-6.3.0.redhat-187/etc
```

At the command line, using a **-dname** value (Distinguished Name) appropriate for your application, type this command:

```
$JAVA_HOME/bin/keytool -genkey -v -alias jbossalias -keyalg RSA -keysize 1024 -keystore jbossweb.keystore -validity 3650 -keypass JbossPassword -storepass JbossPassword -dname "CN=127.0.0.1, OU=RedHat Software Unit, O=RedHat, L=Boston, S=Mass, C=USA"
```

**IMPORTANT**

Type the entire command on a single command line.

The command returns output that looks like this:

```
Generating 1,024 bit RSA key pair and self-signed certificate (SHA256withRSA) with a validity of
3,650 days
for: CN=127.0.0.1, OU=RedHat Software Unit, O=RedHat, L=Boston, ST=Mass, C=USA
New certificate (self-signed):
[
[
Version: V3
Subject: CN=127.0.0.1, OU=RedHat Software Unit, O=RedHat, L=Boston, ST=Mass, C=USA
Signature Algorithm: SHA256withRSA, OID = 1.2.840.113549.1.1.11

Key: Sun RSA public key, 1024 bits
modulus:
1123086025790567043604962990501918169461098372864273201795342440080393808
```

```

1594100776075008647459910991413806372800722947670166407814901754459100720279046
3944621813738177324031064260382659483193826177448762030437669318391072619867218
036972335210839062722456085328301058362052369248473659880488338711351959835357
public exponent: 65537
Validity: [From: Thu Jun 05 12:19:52 EDT 2014,
          To: Sun Jun 02 12:19:52 EDT 2024]
Issuer: CN=127.0.0.1, OU=RedHat Software Unit, O=RedHat, L=Boston, ST=Mass, C=USA
SerialNumber: [ 4666e4e6]

Certificate Extensions: 1
[1]: ObjectID: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: AC 44 A5 F2 E6 2F B2 5A 5F 88 FE 69 60 B4 27 7D .D.../Z_..i'!.
0010: B9 81 23 9C ..#.
]
]
]
Algorithm: [SHA256withRSA]
Signature:
0000: 01 1D 95 C0 F2 03 B0 FD CF 3A 1A 14 F5 2E 04 E5 .....:.....
0010: DD 18 DD 0E 24 60 00 54 35 AE FE 36 7B 38 69 4C ....$.T5..6.8iL
0020: 1E 85 0A AF AE 24 1B 40 62 C9 F4 E5 A9 02 CD D3 ....$.@b.....
0030: 91 57 60 F6 EF D6 A4 84 56 BA 5D 21 11 F7 EA 09 .W`.....V.}!....
0040: 73 D5 6B 48 4A A9 09 93 8C 05 58 91 6C D0 53 81 s.kHJ.....X.I.S.
0050: 39 D8 29 59 73 C4 61 BE 99 13 12 89 00 1C F8 38 9.)Ys.a.....8
0060: E2 BF D5 3C 87 F6 3F FA E1 75 69 DF 37 8E 37 B5 ...<..?..ui.7.7.
0070: B7 8D 10 CC 9E 70 E8 6D C2 1A 90 FF 3C 91 84 50 .....p.m....<..P
]
[Storing jbossweb.keystore]

```

Check whether `<installDir>/jboss-fuse-6.3.0.redhat-187/etc` now contains the file `jbossweb.keystore`.

Create and deploy the keystore.xml file

1. Using your favorite xml editor, create and save the `keystore.xml` file in the `<installDir>/jboss-fuse-6.3.0.redhat-187/etc` directory.
2. Include this text in the file:

```

<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0">
  <jaas:keystore name="sample_keystore"
    rank="1"
    path="file:/etc/jbossweb.keystore"
    keystorePassword="JbossPassword"
    keyPasswords="jbossalias=JbossPassword" />
</blueprint>

```

3. Deploy the `keystore.xml` file to the container, by copying it into the `<installDir>/jboss-fuse-6.3.0.redhat-187/deploy` directory (the hot deploy directory).

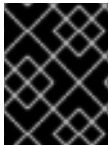
**NOTE**

Subsequently, if you need to undeploy the **keystore.xml** file, you can do so by deleting the **keystore.xml** file from the **deploy/** directory *while the Karaf container is running*.

Add the required properties to org.apache.karaf.management.cfg

Edit the `<installDir>/jboss-fuse-6.3.0.redhat-187/etc/org.apache.karaf.management.cfg` file to include these properties at the end of the file:

```
secured = true
secureProtocol = TLSv1
keyAlias = jbossalias
keyStore = sample_keystore
trustStore = sample_keystore
```

**IMPORTANT**

You must set **secureProtocol** to **TLSv1**, in order to protect against the [Poodle vulnerability \(CVE-2014-3566\)](#)

Restart the AMQ container

You must restart the AMQ container for the new JMX SSL/TLS settings to take effect.

Testing the Secure JMX connection

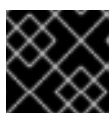
1. Open a command prompt and make sure you are in the **etc/** directory of your AMQ installation:

```
cd <installDir>/jboss-fuse-6.3.0.redhat-187/etc
```

2. Open a terminal, and start up JConsole by entering this command:

```
jconsole -J-Djavax.net.debug=ssl -J-Djavax.net.ssl.trustStore=jbossweb.keystore -J-Djavax.net.ssl.trustStoreType=JKS -J-Djavax.net.ssl.trustStorePassword=JbossPassword
```

Where the **-J-Djavax.net.ssl.trustStore** option specifies the location of the **jbossweb.keystore** file (make sure this location is specified correctly, or the SSL/TLS handshake will fail). The **-J-Djavax.net.debug=ssl** setting enables logging of SSL/TLS handshake messages, so you can verify that SSL/TLS has been successfully enabled.

**IMPORTANT**

Type the entire command on the same command line.

3. When JConsole opens, select the option **Remote Process** in the **New Connection** wizard.
4. Under the **Remote Process** option, enter the following value for the **service:jmx:<protocol>:<sap>** connection URL:

```
service:jmx:rmi://localhost:44444/jndi/rmi://localhost:1099/karaf-root
```


And fill in the **Username**, and **Password** fields with valid JAAS credentials (as set in the **etc/users.properties** file):

Username: admin
Password: ***YourPassword***

CHAPTER 3. SECURING THE JETTY HTTP SERVER

Abstract

You can configure the built-in Jetty HTTP server to use SSL/TLS security by adding the relevant configuration properties to the **etc/org.ops4j.pax.web.cfg** configuration file. In particular, you can add SSL/TLS security to the Fuse Management Console in this way.

JETTY SERVER

The AMQ container is pre-configured with a Jetty server, which acts as a general-purpose HTTP server and HTTP servlet container. Through a single HTTP port (by default, **http://Host:8181**), the Jetty container can host multiple services, for example:

- Fuse Management Console (by default, **http://Host:8181/hawtio**)
- Apache CXF Web services endpoints (by default, **http://Host:8181/cxf**, if the host and port are left unspecified in the endpoint configuration)
- Some Apache Camel endpoints

If you use the default Jetty server for all of your HTTP endpoints, you can conveniently add SSL/TLS security to these HTTP endpoints by following the steps described here.

CREATE X.509 CERTIFICATE AND PRIVATE KEY

Before you can enable SSL, you must create an X.509 certificate and private key for the Web console. The certificate and private key must be in Java keystore format. For details of how to create a signed certificate and private key, see [Appendix A, Managing Certificates](#).

ENABLING SSL/TLS FOR JETTY IN A STANDALONE CONTAINER

To enable SSL/TLS for Jetty in a standalone (non-Fabric) Karaf container:

1. Open **etc/org.ops4j.pax.web.cfg** in a text editor.
2. Replace the original content of the **etc/org.ops4j.pax.web.cfg** file with the following settings:

```
# Configures the SMX Web Console to use SSL
org.ops4j.pax.web.config.file=etc/jetty.xml

org.osgi.service.http.enabled=false
org.osgi.service.http.port=8181

org.ops4j.pax.web.session.cookie.httpOnly=true

org.osgi.service.http.secure.enabled=true
org.osgi.service.http.port.secure=8443
org.ops4j.pax.web.ssl.keystore=etc/alice.ks
org.ops4j.pax.web.ssl.password=alicepass
org.ops4j.pax.web.ssl.keypassword=alicepass
```

Where the new settings disable the existing insecure HTTP port (on 8181) and enable a new secure HTTPS port (on 8443).

3. Customize the SSL/TLS settings in **etc/org.ops4j.pax.web.cfg** as follows:

org.osgi.service.http.port.secure

Specifies the TCP port number of the secure HTTPS port.

org.ops4j.pax.web.ssl.keystore

The location of the Java keystore file on the file system. Relative paths are resolved relative to the **KARAF_HOME** environment variable (by default, the install directory).

org.ops4j.pax.web.ssl.password

The *store password* that unlocks the Java keystore file.

org.ops4j.pax.web.ssl.keypassword

The *key password* that decrypts the private key stored in the keystore (usually the same as the store password).

4. Restart the AMQ container, in order for the configuration changes to take effect.

CUSTOMIZING ALLOWED TLS PROTOCOLS AND CIPHER SUITES

You can customize the allowed TLS protocols and cipher suites by setting the following properties in the **etc/org.ops4j.pax.web.cfg** file:

org.ops4j.pax.web.ssl.protocols.included

Specifies a list of allowed TLS/SSL protocols.

org.ops4j.pax.web.ssl.protocols.excluded

Specifies a list of disallowed TLS/SSL protocols.

org.ops4j.pax.web.ssl.ciphersuites.included

Specifies a list of allowed TLS/SSL cipher suites.

org.ops4j.pax.web.ssl.ciphersuites.excluded

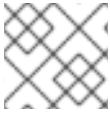
Specifies a list of disallowed TLS/SSL cipher suites.

For full details of the available protocols and cipher suites, consult the appropriate JVM documentation and security provider documentation. For example, for Java 7, see [Java Cryptography Architecture Oracle Providers Documentation for Java Platform Standard Edition 7](#).

CONNECT TO THE SECURE CONSOLE

After configuring SSL security for the Jetty server in the Pax Web configuration file, you should be able to open the Fuse Management Console by browsing to the following URL:

https://Host:8443/hawtio

**NOTE**

Remember to type the **https:** scheme, instead of **http:**, in this URL.

Initially, the browser will warn you that you are using an untrusted certificate. Skip this warning and you will be presented with the login screen for the Fuse Management Console.

ADVANCED JETTY SECURITY CONFIGURATION

In order to have more control over the Jetty security settings, you can enable Jetty security by modifying the configuration settings in the **etc/jetty.xml** file. This approach gives you access to the full Jetty security API:

1. Open **etc/org.ops4j.pax.web.cfg** in a text editor.
2. Disable the insecure HTTP port by adding the `org.osgi.service.http.enabled` and setting it to **false**; and enable the secure HTTPS port by adding the `org.osgi.service.http.secure.enabled` and setting it to **true**. Change the value of **org.ops4j.pax.web.config.file** to reference the file, **etc/jetty-ssl.xml** (which you will create in the next step).

The **etc/org.ops4j.pax.web.cfg** file should now have the following contents:

```
# Configures the SMX Web Console to use SSL
org.ops4j.pax.web.config.file=etc/jetty-ssl.xml

org.osgi.service.http.enabled=false
org.osgi.service.http.port=8181

org.ops4j.pax.web.session.cookie.httpOnly=true

org.osgi.service.http.secure.enabled=true
```

3. Create a new file, **etc/jetty-ssl.xml**, with the following contents:

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN"
"http://www.eclipse.org/jetty/configure_9_0.dtd">

<Configure id="Server" class="org.eclipse.jetty.server.Server">

  <!-- ===== -->
  <!-- Set connectors -->
  <!-- ===== -->
  <!-- One of each type! -->
  <!-- ===== -->

  <!-- Use this connector for many frequently idle connections
       and for threadless continuations. -->
  <New id="httpConfig"
    class="org.eclipse.jetty.server.HttpConfiguration">
    <Set name="secureScheme">https</Set>
    <Set name="securePort">
      <Property name="jetty.secure.port" default="8443" />
    </Set>
    <Set name="outputBufferSize">32768</Set>
```

```

<Set name="requestHeaderSize">8192</Set>
<Set name="responseHeaderSize">8192</Set>
<Set name="sendServerVersion">true</Set>
<Set name="sendDateHeader">false</Set>
<Set name="headerCacheSize">512</Set>
</New>

<!-- ===== -->
<!-- Configure Authentication Realms -->
<!-- Realms may be configured for the entire server here, or -->
<!-- they can be configured for a specific web app in a context -->
<!-- configuration (see $(jetty.home)/contexts/test.xml for an -->
<!-- example). -->
<!-- ===== -->
<Call name="addBean">
  <Arg>
    <New class="org.eclipse.jetty.jaas.JAASLoginService">
      <Set name="name">karaf</Set>
      <Set name="loginModuleName">karaf</Set>
      <Set name="roleClassNames">
        <Array type="java.lang.String">
          <Item>
            org.apache.karaf.jaas.boot.principal.RolePrincipal
          </Item>
        </Array>
      </Set>
    </New>
  </Arg>
</Call>

<New id="sslHttpConfig"
  class="org.eclipse.jetty.server.HttpConfiguration">
  <Arg><Ref refid="httpConfig"/></Arg>
  <Call name="addCustomizer">
    <Arg>
      <New class="org.eclipse.jetty.server.SecureRequestCustomizer"/>
    </Arg>
  </Call>
</New>

<New id="sslContextFactory"
  class="org.eclipse.jetty.util.ssl.SslContextFactory">
  <Set name="KeyStorePath">
    /home/jdoe/Programs/JBossFuse/jboss-fuse-6.3.0.redhat-187/etc/alice.ks
  </Set>
  <Set name="KeyStorePassword">alicepass</Set>
  <Set name="KeyManagerPassword">alicepass</Set>
  <!--Set name="TrustStorePath">
    <Property name="jetty.base" default="." />
    <Property name="jetty.truststore"
      default="quickstarts/switchyard/demos/policy-security-basic/connector.jks"/>
  </Set>
  <Set name="TrustStorePassword">
    <Property name="jetty.truststore.password" default="changeit"/>
  </Set-->

```

```

<Set name="EndpointIdentificationAlgorithm"></Set>
<Set name="NeedClientAuth">
  <Property name="jetty.ssl.needClientAuth" default="false"/>
</Set>
<Set name="WantClientAuth">
  <Property name="jetty.ssl.wantClientAuth" default="false"/>
</Set>
<!-- Disable SSLv3 to protect against POODLE bug -->
<Set name="ExcludeProtocols">
  <Array type="java.lang.String">
    <Item>SSLv3</Item>
  </Array>
</Set>
<Set name="ExcludeCipherSuites">
  <Array type="String">
    <Item>SSL_RSA_WITH_DES_CBC_SHA</Item>
    <Item>SSL_DHE_RSA_WITH_DES_CBC_SHA</Item>
    <Item>SSL_DHE_DSS_WITH_DES_CBC_SHA</Item>
    <Item>SSL_RSA_EXPORT_WITH_RC4_40_MD5</Item>
    <Item>SSL_RSA_EXPORT_WITH_DES40_CBC_SHA</Item>
    <Item>SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA</Item>
    <Item>SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA</Item>
  </Array>
</Set>
</New>

<Call id="httpsConnector" name="addConnector">
  <Arg>
    <New class="org.eclipse.jetty.server.ServerConnector">
      <Arg name="server"><Ref refid="Server" /></Arg>
      <Arg name="acceptors" type="int">
        <Property name="ssl.acceptors" default="-1"/>
      </Arg>
      <Arg name="selectors" type="int">
        <Property name="ssl.selectors" default="-1"/>
      </Arg>
      <Arg name="factories">
        <Array type="org.eclipse.jetty.server.ConnectionFactory">
          <Item>
            <New class="org.eclipse.jetty.server.SslConnectionFactory">
              <Arg name="next">http/1.1</Arg>
              <Arg name="sslContextFactory">
                <Ref refid="sslContextFactory"/>
              </Arg>
            </New>
          </Item>
          <Item>
            <New class="org.eclipse.jetty.server.HttpConnectionFactory">
              <Arg name="config"><Ref refid="sslHttpConfig"/></Arg>
            </New>
          </Item>
        </Array>
      </Arg>
    </New>
  </Arg>
  <Set name="name">0.0.0.0:8443</Set>
  <Set name="host"><Property name="jetty.host" /></Set>
  <Set name="port">

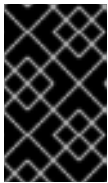
```

```

    <Property name="https.port" default="8443" />
  </Set>
  <Set name="idleTimeout">
    <Property name="https.timeout" default="30000"/>
  </Set>
  <Set name="soLingerTime">
    <Property name="https.soLingerTime" default="-1"/>
  </Set>
  <Set name="acceptorPriorityDelta">
    <Property name="ssl.acceptorPriorityDelta" default="0"/>
  </Set>
  <Set name="selectorPriorityDelta">
    <Property name="ssl.selectorPriorityDelta" default="0"/>
  </Set>
  <Set name="acceptQueueSize">
    <Property name="https.acceptQueueSize" default="0"/>
  </Set>
</New>
</Arg>
</Call>

</Configure>

```



IMPORTANT

The preceding configuration explicitly disables the SSLv3 protocol, in order to safeguard against the [Poodle vulnerability \(CVE-2014-3566\)](#). For more details, see [Disabling SSLv3 in JBoss Fuse 6.x and JBoss A-MQ 6.x](#).

4. (Optional) If you prefer, you can use a system property to help you specify the location of the Java keystore file. For example, instead of setting the **KeyStorePath** property explicitly (in the preceding **etc/jetty-ssl.xml** configuration):

```
<Set name="KeyStorePath">/home/jdoe/Documents/jetty.ks</Set>
```

You could use the **karaf.home** system property to specify the location of the keystore file relative to the AMQ install directory:

```
<Set name="KeyStorePath">
  <SystemProperty name="karaf.home"/>/etc/jetty.ks
</Set>
```

5. Customize the properties of the **SslContextFactory** instance defined in the **etc/jetty-ssl.xml** file, as follows:

KeyStorePath

The location of the Java keystore file on the file system. Relative paths are resolved relative to the **KARAF_HOME** environment variable (by default, the install directory).

KeyStorePassword

The *store password* that unlocks the Java keystore file.

KeyManagerPassword

The *key password* that decrypts the private key stored in the keystore (usually the same as the store password).

- Restart the AMQ container, in order for the configuration changes to take effect.

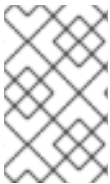


NOTE

The Apache Karaf container does *not* automatically detect changes in the **etc/jetty-ssl.xml** file. Hence, if you make subsequent edits to the **etc/jetty-ssl.xml** file, you must also update the **etc/org.ops4j.pax.web.cfg** file (by making a trivial edit or using the UNIX **touch** command), in order to force Apache Karaf to reload the **etc/jetty-ssl.xml** file.

ENABLING SSL/TLS FOR JETTY IN A FABRIC

Securing Jetty in a Fabric is slightly more complicated than securing Jetty in a standalone Karaf container, because each container must also be configured as a secure client of the Jetty HTTP server. For example, whenever a new container is provisioned in a Fabric, it downloads artifacts by connecting to the Maven proxy through the Jetty HTTPS port on the root container. Hence, each container in the Fabric must be configured to trust the HTTPS connection to the root container (by configuring a trust store).



NOTE

The procedure described here assumes that you are about to create a Fabric from scratch. It is generally not feasible to add SSL/TLS security to a pre-existing Fabric, because this puts you in a Catch-22 situation with respect to provisioning the containers.

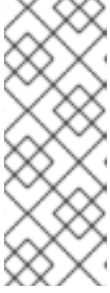
To enable SSL/TLS for Jetty in a Fabric:

- Under the root container's installation directory, create the new directory, **etc/certs**.
- In the **etc/certs** directory, create a new self-signed certificate and private key using the Java **keytool** utility, as follows:

```
keytool -genkeypair -keyalg RSA -dname "CN=Hostname" -ext
SubjectAlternativeName=ip:PUBLIC_IP -validity 365 -keystore alice.ks -alias alice -keypass
KeyPass -storepass StorePass
```

After executing this command, the key pair is stored in the **alice.ks** keystore file under the alias, **alice**. Pay particular attention to the **Hostname** value and the **PUBLIC_IP** value: the specified **Hostname** *must* be the name of the host where the root container is deployed and **PUBLIC_IP** is the public IP address. The other Fabric containers will check that the certificate's Common Name (CN) matches the root container's host name during the SSL/TLS handshake.

For a more detailed explanation of key pairs and instructions for (optionally) signing the resulting certificate with a Certificate Authority (CA), see [Appendix A, *Managing Certificates*](#).

**NOTE**

If there are multiple containers (Fabric servers) in the Fabric ensemble, you must create and deploy a separate key pair for each container in the ensemble, where the specified **Hostname** matches the respective container host. The other containers in the Fabric must then be configured to trust *all* of the ensemble certificates (which you could do, for example, by adding all of the ensemble certificates to a trust store file accessible to the other containers).

3. Start up the root container:

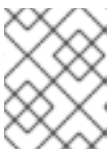
```
./bin/fuse
```

4. Create a new fabric, by entering a console command like the following:

```
JBossFuse:karaf@root> fabric:create --new-user AdminUser
--new-user-password AdminPass
--new-user-role Administrator
--global-resolver manualip
--resolver manualip
--manual-ip Hostname
--zookeeper-password ZooPass
--wait-for-provisioning
```

**IMPORTANT**

The **Hostname** value specified in **fabric:create** must be *exactly the same* **Hostname** value that was assigned to the CN field of the certificate in step 2. Otherwise, when you create a new child container, the hostname check will fail during the SSL/TLS handshake and the child container will fail to provision.

**NOTE**

In a production system (and for any long-running demonstration system), the Fabric server must be deployed on a host that has a static IP address.

5. Edit the Jetty Web server properties for the **org.ops4j.pax.web** persistent ID in the **default** profile. You can edit these properties either from the Fuse Management Console (by navigating to **http://localhost:8181/hawtio** in your browser) or using the built-in editor at the console:

```
JBossFuse:karaf@root> profile-edit --resource org.ops4j.pax.web.properties default
```

Add the following settings to the existing content of the **org.ops4j.pax.web.properties** resource:

```
...
org.osgi.service.http.enabled=false

org.osgi.service.http.secure.enabled=true
org.osgi.service.http.port.secure=${port:8443,8543}
org.ops4j.pax.web.ssl.keystore=AbsolutePathToKeystoreFile
org.ops4j.pax.web.ssl.password=StorePass
org.ops4j.pax.web.ssl.keypassword=KeyPass
```

-

Customize the **org.ops4j.pax.web** settings as follows:

org.osgi.service.http.enabled

Set to **false**, to disable the insecure Jetty HTTP port.

org.osgi.service.http.secure.enabled

Set to **true**, to enable the secure Jetty HTTPS port.

org.osgi.service.http.port.secure

Specifies the TCP port number of the secure HTTPS port. You should use the Fabric port service (see [section "The Port Service" in "Fabric Guide"](#)), which enables you to specify a range of ports for this setting, **port:8443,8543**. This ensure that any child containers are automatically allocated unique port numbers.

org.ops4j.pax.web.ssl.keystore

The location of the Java keystore file on the file system. This should be specified as an *absolute pathname*, to ensure that both the root container and child containers can locate the keystore file (child containers evaluate relatives paths differently from the root container). For example, a typical setting might look like this:

```
org.ops4j.pax.web.ssl.keystore=/opt/servers/jboss-fuse-6.3.0.redhat-187/etc/certs/alice.ks
```

org.ops4j.pax.web.ssl.password

The *store password* that unlocks the Java keystore file.

org.ops4j.pax.web.ssl.keypassword

The *key password* that decrypts the private key stored in the keystore (usually the same as the store password).

6. Create a truststore file for the child containers. There are a few different approaches you can take when creating the truststore:
 - The simplest option is to use the keystore file—for example, **etc/certs/alice.ks**—directly as the truststore.
 - If you need to trust multiple certificates, extract the **alice** certificate from the **alice.ks** truststore and add it to an existing truststore file which contains all of the other certificates you want to trust.
 - If you signed the **alice** certificate with a CA, you can add the CA certificate to the truststore file.
7. The current instructions apply to a fabric that has only one container in its ensemble (the root container). If you set up a fabric with three ensemble servers, however, you would need to make sure that you configure the truststores so that each ensemble server trusts the other two. For example, with three ensemble servers:
 - Add public keys from servers 1 and 2 to truststore for server 3.
 - Add public keys from servers 2 and 3 to truststore for server 1.

- Add public keys from servers 3 and 1 to truststore for server 2.

Alternatively, if you have set up a certificate authority (CA), a more practical approach would be to sign all of the certificates with the same CA certificate and then put the CA certificate into the truststore (that is, in this case only the CA certificate needs to be in the truststore and the same truststore can be used on all of the hosts).

8. Shut down the root container (for example, by entering **shutdown -f** at the console) and specify the truststore and truststore password on the root container. To specify the truststore as a JVM argument, edit the root container's **etc/setenv** file and add the following line:

```
EXTRA_JAVA_OPTS="-Djavax.net.ssl.trustStore=/opt/servers/jboss-fuse-6.3.0.redhat-187/etc/certs/alice.ks -Djavax.net.ssl.trustStorePassword=StorePass"
```

Where this example assumes you are using the **alice.ks** file directly as the truststore.

9. Restart the root container. Search the log (for example, by entering the **log:display** console command) and look for a line like the following:

```
17:37:35,576 | INFO | pool-3-thread-1 | JettyServerImpl | 117 - org.ops4j.pax.web.pax-web-jetty - 4.2.6 | Pax Web available at [0.0.0.0]:[8453]
```

This gives you the port number of the secure Jetty Web server. You can login to the Fuse Management Console using this port—for example, using a URL like the following (not forgetting to specify the scheme as **https**):

```
https://Host:8543
```

10. You can now create a new child container with Jetty security enabled, by specifying the truststore and truststore password as JVM arguments when you create the child container. For example, assuming that you are using the **alice.ks** file directly as a truststore, you can create a secure child container with a command like the following:

```
JBossFuse:karaf@root> container-create-child --jvm-opts='-Djavax.net.ssl.trustStore=/opt/servers/jboss-fuse-6.3.0.redhat-187/etc/certs/alice.ks -Djavax.net.ssl.trustStorePassword=StorePass' --profile fabric root child
```

11. Check the provision status of the new child using the **fabric:container-list** console command (or by monitoring the **Container** tab of the Fuse Management Console). If the child fails to provision, check the logs of both the root container and the child container for errors.

REFERENCES

The Jetty server provides flexible and sophisticated options for configuring security. You can exploit these advanced options by editing the **etc/jetty-ssl.xml** file and configuring it as described in the Jetty security documentation:

- [Configuring SSL](#)
- [API documentation \(all Jetty versions\)](#)

CHAPTER 4. SECURING THE MANAGEMENT CONSOLE

Abstract

The default setting for **Access-Control-Allow-Origin** header for the AMQ Management Console permits unrestricted sharing. To restrict access to the AMQ Management Console, create an access management file which contains a list of the allowed origin URLs. To implement the restrictions, add a system property that references the access management file

4.1. CONTROLLING ACCESS TO THE FUSE MANAGEMENT CONSOLE

Create an access management file called **access-management.xml** in `<installDir>/etc/`. The access management file must contain `<allow-origin>` sections within a `<cors>` section. The `<allow-origin>` section can contain the origin URL provided by browsers with the **Origin:** header, or a wildcard specification with `*`. For example:

```
<cors>
  <!-- Allow cross origin access from www.jolokia.org ... -->
  <allow-origin>http://www.jolokia.org</allow-origin>
  <!-- ... and all servers from jmx4perl.org with any protocol -->
  <allow-origin>*://*.jmx4perl.org</allow-origin>
  <!-- optionally allow access to web console from localhost -->
  <allow-origin>http://localhost:8181/*</allow-origin>
  <!-- Check for the proper origin on the server side, too -->
  <strict-checking/>
</cors>
```

Add the following line to AMQ config script `./bin/setenv`, adding the path to the access management file.

```
export EXTRA_JAVA_OPTS='-Djolokia.policyLocation=file:etc/access-management.xml'
```

When the command `./bin/fuse` is executed, the access management file is referenced and used to restrict access to the AMQ Management Console.

CHAPTER 5. SECURING AN APACHE ACTIVEMQ BROKER

Abstract

Apache ActiveMQ provides two layers of security: *an SSL/TLS security layer*, which can authenticate the broker to its clients, encrypt messages, and guarantee message integrity, and a *JAAS security layer*, which can authenticate clients to the broker. This chapter describes the approach you should take to enable both of these security layers, when the broker is deployed in the Red Hat AMQ OSGi container.

5.1. PROGRAMMING CLIENT CREDENTIALS

Overview

Currently, for Java clients of Red Hat AMQ, you must set the username/password credentials by programming. The `ActiveMQConnectionFactory` provides several alternative methods for specifying the username and password, as follows:

```
ActiveMQConnectionFactory(String userName, String password, String brokerURL);
ActiveMQConnectionFactory(String userName, String password, URI brokerURL);
Connection createConnection(String userName, String password);
QueueConnection createQueueConnection(String userName, String password);
TopicConnection createTopicConnection(String userName, String password);
```

Of these methods, **`createConnection(String userName, String password)`** is the most flexible, since it enables you to specify credentials on a connection-by-connection basis.

Setting login credentials for the Openwire protocol

To specify the login credentials on the client side, pass the username/password credentials as arguments to the **`ActiveMQConnectionFactory.createConnection()`** method, as shown in the following example:

```
// Java
...
public void run() {
    ...
    user = "jdoe";
    password = "secret";
    ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory(url);
    Connection connection = connectionFactory.createConnection(user, password);
    ...
}
```

5.2. CONFIGURING CREDENTIALS FOR BROKER COMPONENTS

Overview

Once authentication is enabled in the broker, every application component that opens a connection to the broker must be configured with credentials. This includes some standard broker components, which are normally configured using Spring XML. To enable you to set credentials on these components, the XML schemas for these components have been extended as described in this section.

Command agent

You can configure the command agent with credentials by setting the **username** attribute and the **password** attribute on the **commandAgent** element in the broker configuration file, *InstallDir/etc/activemq.xml*. By default, the command agent is configured to pick up its credentials from the **activemq.username** property and the **activemq.password** property as shown in the following example:

```
<beans>
...
<commandAgent xmlns="http://activemq.apache.org/schema/core"
  brokerUrl="vm://localhost"
  username="Username" password="Password" />
...
</beans>
```

Apache Camel

The default broker configuration file contains an example of an Apache Camel route that is integrated with the broker. This sample route is defined as follows:

```
<beans>
...
<camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring">
  <package>org.foo.bar</package>
  <route>
    <from uri="activemq:example.A"/>
    <to uri="activemq:example.B"/>
  </route>
</camelContext>
...
</beans>
```

The preceding route integrates with the broker using endpoint URIs that have the component prefix, **activemq:**. For example, the URI, **activemq:example.A**, represents a queue named **example.A** and the endpoint URI, **activemq:example.B**, represents a queue named **example.B**.

The integration with the broker is implemented by the Camel component with bean ID equal to **activemq**. When the broker has authentication enabled, it is necessary to configure this component with a **userName** property and a **password** property, as follows:

```
<beans>
...
<bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent" >
  <property name="connectionFactory">
    <bean class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL" value="vm://localhost?create=false&waitForStart=10000" />
    </bean>
  </property>
  <property name="userName" value="Username" /> <property name="password"
  value="Password" />
</bean>
...
</beans>
```

5.3. BROKER-TO-BROKER AUTHENTICATION

Overview

If you are deploying your brokers in a cluster configuration, and one or more of the brokers is configured to require authentication, then it is necessary to equip *all* of the brokers in the cluster with the appropriate credentials, so that they can all talk to each other.

Configuring the network connector

Given two brokers, Broker A and Broker B, where Broker A is configured to perform authentication, you can configure Broker B to log on to Broker A by setting the **userName** attribute and the **password** attribute in the **networkConnector** element, as follows:

```
<beans ...>
  <broker ...>
    ...
    <networkConnectors>
      <networkConnector name="BrokerABridge"
        username="Username"
        password="Password"
        uri="static://(ssl://brokerA:61616)"/>
      ...
    </networkConnectors>
    ...
  </broker>
</beans>
```

If Broker A is configured to connect to Broker B, Broker A's **networkConnector** element must also be configured with username/password credentials, even if Broker B is not configured to perform authentication. This is because Broker A's authentication plug-in checks for Broker A's username.

5.4. TUTORIAL I: JAAS AUTHENTICATION

Overview

This tutorial shows you how to communicate with the AMQ broker using example producer and consumer JMS clients. The JMS clients must first be modified, however, to provide the requisite username/password JMS credentials.

Prerequisites

The following prerequisites are needed for this tutorial:

- *Apache Ant*—Apache Ant is a free, open source build tool from Apache. You can download the latest version from <http://ant.apache.org/bindownload.cgi> (minimum is 1.8).
- *Apache ActiveMQ installation*—the standalone installation of Apache ActiveMQ has some demonstration code that is not available in Red Hat AMQ. The Apache ActiveMQ distribution is provided in the **InstallDir/extras** directory in an archive format. Uncompress and extract the archive to a convenient installation location, **ActiveMQInstallDir**.

Tutorial steps

To test the example JMS clients with AMQ, perform the following steps:

1. [the section called "Install the consumer and producer JMS clients"](#) .
2. [the section called "Customize the users.properties file"](#) .
3. [the section called "Start the container"](#) .
4. [the section called "Run the consumer with JMS credentials"](#) .
5. [the section called "Run the producer with JMS credentials"](#) .

Install the consumer and producer JMS clients

The Apache ActiveMQ distribution is provided in the **InstallDir/extras** directory in an archive format. Uncompress and extract the archive to a convenient installation location, **ActiveMQInstallDir** (the consumer and producer clients can be accessed by running **ant** targets under the **ActiveMQInstallDir/examples/openwire/swissarmy** directory).

Customize the users.properties file

The **karaf** JAAS realm can be administered by editing the **InstallDir/etc/users.properties** file, where the file contains entries in the following format:

```
Username=Password,Role1,Role2,...
```

For example, the default **users.properties** file shows a sample entry (which is commented out) for the user, **admin**, with password, **admin**, as follows:

```
#admin=admin,admin
```

Customize the **users.properties** file by adding at least one user entry with the **Administrator** role. For example:

```
Username=Password,Administrator
```

Start the container

Change directory to **InstallDir/bin** and enter the following command:

```
./amq
```

Run the consumer with JMS credentials

To connect the consumer tool to the **tcp://localhost:61616** endpoint, change directory to **ActiveMQInstallDir/examples/openwire/swissarmy** and enter the following command:

```
ant consumer -Duser=admin -Dpassword=admin -Durl=tcp://localhost:61616 -Dmax=100
```

You should see some output like the following:

```
-
```



```

Buildfile: build.xml
init:
compile:
consumer:
  [echo] Running consumer against server at $url = tcp://localhost:61616 for subject $subject =
  TEST.FOO
  [java] Connecting to URL: tcp://localhost:61616 (admin:admin)
  [java] Consuming queue: TEST.FOO
  [java] Using a non-durable subscription
  [java] Running 1 parallel threads
  [java] [Thread-2] We are about to wait until we consume: 100 message(s) then we will shutdown

```

Run the producer with JMS credentials

To connect the producer tool to the **tcp://localhost:61616** endpoint, open a new command prompt, change directory to **ActiveMQInstallDir/examples/openwire/swissarmy** and enter the following command:

```
ant producer -Duser=admin -Dpassword=admin -Durl=tcp://localhost:61616 -Dmax=100
```

In the window where the *consumer* tool is running, you should see some output like the following:

```

[java] [Thread-2] Received: 'Message: 0 sent at: Mon Mar 18 17:12:16 CET 2013 ...' (length 1000)
[java] [Thread-2] Received: 'Message: 1 sent at: Mon Mar 18 17:12:16 CET 2013 ...' (length 1000)
[java] [Thread-2] Received: 'Message: 2 sent at: Mon Mar 18 17:12:16 CET 2013 ...' (length 1000)
[java] [Thread-2] Received: 'Message: 3 sent at: Mon Mar 18 17:12:16 CET 2013 ...' (length 1000)
[java] [Thread-2] Received: 'Message: 4 sent at: Mon Mar 18 17:12:16 CET 2013 ...' (length 1000)

```

5.5. TUTORIAL II: SSL/TLS SECURITY

Overview

This tutorial shows you how to enable an SSL/TLS endpoint on the broker and how to configure the example JMS consumer and producer clients so that they can connect to the secure endpoint.

Tutorial steps

To configure SSL/TLS security for a broker deployed in the OSGi container, perform the following steps:

1. [the section called "Install the consumer and producer JMS clients"](#) .
2. [the section called "Install sample broker keystore files"](#) .
3. [the section called "Configure the broker"](#) .
4. [the section called "Encrypt the passwords"](#) .
5. [the section called "Start the container"](#) .
6. [the section called "Configure the consumer and the producer clients"](#) .
7. [the section called "Run the consumer with the SSL protocol"](#) .

8. [the section called "Run the producer with the SSL protocol"](#) .

Install the consumer and producer JMS clients

If you have not already installed the consumer and producer JMS clients, install them now.

The Apache ActiveMQ distribution is provided in the **InstallDir/extras** directory in an archive format. Uncompress and extract the archive to a convenient installation location, **ActiveMQInstallDir** (the consumer and producer clients can be accessed by running **ant** targets under the **ActiveMQInstallDir/examples/openwire/swissarmy** directory).

Install sample broker keystore files

The broker requires the following keystore files:

- *Key store containing broker's own certificate and private key* –used to identify the broker during an SSL handshake.
- *Trust store containing CA certificate* –used to verify that a received client certificate is correctly signed (strictly speaking, the trust store file is only needed by the broker, if the **transport.needClientAuth** options is set to **true** on the broker URI).

For this tutorial, you can use the demonstration certificates provided with the Apache ActiveMQ distribution, in **ActiveMQInstallDir**.

Copy the **broker.ks** and **broker.ts** files from the Apache ActiveMQ distribution's **conf** directory, **ActiveMQInstallDir/conf**, to the **InstallDir/etc** directory of JBoss A-MQ.



WARNING

The demonstration broker key store and broker trust store are provided for testing purposes only. *Do not deploy these certificates in a production system.* To set up a genuinely secure SSL/TLS system, you must generate custom certificates, as described in [Appendix A, Managing Certificates](#).

Configure the broker

Before editing, make a backup copy of the **InstallDir/etc/activemq.xml** file. Use your favorite text editor to edit the file, **InstallDir/etc/activemq.xml**, adding the bolded XML fragments:

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

  <broker xmlns="http://activemq.apache.org/schema/core"
    brokerName="${broker-name}"
    dataDirectory="${data}"
    start="false">
    ...
    <b>sslContext</b>
    <b>sslContext</b>
```

```

    keyStore="${karaf.base}/etc/broker.ks"
    keyStorePassword="password"
    trustStore="${karaf.base}/etc/broker.ts"
    trustStorePassword="password"
  />
</sslContext>

  <transportConnectors>
<transportConnector name="ssl" uri="ssl://0.0.0.0:61617?
transport.enabledProtocols=TLSv1,TLSv1.1,TLSv1.2&maximumConnections=1000"/>
  </transportConnectors>
</broker>

</beans>

```

Note the following key aspects of the broker configuration:

- The Openwire network connector is configured to use SSL, **ssl://localhost:61617?...**
- The enabled protocols are specified explicitly, using the **transport.enabledProtocols** option. This setting effectively disables the SSLv3 protocol, which must not be used because of the POODLE security vulnerability.
- The key store and trust store file locations and passwords are specified by the broker's **sslContext** element.



WARNING

If you are planning to enable SSL/TLS security, you must ensure that you explicitly disable SSLv3 protocol, in order to safeguard against the [Poodle vulnerability \(CVE-2014-3566\)](#). For more details, see [Disabling SSLv3 in JBoss Fuse 6.x and JBoss A-MQ 6.x](#).

Encrypt the passwords

(Optional) If you prefer not to expose passwords in plaintext in the **etc/activemq.xml** file, you can optionally use a Jasypt encrypted property placeholder to obscure the passwords. For example, you can create a **etc/credentials-enc.properties** properties file, with contents like the following:

```

keystore.password=ENC(Cf3Jf3tM+UrSOoaKU50od5CuBa8rxjoL)
truststore.password=ENC(eeWjNyX6FY8Fjp3E+F6qTytV11bZltDp)

```

For instructions on how to generate the encrypted passwords in this file, see [Section 2.3, "Using Encrypted Property Placeholders"](#).

Set the **JASYPT_ENCRYPTION_PASSWORD** environment variable to the value of the master password (which was used to generate the encrypted passwords), as follows:

```

export JASYPT_ENCRYPTION_PASSWORD=MasterPass

```

You must also configure the Jasypt encrypted property placeholder by adding the following bean definitions to the **etc/activemq.xml** file (which replaces the existing plain Spring property placeholder, of **org.springframework.beans.factory.config.PropertyPlaceholderConfigurer** type):

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
  ...
  <bean id="environmentVariablesConfiguration"
class="org.jasypt.encryption.pbe.config.EnvironmentStringPBEConfig">
    <property name="algorithm" value="PBEWithMD5AndDES" />
    <property name="passwordEnvName" value="JASYPT_ENCRYPTION_PASSWORD" />
  </bean>

  <bean id="configurationEncryptor" class="org.jasypt.encryption.pbe.StandardPBEStringEncryptor">
    <property name="config" ref="environmentVariablesConfiguration" />
  </bean>

  <bean id="propertyConfigurer"
class="org.jasypt.spring31.properties.EncryptablePropertyPlaceholderConfigurer">
    <constructor-arg ref="configurationEncryptor" />
    <property name="location" value="file:${karaf.base}/etc/credentials-enc.properties"/>
    <property name="properties">
      <bean class="io.fabric8.mq.fabric.ConfigurationProperties"/>
    </property>
  </bean>
  ...
</beans>
```

You can then configure the passwords in the **sslContext** element as follows:

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

  <broker xmlns="http://activemq.apache.org/schema/core"
    brokerName="${broker-name}"
    dataDirectory="${data}"
    start="false">
    ...
    <sslContext>
      <sslContext>
        keyStore="${karaf.base}/etc/broker.ks"
        keyStorePassword="${keystore.password}"
        trustStore="${karaf.base}/etc/broker.ts"
        trustStorePassword="${truststore.password}"
      />
    </sslContext>

    <transportConnectors>
      <transportConnector name="ssl" uri="ssl://0.0.0.0:61617?
transport.enabledProtocols=TLSv1,TLSv1.1,TLSv1.2&maximumConnections=1000"/>
    </transportConnectors>
  </broker>

</beans>
```

For more details about Jasypt encrypted property placeholders, see [Section 2.3, "Using Encrypted Property Placeholders"](#).

Start the container

Change directory to **InstallDir/bin** and enter the following command:

```
./amq
```



NOTE

If you have configured encrypted property placeholders, you must set the **JASYPT_ENCRYPTION_PASSWORD** environment variable to the Jasypt master password value before starting up the container.

If you are using Jasypt encryption, you must ensure that the **jasypt-encryption** feature is installed in the container. If necessary, install the **jasypt-encryption** feature with the following console command:

```
JBossA-MQ:karaf@root> features:install jasypt-encryption
```

Configure the consumer and the producer clients

To test the broker configured in the OSGi container, you are going to use the example consumer tool and producer tool supplied with the Apache ActiveMQ installation.

Configure the consumer and the producer clients to pick up the client trust store.

1. Open the Ant build file, **ActiveMQInstallDir/examples/openwire/swissarmy/build.xml**, with your favourite text editor.
2. Delete the existing **javax.net.ssl.*** system property settings from the **consumer** target and the **producer** target. That is, remove the lines highlighted in the following example:

```
<project ...>
...
  <target name="consumer" depends="compile" description="Runs a simple consumer">
...
    <java classname="ConsumerTool" fork="yes" maxmemory="100M">
      <classpath refid="javac.classpath" />
      <jvmarg value="-server" />
      <sysproperty key="activemq.home" value="{activemq.home}"/>
      <sysproperty key="javax.net.ssl.keyStore"
value="{javax.net.ssl.keyStore}"/>
      <sysproperty key="javax.net.ssl.trustStore"
value="{javax.net.ssl.trustStore}"/>
      <sysproperty key="javax.net.ssl.keyStorePassword"
value="{javax.net.ssl.keyStorePassword}"/>
      <arg value="--url={url}" />
...
    </java>
  </target>

  <target name="producer" depends="compile" description="Runs a simple producer">
```

```

...
    <java classname="ProducerTool" fork="yes" maxmemory="100M">
      <classpath refid="javac.classpath" />
      <jvmarg value="-server" />
      <sysproperty key="activemq.home" value="${activemq.home}"/>
      <sysproperty key="javax.net.ssl.keyStore"
value="${javax.net.ssl.keyStore}"/>
      <sysproperty key="javax.net.ssl.trustStore"
value="${javax.net.ssl.trustStore}"/>
      <sysproperty key="javax.net.ssl.keyStorePassword"
value="${javax.net.ssl.keyStorePassword}"/>
      <arg value="--url=${url}" />
    </java>
  </target>
...
</project>

```

3. Add the **javax.net.ssl.trustStore** and **javax.net.ssl.trustStorePassword** JSSE system properties to the consumer target and the producer target as shown in the following example:

```

<project ...>
...
  <target name="consumer" depends="compile" description="Runs a simple consumer">
...
    <java classname="ConsumerTool" fork="yes" maxmemory="100M">
      <classpath refid="javac.classpath" />
      <jvmarg value="-server" />
      <sysproperty key="activemq.home" value="${activemq.home}"/>
      <sysproperty key="javax.net.ssl.trustStore"
        value="${activemq.home}/conf/client.ts"/>
      <sysproperty key="javax.net.ssl.trustStorePassword"
        value="password"/>
      <arg value="--url=${url}" />
    </java>
  </target>

  <target name="producer" depends="compile" description="Runs a simple producer">
...
    <java classname="ProducerTool" fork="yes" maxmemory="100M">
      <classpath refid="javac.classpath" />
      <jvmarg value="-server" />
      <sysproperty key="activemq.home" value="${activemq.home}"/>
      <sysproperty key="javax.net.ssl.trustStore"
        value="${activemq.home}/conf/client.ts"/>
      <sysproperty key="javax.net.ssl.trustStorePassword"
        value="password"/>
      <arg value="--url=${url}" />
    </java>
  </target>
...
</project>

```

In the context of the Ant build tool, this is equivalent to adding the system properties to the command line.

Run the consumer with the SSL protocol

To connect the consumer tool to the **ssl://localhost:61617** endpoint (Openwire over SSL), change directory to **ActiveMQInstallDir/examples/openwire/swissarmy** and enter the following command:

```
ant consumer -Duser=admin -Dpassword=admin -Durl=ssl://localhost:61617 -Dmax=100 -
Dactivemq.home=../../..
```

You should see some output like the following:

```
Buildfile: build.xml
init:
compile:
consumer:
 [echo] Running consumer against server at $url = ssl://localhost:61617 for subject $subject =
TEST.FOO
 [java] Connecting to URL: ssl://localhost:61617 (admin:admin)
 [java] Consuming queue: TEST.FOO
 [java] Using a non-durable subscription
 [java] Running 1 parallel threads
 [java] [Thread-2] We are about to wait until we consume: 100 message(s) then we will shutdown
```

Run the producer with the SSL protocol

To connect the producer tool to the **ssl://localhost:61617** endpoint, open a new command prompt, change directory to **ActiveMQInstallDir/examples/openwire/swissarmy** and enter the following command:

```
ant producer -Duser=admin -Dpassword=admin -Durl=ssl://localhost:61617 -Dmax=100 -
Dactivemq.home=../../..
```

In the window where the *consumer* tool is running, you should see some output like the following:

```
[java] [Thread-2] Received: 'Message: 0 sent at: Tue Mar 19 10:07:25 CET 2013 ...' (length 1000)
 [java] [Thread-2] Received: 'Message: 1 sent at: Tue Mar 19 10:07:25 CET 2013 ...' (length 1000)
 [java] [Thread-2] Received: 'Message: 2 sent at: Tue Mar 19 10:07:26 CET 2013 ...' (length 1000)
 [java] [Thread-2] Received: 'Message: 3 sent at: Tue Mar 19 10:07:26 CET 2013 ...' (length 1000)
 [java] [Thread-2] Received: 'Message: 4 sent at: Tue Mar 19 10:07:26 CET 2013 ...' (length 1000)
```

5.6. SECURITY OPTIONS FOR JMS OBJECTMESSAGE SERIALIZATION

Overview

The **javax.jms.ObjectMessage** type can be used to send messages containing a serialized Java object. This feature has been available since JMS 1.0 and is supported by Apache ActiveMQ, but use of this feature is generally *not* recommended, for the following reasons:

- Architecturally, messaging systems enable loose coupling between producers and consumers, but **ObjectMessage** re-introduces tight coupling, thereby negating a key benefit of the message-based architecture.
- Using **ObjectMessage** introduces coupling of class paths between producers and consumers.
- Using **ObjectMessage** presents a significant security risk, because the serialized objects can be used to transfer malicious code.

Security risks

ObjectMessage objects depend on Java serialization to marshal and unmarshal their object payload. This process is generally considered unsafe, because a malicious payload can exploit the host system. For this reason, starting from AMQ 6.3, AMQ forces users to explicitly whitelist packages that can be exchanged using **ObjectMessage** messages.

Whitelisting Java packages

To whitelist Java packages, so that they can be used in a serialized object message, you are required to list the acceptable packages in the **org.apache.activemq.SERIALIZABLE_PACKAGES** system property. This system property can be used both on the server side (broker) and on the client side.

For example, on the broker side, you can set this property in the ***InstallDir/etc/system.properties*** file, as follows:

```
org.apache.activemq.SERIALIZABLE_PACKAGES="java.lang,java.util,org.apache.activemq,org.fusesource.hawtbuf,com.thoughtworks.xstream.mapper,com.mycompany.myapp"
```

Which adds the **com.mycompany.myapp** package to the list of trusted packages. Note that the other packages listed here are enabled by default, because they are necessary for the regular broker to work.

Bypassing the whitelist mechanism

In case you want to bypass the whitelist mechanism, you can allow all packages to be trusted by specifying the * wildcard character—for example:

```
org.apache.activemq.SERIALIZABLE_PACKAGES="*"
```



WARNING

Disabling the whitelist mechanism can be useful in a testing environment, but it should *not* be used in a deployed system.

Client-side whitelisting API

On the client side, you can also configure trusted packages using the **org.apache.activemq.SERIALIZABLE_PACKAGES** system property, but this approach is usually not convenient for client applications. An alternative approach is to use the API methods defined on the

ActiveMQConnectionFactory class (where these API settings would override the system property, if it is set). There are two additional methods defined available:

- The **setTrustedPackages()** method allows you to set the list of trusted packages you want to be to unserialize—for example:

```
ActiveMQConnectionFactory factory = new
ActiveMQConnectionFactory("tcp://localhost:61616");
factory.setTrustedPackages(new
ArrayList(Arrays.asList("org.apache.activemq.test,org.apache.camel.test".split(","))));
```

The list of trusted packages can also be set in XML. For example, a Camel endpoint in a Spring XML file can be configured as follows:

```
<bean id="connectionFactory"
class="org.apache.activemq.spring.ActiveMQConnectionFactory">
  <property name="brokerURL" value="tcp://localhost:61616"/>
  <property name="trustedPackages">
    <list>
      <value>org.apache.activemq.test</value>
      <value>org.apache.camel.test</value>
    </list>
  </property>
</bean>
<bean id="jmsConfig" class="org.apache.camel.component.jms.JmsConfiguration">
  <property name="connectionFactory" ref="connectionFactory"/>
</bean>
<bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent">
  <property name="configuration" ref="jmsConfig"/>
</bean>
```

- The **setTrustAllPackages()** allows you to turn off the security check and trust all classes. It can be useful for testing purposes. For example:

```
ActiveMQConnectionFactory factory = new
ActiveMQConnectionFactory("tcp://localhost:61616");
factory.setTrustAllPackages(true);
```

The whitelist mechanism can also be disabled in Spring XML as follows:

```
<bean id="connectionFactory"
class="org.apache.activemq.spring.ActiveMQConnectionFactory">
  <property name="brokerURL" value="tcp://localhost:61616"/>
  <property name="trustAllPackages" value="true"/>
</bean>
<bean id="jmsConfig" class="org.apache.camel.component.jms.JmsConfiguration">
  <property name="connectionFactory" ref="connectionFactory"/>
</bean>
<bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent">
  <property name="configuration" ref="jmsConfig"/>
</bean>
```



WARNING

Disabling the whitelist mechanism can be useful in a testing environment, but it should *not* be used in a deployed system.

CHAPTER 6. SECURING THE CAMEL ACTIVEMQ COMPONENT

Abstract

The Camel ActiveMQ component enables you to define JMS endpoints in your routes that can connect to an Apache ActiveMQ broker. In order to make your Camel ActiveMQ endpoints secure, you must create an instance of a Camel ActiveMQ component that uses a *secure* connection factory.

6.1. SECURE ACTIVEMQ CONNECTION FACTORY

Overview

Apache Camel provides an Apache ActiveMQ component for defining Apache ActiveMQ endpoints in a route. The Apache ActiveMQ endpoints are effectively Java clients of the broker and you can either define a consumer endpoint (typically used at the start of a route to *poll for JMS* messages) or define a producer endpoint (typically used at the end or in the middle of a route to *send JMS* messages to a broker).

When the remote broker is secure (SSL security, JAAS security, or both), the Apache ActiveMQ component must be configured with the required client security settings.

Programming the security properties

Apache ActiveMQ enables you to program SSL security settings (and JAAS security settings) by creating and configuring an instance of the **ActiveMQSslConnectionFactory** JMS connection factory. Programming the JMS connection factory is the correct approach to use in the context of the containers such as OSGi, J2EE, Tomcat, and so on, because these settings are local to the application using the JMS connection factory instance.



NOTE

A standalone broker can configure SSL settings using *Java system properties*. For clients deployed in a container, however, this is *not* a practical approach, because the configuration must apply only to individual bundles, not the entire OSGi container. A Camel ActiveMQ endpoint is effectively a kind of Apache ActiveMQ Java client, so this restriction applies also to Camel ActiveMQ endpoints.

Defining a secure connection factory

[Example 6.1, "Defining a Secure Connection Factory Bean"](#) shows how to create a secure connection factory bean in Spring XML, enabling both SSL/TLS security *and* JAAS authentication.

Example 6.1. Defining a Secure Connection Factory Bean

```
<bean id="jmsConnectionFactory"
  class="org.apache.activemq.ActiveMQSslConnectionFactory">
  <property name="brokerURL" value="ssl://localhost:61617" />
  <property name="userName" value="Username"/>
  <property name="password" value="Password"/>
  <property name="trustStore" value="/conf/client.ts"/>
  <property name="trustStorePassword" value="password"/>
</bean>
```

The following properties are specified on the **ActiveMQSslConnectionFactory** class:

brokerURL

The URL of the remote broker to connect to, where this example connects to an SSL-enabled OpenWire port on the local host. The broker must also define a corresponding transport connector with compatible port settings.

userName and password

Any valid JAAS login credentials, **Username** and **Password**.

trustStore

Location of the Java keystore file containing the certificate trust store for SSL connections. The location is specified as a classpath resource. If a relative path is specified, the resource location is relative to the **org/jbossfuse/example** directory on the classpath.

trustStorePassword

The password that unlocks the keystore file containing the trust store.

It is also possible to specify **keyStore** and **keyStorePassword** properties, but these would only be needed, if SSL mutual authentication is enabled (where the client presents an X.509 certificate to the broker during the SSL handshake).

6.2. EXAMPLE CAMEL ACTIVEMQ COMPONENT CONFIGURATION

Overview

This section describes how to initialize and configure a sample Camel ActiveMQ component instance, which you can then use to define ActiveMQ endpoints in a Camel route. This makes it possible for a Camel route to send or receive messages from a broker.

Prerequisites

The **camel-activemq** feature, which defines the bundles required for the Camel ActiveMQ component, is *not* installed by default. To install the **camel-activemq** feature, enter the following console command:

```
JBossFuse:karaf@root> features:install camel-activemq
```

Sample Camel ActiveMQ component

The following Spring XML sample shows a complete configuration of a Camel ActiveMQ component that has both SSL/TLS security and JAAS authentication enabled. The Camel ActiveMQ component instance is defined to with the **activemqssl** bean ID, which means it is associated with the **activemqssl** scheme (which you use when defining endpoints in a Camel route).

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

Configure the activemqssl component:

```
-->
<bean id="jmsConnectionFactory"
  class="org.apache.activemq.ActiveMQSslConnectionFactory">
  <property name="brokerURL" value="ssl://localhost:61617" />
  <property name="userName" value="Username"/>
  <property name="password" value="Password"/>
  <property name="trustStore" value="/conf/client.ts"/>
  <property name="trustStorePassword" value="password"/>
</bean>

<bean id="pooledConnectionFactory"
  class="org.apache.activemq.pool.PooledConnectionFactory">
  <property name="maxConnections" value="8" />
  <property name="maximumActive" value="500" />
  <property name="connectionFactory" ref="jmsConnectionFactory" />
</bean>

<bean id="jmsConfig" class="org.apache.camel.component.jms.JmsConfiguration">
  <property name="connectionFactory" ref="pooledConnectionFactory"/>
  <property name="transacted" value="false"/>
  <property name="concurrentConsumers" value="10"/>
</bean>

<bean id="activemqssl"
  class="org.apache.activemq.camel.component.ActiveMQComponent">
  <property name="configuration" ref="jmsConfig"/>
</bean>

</beans>
```

Sample Camel route

The following Camel route defines a sample endpoint that sends messages securely to the **security.test** queue on the broker, using the **activemqssl** scheme to reference the Camel ActiveMQ component defined in the preceding example:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
  ...
  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
      <from uri="timer://myTimer?fixedRate=true&period=5000"/>
      <transform><constant>Hello world!</constant></transform>
      <to uri="activemqssl:security.test"/>
    </route>
  </camelContext>
  ...
</beans>
```

CHAPTER 7. SSL/TLS SECURITY

Abstract

You can use SSL/TLS security to secure connections to brokers for a variety of different protocols: Openwire over TCP/IP, Openwire over HTTP, and Stomp.

7.1. INTRODUCTION TO SSL/TLS

Overview

The Secure Sockets Layer (SSL) protocol was originally developed by Netscape Corporation to provide a mechanism for secure communication over the Internet. Subsequently, the protocol was adopted by the Internet Engineering Task Force (IETF) and renamed to Transport Layer Security (TLS). The latest specification of the TLS protocol is [RFC 5246](#).

The SSL/TLS protocol sits between an application protocol layer and a reliable transport layer (such as TCP/IP). It is independent of the application protocol and can thus be layered underneath many different protocols, for example: HTTP, FTP, SMTP, and so on.

SSL/TLS security features

The SSL/TLS protocol supports the following security features:

- *Privacy*—messages are encrypted using a secret symmetric key, making it impossible for eavesdroppers to read messages sent over the connection.
- *Message integrity*—messages are digitally signed, to ensure that they cannot be tampered with.
- *Authentication*—the identity of the target (server program) is authenticated and (optionally) the client as well.
- *Immunity to man-in-the-middle attacks*—because of the way authentication is performed in SSL/TLS, it is impossible for an attacker to interpose itself between a client and a target.

Cipher suites

To support all of the facets of SSL/TLS security, a number of different security algorithms must be used together. Moreover, for each of the security features (for example, message integrity), there are typically several different algorithms available. To manage these alternatives, the security algorithms are grouped together into *cipher suites*. Each cipher suite contains a complete collection of security algorithms for the SSL/TLS protocol. />

Public key cryptography

Public key cryptography (also known as *asymmetric cryptography*) plays a critically important role in SSL/TLS security. With this form of cryptography, encryption and decryption is performed using a matching pair of keys: a *public key* and a *private key*. A message encrypted by the public key can *only* be decrypted by the private key; and a message encrypted by the private key can *only* be decrypted by the public key. This basic mathematical property has some important consequences for cryptography:

- It becomes extremely easy to establish secure communications with people you have never previously had any contact with. Simply publish the public key in some accessible place. Anyone

can now download the public key and use it to encrypt a message that *only you* can decrypt, using your private key.

- You can use your private key to digitally sign messages. Given a message to sign, simply generate a hash value from the message, encrypt that hash value using your private key, and append it to the message. Now, anyone can use the public key to decrypt the hash value and check that the message has not been tampered with.



NOTE

Actually, it is not compulsory to use public key cryptography with SSL/TLS. But the SSL/TLS protocol is practically useless (and very insecure) without it.

X.509 certificates

An X.509 certificate provides a way of binding an identity (in the form of an X.500 *distinguished name*) to a public key. X.509 is a standard specified by the IETF and the most recent specification is [RFC 4158](#). The X.509 certificate consists essentially of an identity concatenated with a public key, with the whole certificate being digitally signed in order to guarantee the association between the identity and the public key.

But who signs the certificate? It has to be someone (or some identity) that you trust. The certificate signer could be one of the following:

- *Self*—if the certificate signs itself, it is called a *self-signed certificate*. If you need to deploy a self-signed certificate, the certificate must be obtained from a secure channel. The only guarantee you have of the certificate's authenticity is that you obtained it from a trusted source.
- *CA certificate*—a more scalable solution is to sign certificates using a Certificate Authority (CA) certificate. In this case, you only need to be careful about deploying the original CA certificate (that is, obtaining it through a secure channel). All of the certificates signed by this CA, on the other hand, can be distributed over insecure, public channels. The trusted CA can then be used to verify the signature on the certificates. In this case, the CA certificate is self-signed.
- *Chain of CA certificates*—an extension of the idea of signing with a CA certificate is to use a chain of CA certificates. For example, certificate X could be signed by CA foo, which is signed by CA bar. The last CA certificate in the chain (the *root certificate*) is self-signed.

For more details about managing X.509 certificates, see [Appendix A, Managing Certificates](#).

Target-only authentication

The most common way to configure SSL/TLS is to associate an X.509 certificate with the target (server side) but not with the client. This implies that the client can verify the identity of the target, but the target cannot verify the identity of the client (at least, not through the SSL/TLS protocol). It might seem strange that we worry about protecting clients (by confirming the target identity) but not about protecting the target. Keep in mind, though, that SSL/TLS security was originally developed for the Internet, where protecting clients is a high priority. For example, if you are about to connect to your bank's Web site, you want to be very sure that the Web site is authentic. Also, it is typically easier to authenticate clients using other mechanisms (such as HTTP Basic Authentication), which do not incur the high maintenance overhead of generating and distributing X.509 certificates.

7.2. SECURE TRANSPORT PROTOCOLS

Overview

Red Hat AMQ provides a common framework for adding SSL/TLS security to its transport protocols. All of the transport protocols discussed here are secured using the JSSE framework and most of their configuration settings are shared.

Transport protocols

Table 7.1, “Secure Transport Protocols” shows the transport protocols that can be secured using SSL/TLS.

Table 7.1. Secure Transport Protocols

URL	Description
ssl://Host:Port	Endpoint URL for Openwire over TCP/IP, where the socket layer is secured using SSL or TLS.
https://Host:Port	Endpoint URL for Openwire over HTTP, where the socket layer is secured using SSL or TLS.
stomp+ssl://Host:Port	Endpoint URL for Stomp over TCP/IP, where the socket layer is secured using SSL or TLS.
mqtt+nio+ssl://Host:Port	Endpoint URL for MQTT over Java NIO, where the socket layer is secured using SSL or TLS.

Verify Host Name

Disabled by default, you can enable the capability of verifying the host name on the transport configuration by using a URL parameter as follows:

```
ssl://localhost:61616?transport.verifyHostName=true
```

To enable on the client side, use the following:

```
ssl://localhost:61616?socket.verifyHostName=true
```



IMPORTANT

It is especially important to **enable** this option to prevent against *man-in-the-middle* attacks, particularly in locked down systems.

7.3. JAVA KEYSTORES

Overview

Java keystores provide a convenient mechanism for storing and deploying X.509 certificates and private keys. Red Hat AMQ uses Java keystore files as the standard format for deploying certificates

Prerequisites

The Java keystore is a feature of the *Java platform Standard Edition (SE)* from Oracle. To perform the tasks described in this section, you will need to install a recent version of the Java Development Kit (JDK) and ensure that the JDK **bin** directory is on your path. See [Java SE](#).

Default keystore provider

Oracle's JDK provides a standard file-based implementation of the keystore. The instructions in this section presume you are using the standard keystore. If there is any doubt about the kind of keystore you are configured to use, check the following line in your **java.security** file (located either in **JavalninstallDir/lib/security** or **JavalninstallDir/jre/lib/security**):

```
keystore.type=jks
```

The **jks** (or **JKS**) keystore type represents the standard keystore.

Customizing the keystore provider

Java also allows you to provide a custom implementation of the keystore, by implementing the **java.security.KeyStoreSpi** class. For details of how to do this see the following references:

- [Key and Certificate Management Tool](#)
- [How to Implement a Provider for the JCA](#)

If you use a custom keystore provider, you should consult the third-party provider documentation for details of how to manage certificates and private keys with this provider.

Store password

The keystore repository is protected by a *store password*, which is defined at the same time the keystore is created. Every time you attempt to access or modify the keystore, you must provide the store password.



NOTE

The store password can also be referred to as a *keystore password* or a *truststore password*, depending on what kind of entries are stored in the keystore file. The function of the password in both cases is the same: that is, to unlock the keystore file.

Keystore entries

The keystore provides two distinct kinds of entry for storing certificates and private keys, as follows:

- *Key entries*—each key entry contains the following components:
 - A private key.
 - An X.509 certificate (can be v1, v2, or v3) containing the public key that matches this entry's private key.
 - Optionally, one or more CA certificates that belong to the preceding certificate's trust chain.

**NOTE**

The CA certificates belonging to a certificate's trust chain can be stored either in its key entry or in trusted certificate entries.

In addition, each key entry is tagged by an alias and protected by a key password. To access a particular key entry in the keystore, you must provide both the alias and the key password.

- *Trusted certificate entries*—each trusted certificate entry contains just a single X.509 certificate.

Each trusted certificate entry is tagged by an alias. There is no need to protect the entry with a password, however, because the X.509 certificate contains only a public key.

Keystore utilities

The Java platform SE provides two keystore utilities: **keytool** and **jarsigner**. Only the **keytool** utility is needed here.

7.4. HOW TO USE X.509 CERTIFICATES

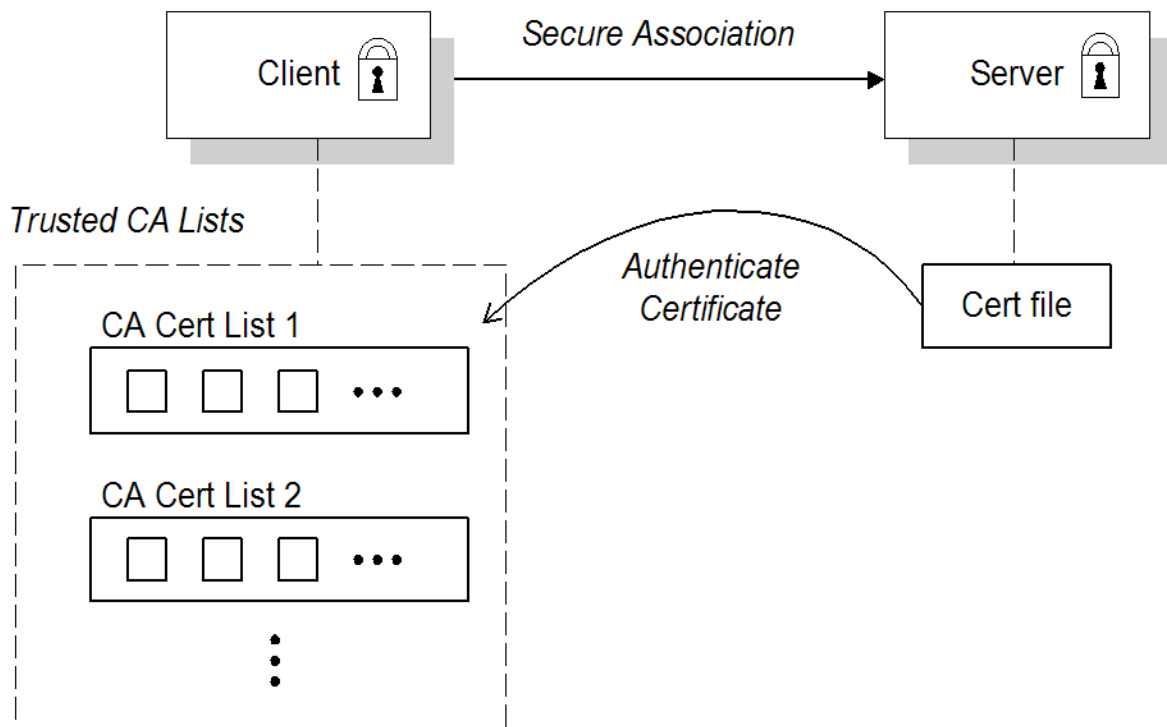
Overview

Before you can understand how to deploy X.509 certificates in a real system, you need to know about the different authentication scenarios supported by the SSL/TLS protocol. The way you deploy the certificates depends on what kind of authentication scenario you decide to adopt for your application.

Target-only authentication

In the target-only authentication scenario, as shown in [Figure 7.1, "Target-Only Authentication Scenario"](#), the target (in this case, the broker) presents its own certificate to the client during the SSL/TLS handshake, so that the client can verify the target's identity. In this scenario, therefore, the target is authentic to the client, but the client is not authentic to the target.

Figure 7.1. Target-Only Authentication Scenario

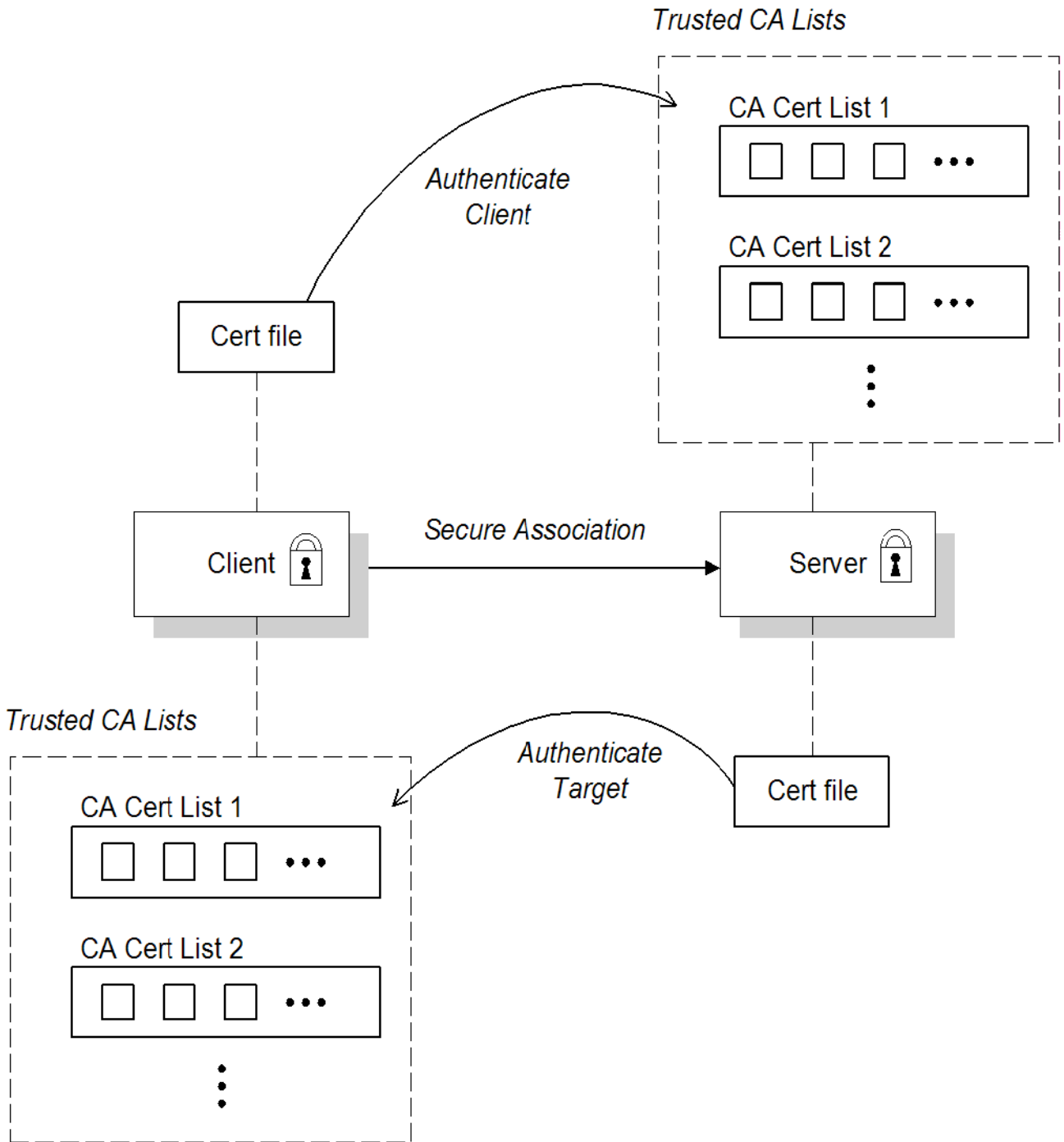


The broker is configured to have its own certificate and private key, which are both stored in the file, **broker.ks**. The client is configured to have a trust store, **client.ts**, that contains the certificate that originally signed the broker certificate. Normally, the trusted certificate is a Certificate Authority (CA) certificate.

Mutual authentication

In the mutual authentication scenario, as shown in [Figure 7.2, "Mutual Authentication Scenario"](#), the target presents its own certificate to the client and the client presents its own certificate to the target during the SSL/TLS handshake, so that both the client and the target can verify each other's identity. In this scenario, therefore, the target is authentic to the client and the client is authentic to the target.

Figure 7.2. Mutual Authentication Scenario



Because authentication is mutual in this scenario, both the client and the target must be equipped with a full set of certificates. The client is configured to have its own certificate and private key in the file, **client.ks**, and a trust store, **client.ts**, which contains the certificate that signed the target certificate. The target is configured to have its own certificate and private key in the file, **broker.ks**, and a trust store, **broker.ts**, which contains the certificate that signed the client certificate.

Selecting the authentication scenario

Various combinations of target and client authentication are supported by the SSL/TLS protocols. In general, SSL/TLS authentication scenarios are controlled by selecting a specific cipher suite (or cipher suites) and by setting the **WantClientAuth** or **NeedClientAuth** flags in the SSL/TLS protocol layer. The following list describes all of the possible authentication scenarios:

- *Target-only authentication*—this is the most important authentication scenario. If you want to

authenticate the client as well, the most common approach is to let the client log on using username/password credentials, which can be sent securely through the encrypted channel established by the SSL/TLS session.

- *Target authentication and optional client authentication*—if you want to authenticate the client using an X.509 certificate, simply configure the client to have its own certificate. By default, the target will authenticate the client's certificate, if it receives one.
- *Target authentication and required client authentication*—if want to enforce client authentication using an X.509 certificate, you can set the **NeedClientAuth** flag on the SSL/TLS protocol layer. When this flag is set, the target would raise an error if the client fails to send a certificate during the SSL/TLS handshake.
- *No authentication*—this scenario is potentially dangerous from a security perspective, because it is susceptible to a man-in-the-middle attack. *It is therefore recommended that you always avoid using this (non-)authentication scenario.*



NOTE

It is theoretically possible to get this scenario, if you select one of the anonymous Diffie-Hellman cipher suites for the SSL/TLS session. In practice, however, you normally do not need to worry about these cipher suites, because they have a low priority amongst the cipher suites supported by the **SunJSSE** security provider. Other, more secure cipher suites normally take precedence.

Custom certificates

For a real deployment of a secure SSL/TLS application, you must first create a collection of custom X.509 certificates and private keys. For detailed instructions on how to go about creating and managing your X.509 certificates, see [Appendix A, Managing Certificates](#).

7.5. CONFIGURING JSSE SYSTEM PROPERTIES

Overview

Java Secure Socket Extension (JSSE) provides the underlying framework for the SSL/TLS implementation in Red Hat AMQ. In this framework, you configure the SSL/TLS protocol and deploy X.509 certificates using a variety of JSSE system properties.

JSSE system properties

[Table 7.2, "JSSE System Properties"](#) shows the JSSE system properties that can be used to configure SSL/TLS security for the SSL (Openwire over SSL), HTTPS (Openwire over HTTPS), and Stomp+SSL (Stomp over SSL) transport protocols.

Table 7.2. JSSE System Properties

System Property Name	Description
javax.net.ssl.keyStore	Location of the Java keystore file containing an application process's own certificate and private key. On Windows, the specified pathname must use forward slashes, /, in place of backslashes, \.

System Property Name	Description
javax.net.ssl.keyStorePassword	<p>Password to access the private key from the keystore file specified by javax.net.ssl.keyStore. This password is used twice:</p> <ul style="list-style-type: none"> ● To unlock the keystore file (store password), and ● To decrypt the private key stored in the keystore (key password). <p>In other words, the JSSE framework requires these passwords to be identical.</p>
javax.net.ssl.keyStoreType	<p><i>(Optional)</i> For Java keystore file format, this property has the value jks (or JKS). You do not normally specify this property, because its default value is already jks.</p>
javax.net.ssl.trustStore	<p>Location of the Java keystore file containing the collection of CA certificates trusted by this application process (trust store). On Windows, the specified pathname must use forward slashes, /, in place of backslashes, \.</p> <p>If a trust store location is not specified using this property, the SunJSSE implementation searches for and uses a keystore file in the following locations (in order):</p> <ol style="list-style-type: none"> 1. \$JAVA_HOME/lib/security/jssecacerts 2. \$JAVA_HOME/lib/security/cacerts
javax.net.ssl.trustStorePassword	<p>Password to unlock the keystore file (store password) specified by javax.net.ssl.trustStore.</p>
javax.net.ssl.trustStoreType	<p><i>(Optional)</i> For Java keystore file format, this property has the value jks (or JKS). You do not normally specify this property, because its default value is already jks.</p>
javax.net.debug	<p>To switch on logging for the SSL/TLS layer, set this property to ssl.</p>



WARNING

The default trust store locations (in the **jssecacerts** and the **cacerts** directories) present a potential security hazard. If you do not take care to manage the trust stores under the JDK installation or if you do not have control over which JDK installation is used, you might find that the effective trust store is too lax.

To be on the safe side, it is recommended that you *always* set the **javax.net.ssl.trustStore** property for a secure client or server, so that you have control over the CA certificates trusted by your application.

Setting properties at the command line

On the client side and in the broker, you can set the JSSE system properties on the Java command line using the standard syntax, **-DProperty=Value**. For example, to specify JSSE system properties to a client program, **com.redhat.Client**:

```
java -Djavax.net.ssl.trustStore=truststores/client.ts com.redhat.Client
```

To configure a broker to use the demonstration broker keystore and demonstration broker trust store, you can set the **SSL_OPTS** environment variable as follows, on Windows:

```
set SSL_OPTS=-Djavax.net.ssl.keyStore=C:/Programs/FUSE/fuse-message-broker-6.3.0.redhat-187/conf/broker.ks
-Djavax.net.ssl.keyStorePassword=password
-Djavax.net.ssl.trustStore=C:/Programs/FUSE/fuse-message-broker-6.3.0.redhat-187/conf/broker.ts
-Djavax.net.ssl.trustStorePassword=password
```

Or on UNIX platforms (Bourne shell):

```
SSL_OPTS=-Djavax.net.ssl.keyStore=/local/FUSE/fuse-message-broker-6.3.0.redhat-187/conf/broker.ks
-Djavax.net.ssl.keyStorePassword=password
-Djavax.net.ssl.trustStore=/local/FUSE/fuse-message-broker-6.3.0.redhat-187/conf/broker.ts
-Djavax.net.ssl.trustStorePassword=password
export SSL_OPTS
```

You can then launch the broker using the **bin/activemq[.bat|.sh]** script



NOTE

The **SSL_OPTS** environment variable is simply a convenient way of passing command-line properties to the **bin/activemq[.bat|.sh]** script. It is *not* accessed directly by the broker runtime or the JSSE package.

Setting properties by programming

You can also set JSSE system properties using the standard Java API, as long as you set the properties before the relevant transport protocol is initialized. For example:

```
// Java
import java.util.Properties;
...
Properties systemProps = System.getProperties();
systemProps.put(
    "javax.net.ssl.trustStore",
    "C:/Programs/FUSE/fuse-message-broker-6.3.0.redhat-187/conf/client.ts"
);
System.setProperties(systemProps);
```

7.6. SETTING SECURITY CONTEXT FOR THE OPENWIRE/SSL PROTOCOL

Overview

Apart from configuration using JSSE system properties, the Openwire/SSL protocol (with schema, **ssl:**) also supports an option to set its SSL security context using the broker configuration file.



NOTE

The methods for setting the security context described in this section are available *exclusively* for the Openwire/SSL protocol. These features are *not* supported by the HTTPS protocol.

Setting security context in the broker configuration file

To configure the Openwire/SSL security context in the broker configuration file, edit the attributes in the **sslContext** element. For example, the default broker configuration file, **etc/activemq.xml**, includes the following entry:

```
<beans ...>
...
<broker ...>
  <sslContext>
    <sslContext keyStore="file:${activemq.base}/conf/broker.ks"
      keyStorePassword="password"
      trustStore="file:${activemq.base}/conf/broker.ts"
      trustStorePassword="password"/>
  </sslContext>
...
</broker>
...
</beans>
```

Where the **activemq.base** property is defined in the **activemq[.bat|.sh]** script. You can specify any of the following **sslContext** attributes:

- **keyStore**—equivalent to setting **javax.net.ssl.keyStore**.
- **keyStorePassword**—equivalent to setting **javax.net.ssl.keyStorePassword**.

- **keyStoreType**—equivalent to setting **javax.net.ssl.keyStoreType**.
- **keyStoreAlgorithm**—defaults to JKS.
- **trustStore**—equivalent to setting **javax.net.ssl.trustStore**.
- **trustStorePassword**—equivalent to setting **javax.net.ssl.trustStorePassword**.
- **trustStoreType**—equivalent to setting **javax.net.ssl.trustStoreType**.

7.7. SECURING JAVA CLIENTS

ActiveMQSslConnectionFactory class

To support SSL/TLS security in Java clients, Red Hat AMQ provides the **org.apache.activemq.ActiveMQSslConnectionFactory** class. Use the **ActiveMQSslConnectionFactory** class in place of the insecure **ActiveMQConnectionFactory** class in order to enable SSL/TLS security in your clients.

The **ActiveMQSslConnectionFactory** class exposes the following methods for configuring SSL/TLS security:

setTrustStore(String)

Specifies the location of the client's trust store file, in JKS format (as managed by the Java **keystore** utility).

setTrustStorePassword(String)

Specifies the password that unlocks the client trust store.

setKeyStore(String)

(Optional) Specifies the location of the client's own X.509 certificate and private key in a key store file, in JKS format (as managed by the Java **keystore** utility). Clients normally do *not* need to provide their own certificate, unless the broker SSL/TLS configuration specifies that client authentication is required.

setKeyStorePassword(String)

(Optional) Specifies the password that unlocks the client key store. This password is also used to decrypt the private key from in the key store.



NOTE

For more advanced applications, **ActiveMQSslConnectionFactory** also exposes the **setKeyAndTrustManagers** method, which lets you specify the **javax.net.ssl.KeyManager[]** array and the **javax.net.ssl.TrustManager[]** array directly.

Specifying the trust store and key store locations

Location strings passed to the **setTrustStore** and **setKeyStore** methods can have either of the following formats:

- A *pathname*—where no scheme is specified, for example, `/conf/client.ts`. In this case the resource is loaded from the classpath, which is convenient to use when the client and its certificates are packaged in a JAR file.
- A *Java URL*—where you can use any of the standard Java URL schemes, such as `http` or `file`. For example, to reference the file, `C:\ActiveMQ\conf\client.ts`, in the filesystem on a Windows O/S, use the URL, `file:///C:/ActiveMQ/conf/client.ts`.

Sample client code

[Example 7.1, "Java Client Using the ActiveMQSslConnectionFactory Class"](#) shows an example of how to initialize a message producer client in Java, where the message producer connects to the broker using the SSL/TLS protocol. The key step here is that the client uses the `ActiveMQSslConnectionFactory` class to create the connection, also setting the trust store and trust store password (no key store is required here, because we are assuming that the broker port does not require client authentication).

Example 7.1. Java Client Using the ActiveMQSslConnectionFactory Class

```
import javax.jms.Connection;
import javax.jms.Destination;
import javax.jms.MessageProducer;
import javax.jms.Session;

import org.apache.activemq.ActiveMQSslConnectionFactory;
...
String url = "ssl://localhost:61617" // The broker URL

// Configure the secure connection factory.
ActiveMQSslConnectionFactory connectionFactory = new
ActiveMQSslConnectionFactory(url);
connectionFactory.setTrustStore("/conf/client.ts");
connectionFactory.setTrustStorePassword("password");

// Create the connection.
Connection connection = connectionFactory.createConnection();
connection.start();

// Create the session
Session session = connection.createSession(transacted, Session.AUTO_ACKNOWLEDGE);
Destination destination = session.createQueue(subject);

// Create the producer.
MessageProducer producer = session.createProducer(destination);
```

CHAPTER 8. AUTHORIZATION

Abstract

Red Hat AMQ authorization implements group-based access control and allows you to control access at the granularity level of destinations or of individual messages.

8.1. SIMPLE AUTHORIZATION PLUG-IN

Overview

In a security system without authorization, every successfully authenticated user would have unrestricted access to every queue and every topic in the broker. Using the simple authorization plug-in, on the other hand, you can restrict access to specific destinations based on a user's group membership.

Configuring the simple authorization plug-in

To configure the simple authorization plug-in, add an **authorizationPlugin** element to the list of plug-ins in the broker's configuration, as shown in [Example 8.1, "Simple Authorization Plug-In Configuration"](#).

Example 8.1. Simple Authorization Plug-In Configuration

```
<beans>
  <broker ... >
    ...
  <plugins>
    ...
    <jaasAuthenticationPlugin configuration="karaf" />
    <authorizationPlugin>
      <map>
        <authorizationMap groupClass="org.apache.karaf.jaas.boot.principal.RolePrincipal">
          <authorizationEntries>
            <authorizationEntry queue=">"
              read="admins"
              write="admins"
              admin="admins" />
            <authorizationEntry queue="USERS.>"
              read="users"
              write="users"
              admin="users" />
            <authorizationEntry queue="GUEST.>"
              read="guests"
              write="guests,users"
              admin="guests,users" />
            <authorizationEntry topic=">"
              read="admins"
              write="admins"
              admin="admins" />
            <authorizationEntry topic="USERS.>"
              read="users"
              write="users"
              admin="users" />
          </authorizationEntries>
        </authorizationMap>
      </map>
    </authorizationPlugin>
  </plugins>
</beans>
```

```

    <authorizationEntry topic="GUEST.>"
      read="guests"
      write="guests,users"
      admin="guests,users" />
  </authorizationEntries>
  <tempDestinationAuthorizationEntry>
    <tempDestinationAuthorizationEntry
      read="admins"
      write="admins"
      admin="admins"/>
  </tempDestinationAuthorizationEntry>
</authorizationMap>
</map>
</authorizationPlugin>
</plugins>
...
</broker>

</beans>

```

The simple authorization plug-in is specified as a map of destination entries. The map is entered in the configuration using a **authorizationMap** element wrapped in a **map** element.

The authorization map is made up of two elements:

- **authorizationEntries**—a collection of **authorizationEntry** elements that define the permissions assigned to authorized users have for destinations whose name matches the selector
- **tempDestinationAuthorizationEntry**—defines the permissions assigned to authorized users have for temporary destinations

Integration with the Apache Karaf authentication module

The simple authorization plug-in was originally designed to work with the Apache ActiveMQ JAAS authentication module and is compatible with that module by default. In order to integrate with the Apache Karaf authentication module, however, it is necessary to set the **groupClass** attribute on the **authorizationMap** element.

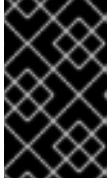
The **groupClass** attribute defines the type of the class that implements the role principal. For example, in order to reuse roles defined for the Apache Karaf JAAS authentication plug-in, you would need to set this property to **org.apache.karaf.jaas.boot.principal.RolePrincipal** (as shown in [Example 8.1, "Simple Authorization Plug-In Configuration"](#)).

The default value is **org.apache.activemq.jaas.GroupPrincipal**.

Named destinations

A named destination is an ordinary JMS queue or topic. The authorization entries for ordinary destinations are defined by the **authorizationEntry** element, which supports the following attributes:

- **queue** or **topic**—specifies the name of the queue or topic to which you are assigning permissions. The greater-than symbol, **>**, acts as a name segment wildcard. For example, an entry with, **queue="USERS.>**, would match any queue name beginning with the **USERS.** string.



IMPORTANT

In order for the `>` wildcard to match multiple segments, it must be preceded by the `.` segment-delimiter character. Hence, `USERS.>` matches any queue name beginning with `USERS.`, but `USERS>` *does not* match.

- **read**—specifies a comma-separated list of roles that have permission to *consume* messages from the matching destinations.
- **write**—specifies a comma-separated list of roles that have permission to *publish* messages to the matching destinations.
- **admin**—specifies a comma-separated list of roles that have permission to create destinations in the destination subtree.

Temporary destinations

A temporary destination is a special feature of JMS that enables you to create a queue for a particular network connection. The temporary destination exists only as long as the network connection remains open and, as soon as the connection is closed, the temporary destination is deleted on the server side. The original motivation for defining temporary destinations was to facilitate request-reply semantics on a destination, without having to define a dedicated reply destination.

Because temporary destinations have no name, there is only one entry in the map for them. This entry is specified using a **tempDestinationAuthorizationEntry** element that contains a **tempDestinationAuthorizationEntry** child element. The permissions set by this entry are for *all* temporary destinations. The attributes supported by the inner **tempDestinationAuthorizationEntry** element are:

- **read**—specifies a comma-separated list of roles that have permission to *consume* messages from all temporary destinations.
- **write**—specifies a comma-separated list of roles that have permission to *publish* messages to all temporary destinations.
- **admin**—specifies a comma-separated list of roles that have permission to create temporary destinations.

Advisory destinations

Advisory destinations are named destinations that Red Hat AMQ uses to communicate administrative information. Networks of brokers also use advisory destinations to coordinate between the brokers.

The authorization entries for advisory destinations are, like ordinary named destinations, defined by the **authorizationEntry** element. For advisory destinations, however, the **topic** attribute is always used and the name is always starts with **ActiveMQ.Advisory**.

Because advisory destinations are used by networks of brokers and a few other broker services, it is advised that full access permissions be granted for all of the advisory destinations by using an entry similar to [Example 8.2, “Setting Access Permissions for Advisory Destinations”](#).

Example 8.2. Setting Access Permissions for Advisory Destinations

```
<authorizationEntry topic="ActiveMQ.Advisory.>"
  read="guests,users"
```

```
write="guests,users"
admin="guests,users" />
```

If you have specific advisories that you want to secure, you can add individual entries for them.

8.2. CACHED LDAP AUTHORIZATION PLUG-IN

Overview

Using the cached LDAP authorization plug-in, you can configure a broker to retrieve its authorization data from an X.500 directory server. For better efficiency, this plug-in caches authorization data in the broker and provides support for updating the cached data at regular intervals.

Updating the cache

Two alternative mechanisms for updating the authorization cache are supported:

- *Push mechanism*—some LDAP directory server implementations support a *persistent search* feature, which enables applications to receive live updates from the LDAP server (push mechanism). By default, the cached LDAP authorization plug-in attempts to register with the LDAP server to receive these updates.
- *Pull mechanism*—if your LDAP directory server does not support live updates, you can configure the cached LDAP authorization plug-in to poll the LDAP server at regular intervals instead (pull mechanism). To enable the pull mechanism, you must set the **refreshInterval** property on the cached LDAP authorization plug-in.

Sample configuration

[Example 8.3, “Cached LDAP Authorization Plug-In Configuration”](#) shows an example of how to configure the cached LDAP authorization plug-in. The **authorizationPlugin** element must be added as a child of the **plugins** element.

Example 8.3. Cached LDAP Authorization Plug-In Configuration

```
<beans ... >
<broker ... >
...
<plugins>
...
<authorizationPlugin>
  <map>
    <cachedLDAPAuthorizationMap
      legacyGroupMapping="false"
      connectionURL="ldap://localhost:10389"
      connectionUsername="uid=admin,ou=system"
      connectionPassword="secret"
      queueSearchBase="ou=Queue,ou=Destination,ou=ActiveMQ,ou=system"
      topicSearchBase="ou=Topic,ou=Destination,ou=ActiveMQ,ou=system"
      tempSearchBase="ou=Temp,ou=Destination,ou=ActiveMQ,ou=system"
      refreshInterval="20000"
    />
  />
```

```

    </map>
  </authorizationPlugin>
</plugins>
...
</broker>
</beans>

```

Configuration properties

The cached LDAP authorization plug-in supports the following properties:

adminPermissionGroupSearchFilter

Specifies the filter used to search for *admin* permission groups. This filter is used when searching under the nodes specified by **queueSearchBase**, **topicSearchBase**, or **tempSearchBase**, to obtain the permission groups for queues, topics, or temporary destinations, respectively.

Default is **(cn=Admin)**.

authentication

The authentication method to use when connecting to the LDAP server.

Default is **simple**.

connectionPassword

The password that matches the DN from **connectionUsername**. In the directory server, the password is normally stored as a **userPassword** attribute in the corresponding directory entry.

Default is **secret**.

connectionProtocol

The connection protocol to use when connecting to the LDAP server.

Default is **s**.

connectionURL

Specifies the location of the directory server using an LDAP URL, **ldap://Host:Port**.

Default is **ldap://localhost:1024**.

connectionUsername

The DN of the user that opens the connection to the directory server.

Default is **uid=admin,ou=system**.

groupClass

Type of the class that implements the role principal. For example, in order to reuse roles defined for the Apache Karaf JAAS authentication plug-in, you would need to set this property to **org.apache.karaf.jaas.boot.principal.RolePrincipal**.

Default is **org.apache.activemq.jaas.GroupPrincipal**.

groupNameAttribute

Specifies which attribute of a permission group node is interpreted as the group name.

Default is **cn**.

groupObjectClass

Specifies the object class of the LDAP nodes used to store permission groups. Typical values are **groupOfNames** or **groupOfUniqueNames**.

Default is **groupOfNames**.

legacyGroupMapping

If **true**, specifies that the role members of a privilege group must be specified using just the Common Name RDN, **cn=CNValue**, of the role group; or if **false**, specifies that the role members of a privilege group must be specified using the full Distinguished Name.

Default is **true**.

permissionGroupMemberAttribute

Specifies which attribute of a permission group node defines a member. For example, if the **groupObjectClass** is set to **groupOfNames**, this attribute should usually be set to **member**. Alternatively, if the **groupObjectClass** is set to **groupOfUniqueNames**, this attribute should usually be set to **uniquemember**.

Default is **member**.

queueSearchBase

The base DN of queue authorization entries.

Default is **ou=Queue,ou=Destination,ou=ActiveMQ,ou=system**.

readPermissionGroupSearchFilter

Specifies the filter used to search for *read* permission groups. This filter is used when searching under the nodes specified by **queueSearchBase**, **topicSearchBase**, or **tempSearchBase**, to obtain the permission groups for queues, topics, or temporary destinations, respectively.

Default is **(cn=Read)**.

refreshDisabled

If **true**, disables cache refreshing.

Default is **false**.

refreshInterval

Time interval between refreshes of the cache, expressed in milliseconds (where the cache is refreshed by pulling data from the LDAP server). The special value, **-1**, disables the pull mechanism for refreshing the cache (but does not affect the *push* mechanism, if the LDAP server supports it).

Default is **-1**.

tempSearchBase

The base DN of authorization entries for temporary destinations.

Default is **ou=Temp,ou=Destination,ou=ActiveMQ,ou=system**.

topicSearchBase

The base DN of topic authorization entries.

Default is **ou=Topic,ou=Destination,ou=ActiveMQ,ou=system**.

userNameAttribute

Specifies which attribute of a user node is interpreted as the username.

Default is **uid**.

userObjectClass

Specifies the object class of the LDAP nodes used to store users.

Default is **person**.

writePermissionGroupSearchFilter

Specifies the filter used to search for *write* permission groups. This filter is used when searching under the nodes specified by **queueSearchBase**, **topicSearchBase**, or **tempSearchBase**, to obtain the permission groups for queues, topics, or temporary destinations, respectively.

Default is **(cn=Write)**.

Authorization settings for different directory servers

The most significant differences between directory servers arise in connection with the object class settings in the cached LDAP authorization plug-in. The precise settings depend ultimately on the organisation of your DIT, but the following table gives an idea of the typical object class settings required for different directory servers:

Directory Server	Object Class Settings
389-DS Red Hat DS	<pre>userObjectClass="inetorgperson" groupObjectClass="groupOfUniqueNames" permissionGroupMemberAttribute="uniqueMember"</pre>
Apache DS	<pre>userObjectClass="person" groupObjectClass="groupOfNames" permissionGroupMemberAttribute="member"</pre>

8.3. LDAP AUTHORIZATION PLUG-IN

Overview

Using the LDAP authorization plug-in, you can configure a broker to retrieve its authorization data from an X.500 directory server. This plug-in does not support caching and contacts the LDAP server every time an authorization needs to be checked.

Configuring the LDAP authorization plug-in

To configure the LDAP authorization plug-in, add the **authorizationPlugin** element to the list of plug-ins in the broker configuration and configure it to use the **LDAPAuthorizationMap** authorization map, as shown in [Example 8.4, "LDAP Authorization Plug-In Configuration"](#).

Example 8.4. LDAP Authorization Plug-In Configuration

```
<beans ... >
  <broker ... >
    ...
    <plugins>
      ...
      <authorizationPlugin>
        <map>
          <bean id="LDAPAuthorizationMap"
class="org.apache.activemq.security.LDAPAuthorizationMap"
  xmlns="http://www.springframework.org/schema/beans">
    <property name="initialContextFactory" value="com.sun.jndi.LdapCtxFactory"/>
    <property name="connectionURL" value="ldap://localhost:10389"/>
    <property name="authentication" value="simple"/>
    <property name="connectionUsername" value="uid=admin,ou=system"/>
    <property name="connectionPassword" value="secret"/>
    <property name="connectionProtocol" value=""/>
    <property name="topicSearchMatchingFormat"
      value="cn={0},ou=Topic,ou=Destination,ou=ActiveMQ,ou=system"/>
    <property name="topicSearchSubtreeBool" value="true"/>
    <property name="queueSearchMatchingFormat"
      value="cn={0},ou=Queue,ou=Destination,ou=ActiveMQ,ou=system"/>
    <property name="queueSearchSubtreeBool" value="true"/>
    <property name="advisorySearchBase"
      value="cn=ActiveMQ.Advisory,ou=Topic,ou=Destination,ou=ActiveMQ,ou=system"/>
    <property name="tempSearchBase"
      value="cn=ActiveMQ.Temp,ou=Topic,ou=Destination,ou=ActiveMQ,ou=system"/>
    <property name="adminBase" value="(cn=admin)/>
    <property name="adminAttribute" value="member"/>
    <property name="readBase" value="(cn=read)/>
    <property name="readAttribute" value="member"/>
    <property name="writeBase" value="(cn=write)/>
    <property name="writeAttribute" value="member"/>
          </bean>
        </map>
      </authorizationPlugin>
    </plugins>
    ...
  </broker>
</beans>
```

LDAP authorization plug-in properties

The LDAP authorization plug-in supports the following properties:

initialContextFactory

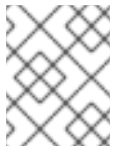
Must always be set to **com.sun.jndi.ldap.LdapCtxFactory**.

connectionURL

Specify the location of the directory server using an ldap URL, **ldap://Host:Port**. You can optionally qualify this URL, by adding a forward slash, */*, followed by the DN of a particular node in the directory tree. For example, **ldap://ldapserver:10389/ou=system**.

authentication

Specifies the authentication method used when binding to the LDAP server. Can take either of the values, **simple** (username and password) or **none** (anonymous).



NOTE

Simple Authentication and Security Layer (SASL) authentication is currently *not* supported.

connectionUsername

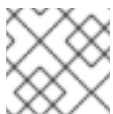
The DN of the user that opens the connection to the directory server. For example, **uid=admin,ou=system**.

connectionPassword

The password that matches the DN from **connectionUsername**. In the directory server, in the DIT, the password is normally stored as a **userPassword** attribute in the corresponding directory entry.

connectionProtocol

Currently, the only supported value is a blank string. In future, this option will allow you to select the Secure Socket Layer (SSL) for the connection to the directory server.



NOTE

This option *must* be set explicitly to an empty string, because it has no default value.

topicSearchMatchingFormat

Specifies the DN of the node whose children provide the permissions for the current topic. Before passing to the LDAP search operation, the string value you provide here is subjected to *string substitution*, as implemented by the **java.text.MessageFormat** class. Essentially, this means that the special string, **{0}**, is substituted by the name of the current topic.

For example, if this property is set to **cn={0},ou=Topic,ou=Destination,ou=ActiveMQ,ou=system** and the current topic is **TEST.FOO**, the DN becomes **cn=TEST.FOO,ou=Topic,ou=Destination,ou=ActiveMQ,ou=system**.

topicSearchSubtreeBool

Specify the search depth for permission entries (admin, read or write entries), relative to the node specified by **topicSearchMatchingFormat**. This option can take boolean values, as follows:

- **false**—(*default*) try to match one of the child entries of the **topicSearchMatchingFormat** node (maps to **javax.naming.directory.SearchControls.ONELEVEL_SCOPE**).
- **true**—try to match *any* entry belonging to the subtree of the **topicSearchMatchingFormat** node (maps to **javax.naming.directory.SearchControls.SUBTREE_SCOPE**).

queueSearchMatchingFormat

Specifies the DN of the node whose children provide the permissions for the current queue. The special string, **{0}**, is substituted by the name of the current queue.

For example, if this property is set to **cn={0},ou=Queue,ou=Destination,ou=ActiveMQ,ou=system** and the current queue is **TEST.FOO**, the DN becomes **cn=TEST.FOO,ou=Queue,ou=Destination,ou=ActiveMQ,ou=system**.

queueSearchSubtreeBool

Specify the search depth for permission entries (admin, read or write entries), relative to the node specified by **topicSearchMatchingFormat**. This option can take boolean values, as follows:

- **false**—(*default*) try to match one of the child entries of the **topicSearchMatchingFormat** node (maps to **javax.naming.directory.SearchControls.ONELEVEL_SCOPE**).
- **true**—try to match *any* entry belonging to the subtree of the **topicSearchMatchingFormat** node (maps to **javax.naming.directory.SearchControls.SUBTREE_SCOPE**).

advisorySearchBase

Specifies the DN of the node whose children provide the permissions for *all* advisory topics. In this case the DN is a literal value (that is, no string substitution is performed on the property value).

For example, a typical value of this property is

cn=ActiveMQ.Advisory,ou=Topic,ou=Destination,ou=ActiveMQ,ou=system.

tempSearchBase

Specifies the DN of the node whose children provide the permissions for *all* temporary queues and topics (apart from advisory topics). In this case the DN is a literal value (that is, no string substitution is performed on the property value).

For example, a typical value of this property is

cn=ActiveMQ.Temp,ou=Topic,ou=Destination,ou=ActiveMQ,ou=system.

adminBase

Specifies an LDAP search filter, which is used when looking up the *admin permissions* for any kind of queue or topic. The search filter attempts to match one of the children (or descendants, if **SUBTREE_SCOPE** is enabled) of the queue or topic node.

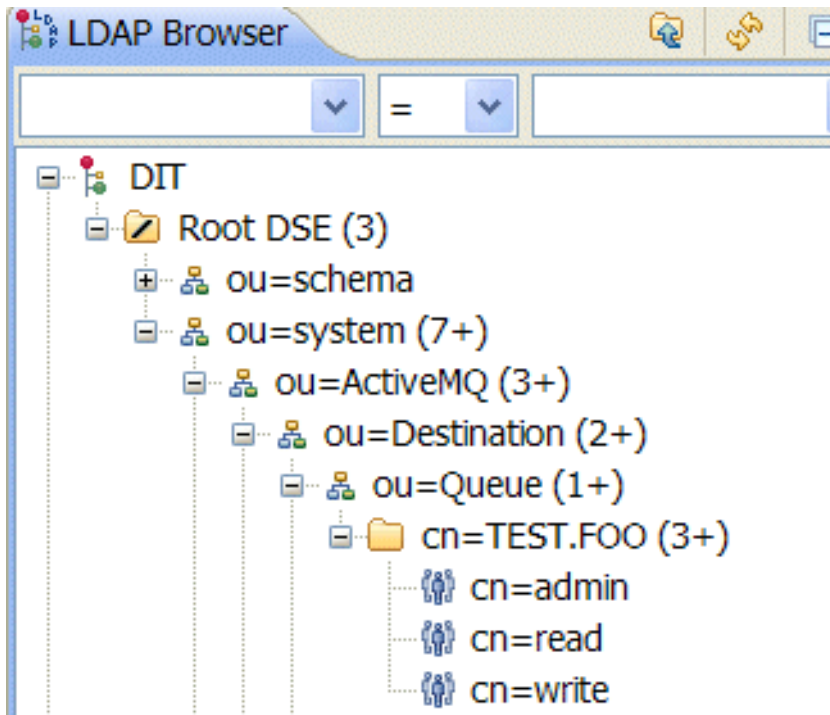
For example, if this property is set to **(cn=admin)**, it will match any child whose **cn** attribute is set to **admin**.

adminAttribute

Specifies an attribute of the node matched by **adminBase**, whose value is the DN of a role/group that has admin permissions.

For example, consider a **cn=admin** node that is a child of the node,

cn=TEST.FOO,ou=Queue,ou=Destination,ou=ActiveMQ,ou=system, as shown:



The **cn=admin** node might typically have some attributes, as follows:

Attribute Description	Value
<i>objectClass</i>	<i>groupOfNames (structural)</i>
<i>objectClass</i>	<i>top (abstract)</i>
cn	admin
member	cn=admins
member	cn=users

If you now set the **adminAttribute** property to **member**, the authorization plug-in grants admin privileges over the **TEST.FOO** queue to the **cn=admins** group and the **cn=users** group.

readBase

Specifies an LDAP search filter, which is used when looking up the *read permissions* for any kind of queue or topic. The search filter attempts to match one of the children (or descendants, if **SUBTREE_SCOPE** is enabled) of the queue or topic node.

For example, if this property is set to (**cn=read**), it will match any child whose **cn** attribute is set to **read**.

readAttribute

Specifies an attribute of the node matched by **readBase**, whose value is the DN of a role/group that has read permissions.

writeBase

Specifies an LDAP search filter, which is used when looking up the *write permissions* for any kind of queue or topic. The search filter attempts to match one of the children (or descendants, if **SUBTREE_SCOPE** is enabled) of the queue or topic node.

For example, if this property is set to (**cn=write**), it will match any child whose **cn** attribute is set to **write**.

writeAttribute

Specifies an attribute of the node matched by **writeBase**, whose value is the DN of a role/group that has write permissions.

8.4. PROGRAMMING MESSAGE-LEVEL AUTHORIZATION

Overview

In the preceding examples, the authorization step is performed at the time of connection creation and access is applied at the *destination* level of granularity. That is, the authorization step grants or denies access to particular queues or topics. It is conceivable, though, that in some systems you might want to grant or deny access at the level of individual *messages*, rather than at the level of destinations. For example, you might want to grant permission to all users to read from a certain queue, but some messages published to this queue should be accessible to administrators only.

You can achieve message-level authorization by configuring a *message authorization policy* in the broker configuration file. To implement this policy, you need to write some Java code.

Implement the MessageAuthorizationPolicy interface

[Example 8.5, "Implementation of MessageAuthorizationPolicy"](#) shows an example of a message authorization policy that allows messages from the **WebServer** application to reach only the **admin** user, with all other users blocked from reading these messages. This example presupposes that the **WebServer** application is configured to set the **JMSXAppID** property in the message's JMS header.

Example 8.5. Implementation of MessageAuthorizationPolicy

```
package com.acme;
...

public class MsgAuthzPolicy implements MessageAuthorizationPolicy {

    public boolean isAllowedToConsume(ConnectionContext context, Message message)
    {
        if (message.getProperty("JMSXAppID").equals("WebServer")) {
            if (context.getUserName().equals("admin")) {
                return true;
            }
            else {
                return false;
            }
        }
        return true;
    }
}
```

The `org.apache.activemq.broker.ConnectionContext` class stores details of the current client connection and the `org.apache.activemq.command.Message` class is essentially an implementation of the standard `javax.jms.Message` interface.

To install the message authorization policy, compile the preceding code, package it as a JAR file, and drop the JAR file into the `$ACTIVEMQ_HOME/lib` directory.

Configure the `messageAuthorizationPolicy` element

To configure the broker to install the message authorization policy from [Example 8.5, "Implementation of `MessageAuthorizationPolicy`"](#), add the following lines to the broker configuration file, `etc/activemq.xml`, inside the `broker` element:

```
<broker>
...
<messageAuthorizationPolicy>
  <bean class="com.acme.MsgAuthzPolicy"
    xmlns="http://www.springframework.org/schema/beans"/>
</messageAuthorizationPolicy>
...
</broker>
```

CHAPTER 9. LDAP AUTHENTICATION TUTORIAL

Abstract

This tutorial explains how to set up an X.500 directory server and configure the OSGi container to use LDAP authentication. For more detailed documentation on the LDAP login module, see also [Section 2.1.7, “JAAS LDAP Login Module”](#).

9.1. TUTORIAL OVERVIEW

Goals

In this tutorial you will:

- Install 389 Directory Server
- Add user entries to the LDAP server
- Add groups to manage security roles
- Configure AMQ to use LDAP authentication
- Configure AMQ to use roles for authorization
- Configure SSL/TLS connections to the LDAP server

9.2. SET-UP A DIRECTORY SERVER AND CONSOLE

Overview

This stage of the tutorial explains how to install the X.500 directory server and the management console from the Fedora [389 Directory Server](#) project. If you already have access to a 389 Directory Server instance, you can skip the instructions for installing the 389 Directory Server and install the 389 Management Console instead.

Prerequisites

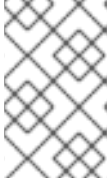
If you are installing on a Red Hat Enterprise Linux platform, you must first install the [Extra Packages for Enterprise Linux \(EPEL\)](#). See the installation notes under [RHEL/Cent OS/ EPEL \(RHEL 6, RHEL 7, Cent OS 6, Cent OS 7\)](#) on the [fedoraproject.org](#) site.

Install 389 Directory Server

If you do not have access to an existing *389 Directory Server* instance, you can install *389 Directory Server* on your local machine, as follows:

1. On Red Hat Enterprise Linux and Fedora platforms, use the standard **yum** package management utility to install *389 Directory Server*. Enter the following command at a command prompt (you must have administrator privileges on your machine):

```
sudo yum install 389-ds
```


**NOTE**

The required **389-ds** and **389-console** RPM packages are available for Fedora, RHEL6+EPEL, and CentOS7+EPEL platforms. At the time of writing, the **389-console** package is not yet available for RHEL 7.

2. After installing the 389 directory server packages, enter the following command to configure the directory server:

```
sudo setup-ds-admin.pl
```

The script is interactive and prompts you to provide the basic configuration settings for the 389 directory server. When the script is complete, it automatically launches the 389 directory server in the background.

3. For more details about how to install *389 Directory Server*, see the [Download](#) page.

Install 389 Management Console

If you already have access to a *389 Directory Server* instance, you only need to install the 389 Management Console, which enables you to log in and manage the server remotely. You can install the 389 Management Console, as follows:

- *On Red Hat Enterprise Linux and Fedora platforms* –use the standard **yum** package management utility to install the 389 Management Console. Enter the following command at a command prompt (you must have administrator privileges on your machine):

```
sudo yum install 389-console
```

- *On Windows platforms*—see the [Windows Console](#) download instructions from fedoraproject.org.

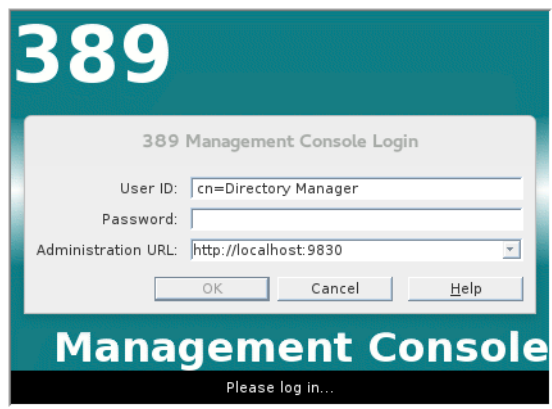
Connect the console to the server

To connect the 389 Directory Server Console to the LDAP server:

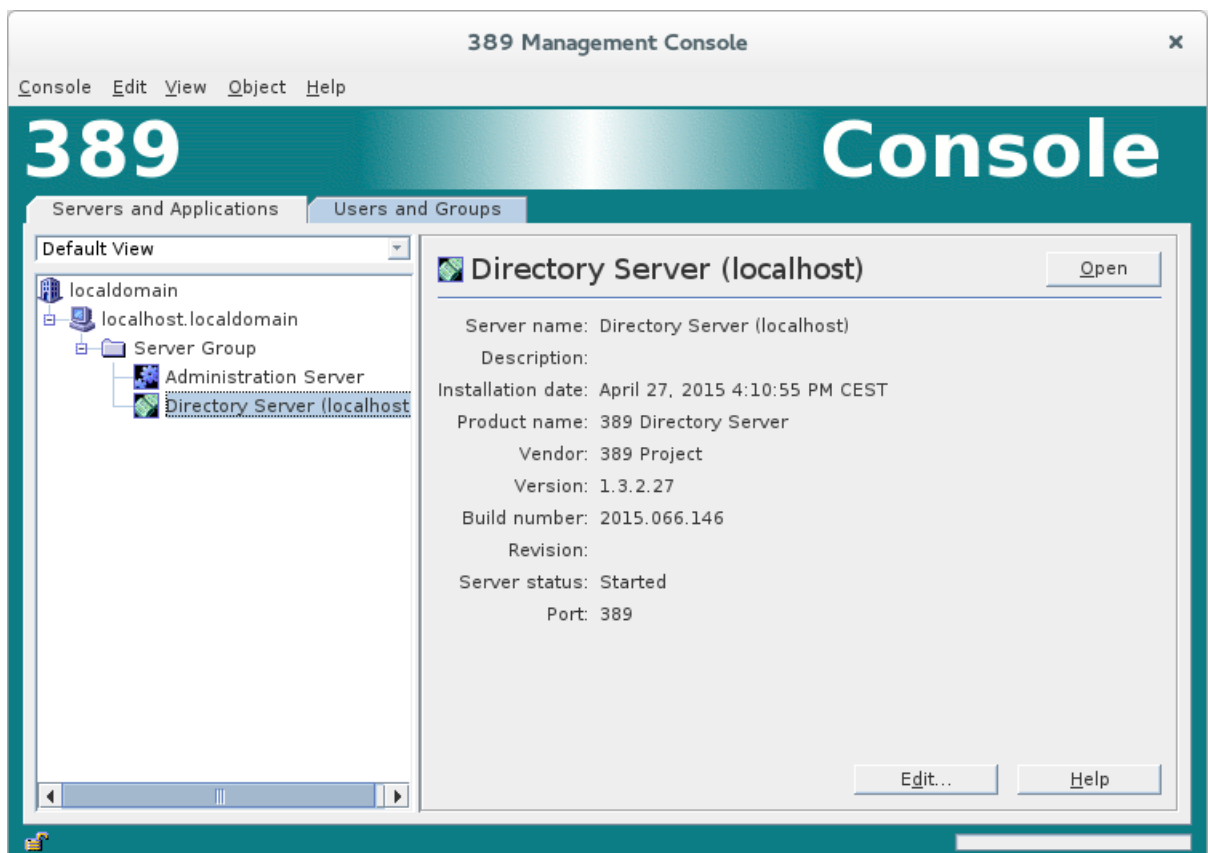
1. Enter the following command to start up the 389 Management Console:

```
389-console
```

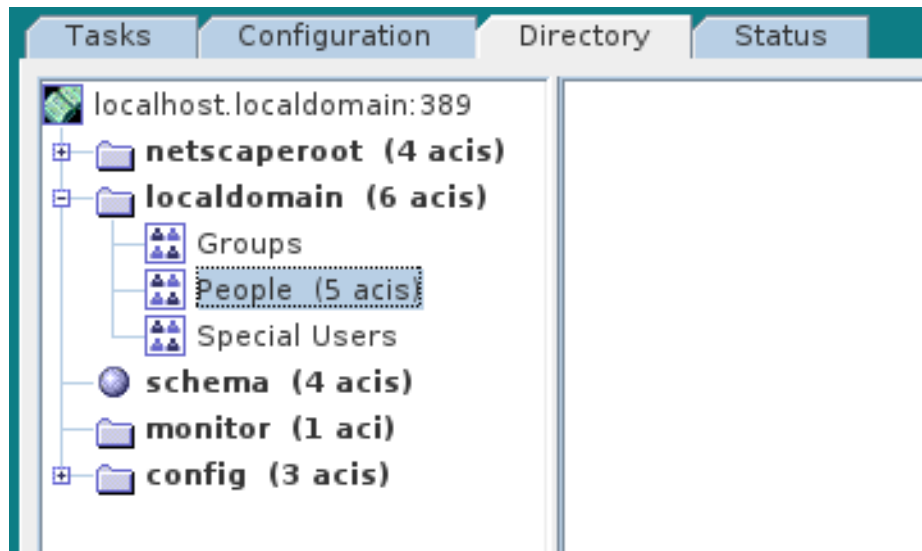
2. A login dialog appears. Fill in the LDAP login credentials in the **User ID** and **Password** fields, and customize the hostname in the **Administration URL** field to connect to your 389 management server instance (port **9830** is the default port for the 389 management server instance).



3. The 389 Management Console window appears. Select the **Servers and Applications** tab.
4. In the left-hand pane, drill down to the **Directory Server** icon.



5. Select the **Directory Server** icon in the left-hand pane and click **Open**, to open the **389 Directory Server Console**.
6. In the **389 Directory Server Console**, click the **Directory** tab, to view the Directory Information Tree (DIT).
7. Expand the root node, **YourDomain** (usually named after a hostname, and shown as **localdomain** in the following screenshot), to view the DIT.



9.3. ADD USER ENTRIES TO THE DIRECTORY SERVER

Overview

The basic prerequisite for using LDAP authentication with the OSGi container is to have an X.500 directory server running and configured with a collection of user entries. For many use cases, you will also want to configure a number of groups to manage user roles.

Alternative to adding user entries

If you already have user entries and groups defined in your LDAP server, you might prefer to map the existing LDAP groups to JAAS roles using the **roles.mapping** property in the **LDAPLoginModule** configuration, instead of creating new entries. For details, see [Section 2.1.7, "JAAS LDAP Login Module"](#).

Goals

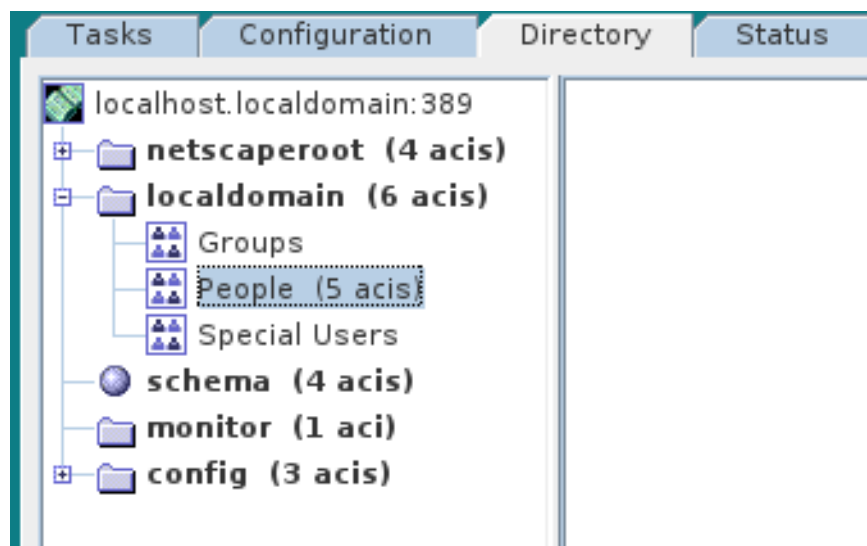
In this portion of the tutorial you will

- [Add three user entries to the LDAP server](#)
- [Add four groups to the LDAP server](#)

Adding user entries

Perform the following steps to add user entries to the directory server:

1. Ensure that the LDAP server and console are running. See [Section 9.2, "Set-up a Directory Server and Console"](#).
2. In the **Directory Server Console**, click on the **Directory** tab, and drill down to the **People** node, under the **YourDomain** node (where **YourDomain** is shown as **localdomain** in the following screenshots).



3. Right-click the **People** node, and select **New** → **User** from the context menu, to open the **Create New User** dialog.
4. Select the **User** tab in the left-hand pane of the **Create New User** dialog.
5. Fill in the fields of the **User** tab, as follows:
 - a. Set the **First Name** field to **John**.
 - b. Set the **Last Name** field to **Doe**.
 - c. Set the **User ID** field to **jdoe**.
 - d. Enter the password, **secret**, in the **Password** field.
 - e. Enter the password, **secret**, in the **Confirm Password** field.

Create New User

Phone:
Fax:

User
Languages
NT User
Posix User
Account

* First Name: John
* Last Name: Doe
* Common Name(s): John Doe
User ID: jdoe
Password:
Confirm Password:
E-Mail: (e.g., user@company.com)
Phone:
Fax:

* Indicates a required field

Advanced... OK Cancel Help

6. Click **OK**.
7. Add a user **Jane Doe** by following [Step 3](#) to [Step 6](#).

In [Step 5.e](#), use **janedoe** for the new user's **User ID** and use the password, **secret**, for the password fields.

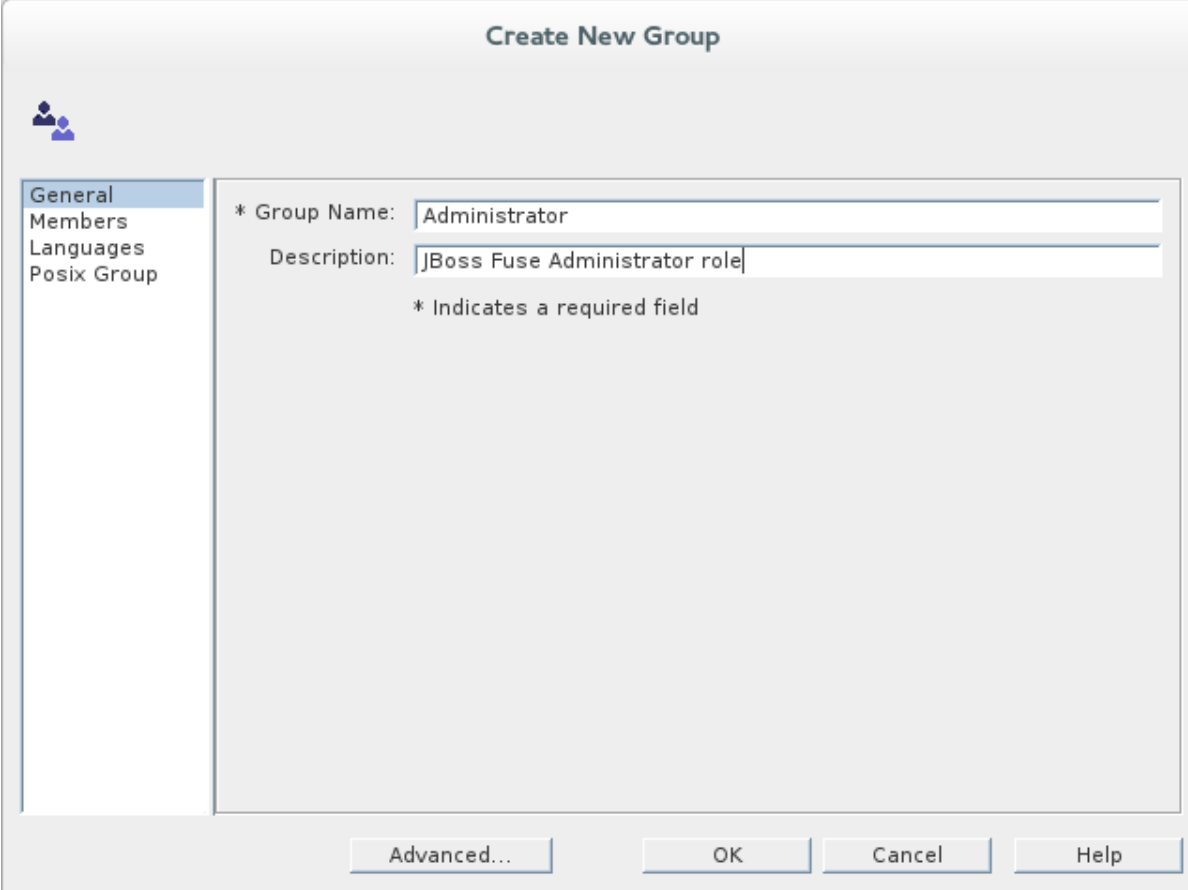
8. Add a user **Camel Rider** by following [Step 3](#) to [Step 6](#).

In [Step 5.e](#), use **crider** for the new user's **User ID** and use the password, **secret**, for the password fields.

Adding groups for the roles

To add the groups that define the roles:

1. In the **Directory** tab of the **Directory Server Console**, drill down to the **Groups** node, under the **YourDomain** node.
2. Right-click the **Groups** node, and select **New** → **Group** from the context menu, to open the **Create New Group** dialog.
3. Select the **General** tab in the left-hand pane of the **Create New Group** dialog.
4. Fill in the fields of the **General** tab, as follows:
 - a. Set the **Group Name** field to **Administrator**.
 - b. Optionally, enter a description in the **Description** field.



The screenshot shows the 'Create New Group' dialog box with the 'General' tab selected. The 'Group Name' field contains 'Administrator' and the 'Description' field contains 'JBoss Fuse Administrator role'. A note below the description states '* Indicates a required field'. The left-hand pane shows 'General', 'Members', 'Languages', and 'Posix Group' tabs. The bottom of the dialog has buttons for 'Advanced...', 'OK', 'Cancel', and 'Help'.

Create New Group

General
Members
Languages
Posix Group

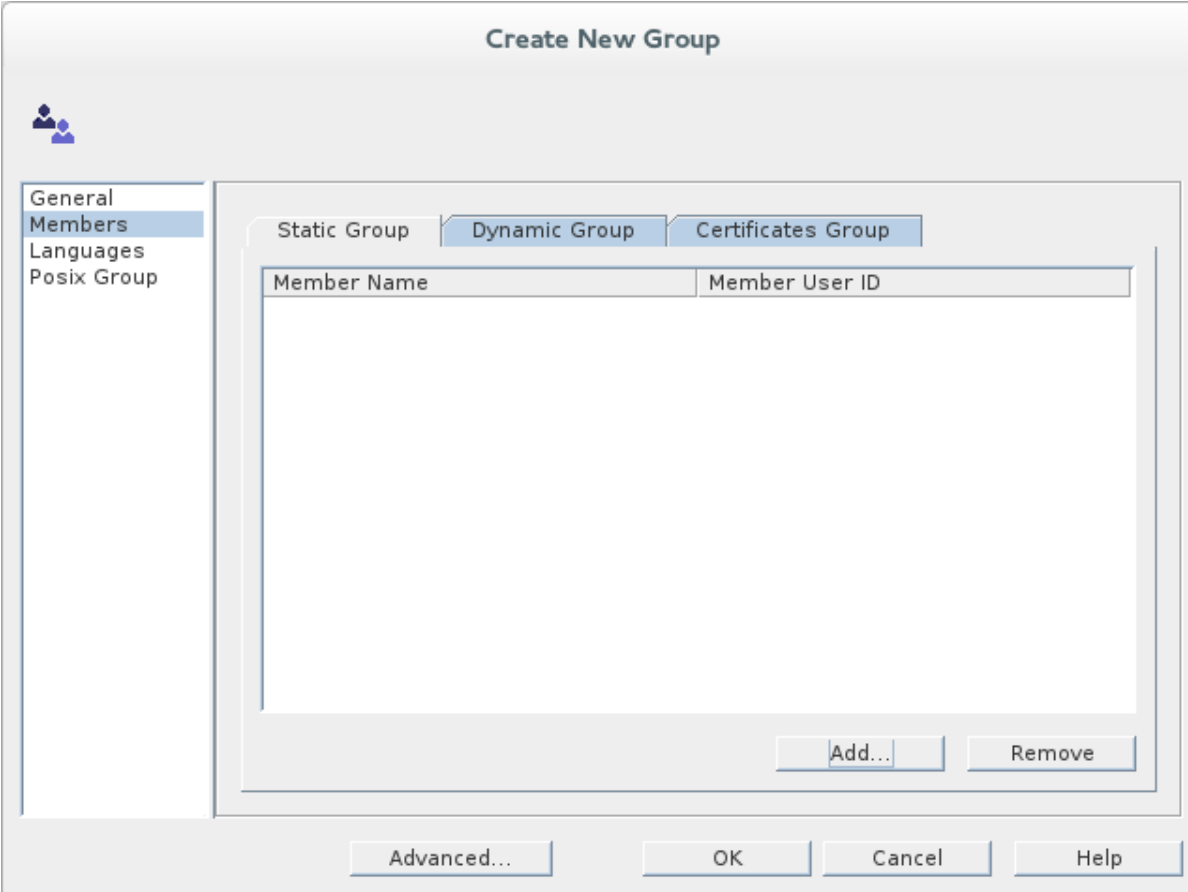
* Group Name: Administrator

Description: JBoss Fuse Administrator role

* Indicates a required field

Advanced... OK Cancel Help

5. Select the **Members** tab in the left-hand pane of the **Create New Group** dialog.



The screenshot shows the 'Create New Group' dialog box with the 'Members' tab selected. The 'Static Group', 'Dynamic Group', and 'Certificates Group' tabs are visible. The 'Member Name' and 'Member User ID' fields are empty. The 'Add...' and 'Remove' buttons are at the bottom right of the member list area. The left-hand pane shows 'General', 'Members', 'Languages', and 'Posix Group' tabs. The bottom of the dialog has buttons for 'Advanced...', 'OK', 'Cancel', and 'Help'.

Create New Group

General
Members
Languages
Posix Group

Static Group Dynamic Group Certificates Group

Member Name Member User ID

Add... Remove

Advanced... OK Cancel Help

6. Click **Add** to open the **Search users and groups** dialog.

- In the **Search** field, select **Users** from the drop-down menu, and click the **Search** button.

Search users and groups

Start searching from: ldap://localhost.localdomain:389/dc=localdomain

Search

for

Name	User ID	E-Mail	Phone
John Doe	jdoe		
Jane Doe	janedoe		
Camel Rider	crider		

- From the list of users that is now displayed, select **John Doe**.
- Click **OK**, to close the **Search users and groups** dialog.
- Click **OK**, to close the **Create New Group** dialog.
- Add a **Deployer** role by following [Step 2](#) to [Step 10](#).
 - In [Step 4](#), enter **Deployer** in the **Group Name** field.
 - In [Step 8](#), select **Jane Doe**.
- Add a **Monitor** role by following [Step 2](#) to [Step 10](#).
 - In [Step 4](#), enter **Monitor** in the **Group Name** field.
 - In [Step 8](#), select **Camel Rider**.

9.4. ENABLE LDAP AUTHENTICATION IN THE OSGI CONTAINER

Overview

This section explains how to configure an LDAP realm in the OSGi container. The new realm overrides the default **karaf** realm, so that the container authenticates credentials based on user entries stored in the X.500 directory server.

References

More detailed documentation is available on LDAP authentication, as follows:

- LDAPLoginModule options*—are described in detail in [Section 2.1.7, “JAAS LDAP Login Module”](#).
- Configurations for other directory servers*—this tutorial covers only [389-DS](#). For details of how to configure other directory servers, such as Microsoft Active Directory, see [the section called “Filter settings for different directory servers”](#).

Procedure for standalone OSGi container

To enable LDAP authentication in a standalone OSGi container:

1. Ensure that the X.500 directory server is running.
2. Start Red Hat AMQ by entering the following command in a terminal window:

```
./bin/amq
```

3. Create a file called **ldap-module.xml**.
4. Copy [Example 9.1, "JAAS Realm for Standalone"](#) into **ldap-module.xml**.

Example 9.1. JAAS Realm for Standalone

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0">

  <jaas:config name="karaf" rank="200">
    <jaas:module flags="required"
      className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule">
      initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
      connection.url=ldap://Hostname:Port
      connection.username=cn=Directory Manager
      connection.password=LDAPPASSWORD
      connection.protocol=
      user.base.dn=ou=People,dc=localdomain
      user.filter=(&objectClass=inetOrgPerson)(uid=%u))
      user.search.subtree=true
      role.base.dn=ou=Groups,dc=localdomain
      role.name.attribute=cn
      role.filter=(uniquemember=%fqdn)
      role.search.subtree=true
      authentication=simple
    </jaas:module>
  </jaas:config>
</blueprint>
```

You must customize the following settings in the **ldap-module.xml** file:

connection.url

Set this URL to the actual location of your directory server instance. Normally, this URL has the format, **ldap://*Hostname:Port***. For example, the default port for the 389 Directory Server is IP port **389**.

connection.username

Specifies the username that is used to authenticate the connection to the directory server. For 389 Directory Server, the default is usually **cn=Directory Manager**.

connection.password

Specifies the password part of the credentials for connecting to the directory server.

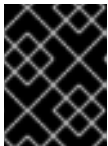
authentication

You can specify either of the following alternatives for the authentication protocol:

- **simple** implies that user credentials are supplied and you are obliged to set the **connection.username** and **connection.password** options in this case.
- **none** implies that authentication is *not* performed. There is no need to set the **connection.username** and **connection.password** options in this case.

This login module creates a JAAS realm called **karaf**, which is the same name as the default JAAS realm used by AMQ. By redefining this realm with a **rank** attribute value greater than **0**, it overrides the standard **karaf** realm which has the rank **0** (but note that in the context of Fabric, the default **karaf** realm has a rank of **99**, so you need to define a new realm with rank **100** or greater to override the default realm in a fabric).

For more details about how to configure AMQ to use LDAP, see [Section 2.1.7, “JAAS LDAP Login Module”](#).



IMPORTANT

When setting the JAAS properties above, do *not* enclose the property values in double quotes.

5. To deploy the new LDAP module, copy the **ldap-module.xml** into the AMQ **deploy/** directory.

The LDAP module is automatically activated.



NOTE

Subsequently, if you need to undeploy the LDAP module, you can do so by deleting the **ldap-module.xml** file from the **deploy/** directory *while the Karaf container is running*.

Procedure for a Fabric

To enable LDAP authentication in a Fabric (affecting all of the containers in the current fabric):

1. Ensure that the X.500 directory server is running.
2. If your local Fabric container is not already running, start it now, by entering the following command in a terminal window:

```
./bin/amq
```

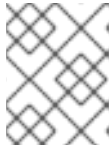


NOTE

If the Fabric container you want to connect to is running on a remote host, you can connect to it using the **client** command-line utility in the **InstallDir/bin** directory.

3. Create a new version of the Fabric profile data, by entering the following console command:

```
JBossFuse:karaf@root> version-create
Created version: 1.1 as copy of: 1.0
```



NOTE

In effect, this command creates a new branch named **1.1** in the Git repository underlying the ZooKeeper registry.

4. Create the new profile resource, **ldap-module.xml** (a Blueprint configuration file), in version **1.1** of the **default** profile, as follows:

```
JBossFuse:karaf@root> profile-edit --resource ldap-module.xml default 1.1
```

The built-in profile editor opens automatically, which you can use to edit the contents of the **ldap-module.xml** resource.

5. Copy [Example 9.2, "JAAS Realm for Fabric"](#) into the **ldap-module.xml** resource, customizing the configuration properties, as necessary.

Example 9.2. JAAS Realm for Fabric

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0"
  xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.1.0">

  <command-bundle xmlns="http://karaf.apache.org/xmlns/shell/v1.0.0">
    <command name="jasyp/encrypt">
      <action class="io.fabric8.fabric.jaas.EncryptPasswordCommand" />
    </command>
  </command-bundle>

  <!-- AdminConfig property place holder for the org.apache.karaf.jaas -->
  <cm:property-placeholder persistent-id="io.fabric8.fabric.jaas"
    update-strategy="reload">
    <cm:default-properties>
      <cm:property name="encryption.name" value="" />
      <cm:property name="encryption.enabled" value="true" />
      <cm:property name="encryption.prefix" value="{CRYPT}" />
      <cm:property name="encryption.suffix" value="{CRYPT}" />
      <cm:property name="encryption.algorithm" value="MD5" />
      <cm:property name="encryption.encoding" value="hexadecimal" />
    </cm:default-properties>
  </cm:property-placeholder>

  <jaas:config name="karaf" rank="200">
    <jaas:module className="io.fabric8.jaas.ZookeeperLoginModule"
      flags="sufficient">
      path = /fabric/authentication/users
      encryption.name = ${encryption.name}
      encryption.enabled = ${encryption.enabled}
```

```

encryption.prefix = ${encryption.prefix}
encryption.suffix = ${encryption.suffix}
encryption.algorithm = ${encryption.algorithm}
encryption.encoding = ${encryption.encoding}
</jaas:module>
<jaas:module className="org.apache.karaf.jaas.modules.Ldap.LDAPLoginModule"
    flags="sufficient">
    initialContextFactory=com.sun.jndi.Ldap.LdapCtxFactory
    connection.url=ldap://Hostname:Port
    connection.username=cn=Directory Manager
    connection.password=LDAPPASSWORD
    connection.protocol=
    user.base.dn=ou=People,dc=localdomain
    user.filter=(&objectClass=inetOrgPerson)(uid=%u)
    user.search.subtree=true
    role.base.dn=ou=Groups,dc=localdomain
    role.name.attribute=cn
    role.filter=(uniquemember=%fqdn)
    role.search.subtree=true
    authentication=simple
</jaas:module>
</jaas:config>

<!-- The Backing Engine Factory Service for the ZookeeperLoginModule -->
<service interface="org.apache.karaf.jaas.modules.BackingEngineFactory">
    <bean class="io.fabric8.jaas.ZookeeperBackingEngineFactory" />
</service>
</blueprint>

```

You must customize the following settings in the **ldap-module.xml** file:

connection.url

Set this URL to the actual location of your directory server instance. Normally, this URL has the format, **ldap://*Hostname:Port***. You must be sure to use a hostname that is accessible to *all* of the containers in the fabric (hence, you cannot use **localhost** as the hostname here). The default port for the 389 Directory Server is IP port **389**.

connection.username

Specifies the username that is used to authenticate the connection to the directory server. For 389 Directory Server, the default is usually **cn=Directory Manager**.

connection.password

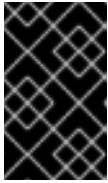
Specifies the password part of the credentials for connecting to the directory server.

authentication

You can specify either of the following alternatives for the authentication protocol:

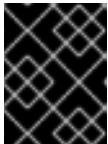
- **simple** implies that user credentials are supplied and you are obliged to set the **connection.username** and **connection.password** options in this case.
- **none** implies that authentication is *not* performed. There is no need to set the **connection.username** and **connection.password** options in this case.

This login module creates a JAAS realm called **karaf**, which is the same name as the default JAAS realm used by Red Hat AMQ. By redefining this realm with a **rank** of **200**, it overrides all of the previously installed **karaf** realms (in the context of Fabric, you need to override the default **ZookeeperLoginModule**, which has a rank of **99**).



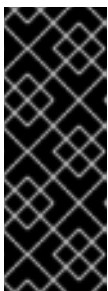
IMPORTANT

Pay particular attention to the value of the **rank** to ensure that it is higher than all previously installed **karaf** realms. If the **rank** is not sufficiently high, the new realm will not be used by the fabric.



IMPORTANT

When setting the JAAS properties above, do *not* enclose the property values in double quotes.



IMPORTANT

In a Fabric, the Zookeeper login module *must* be enabled, in addition to the LDAP login module. This is because Fabric uses the Zookeeper login module internally, to support authentication between ensemble servers. With the configuration shown here, Fabric tries to authenticate first of all against the Zookeeper login module and, if that step fails, it tries to authenticate against the LDAP login module.

6. Save and close the **ldap-module.xml** resource by typing Ctrl-S and Ctrl-X.
7. Edit the agent properties of version 1.1 of the **default** profile, adding an instruction to deploy the Blueprint resource file defined in the previous step. Enter the following console command:

```
JBossFuse:karaf@root> profile-edit default 1.1
```

The built-in profile editor opens automatically. Add the following line to the agent properties:

```
bundle.ldap-realm=blueprint:profile:ldap-module.xml
```

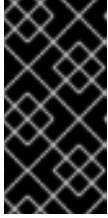
Save and close the agent properties by typing Ctrl-S and Ctrl-X.

8. The new LDAP realm is not activated, until you upgrade a container to use the new version, **1.1**. To activate LDAP on a *single* container (for example, on a container called **root**), enter the following console command:

```
JBossFuse:karaf@root> container-upgrade 1.1 root
```

To activate LDAP on *all* containers in the fabric, enter the following console command:

```
JBossFuse:karaf@root> container-upgrade --all 1.1
```



IMPORTANT

It is advisable to upgrade just a single container initially, to make sure that everything is working properly. This is particularly important, if you have only remote access to the fabric: if you upgrade all of the containers at once, you might not be able to reconnect to the fabric.

- To check that the LDAP realm is activated, enter the following console command:

```
JBossFuse:karaf@root> jaas-realms
Index Realm      Module Class
  1 karaf        org.apache.karaf.jaas.modules.ldap.LDAPLoginModule
```

If the output of this command lists the **ZookeeperLoginModule**, this means the LDAP realm is not yet activated. It might take a minute or so for activation of the LDAP realm to complete.

Test the LDAP authentication

Test the new LDAP realm by connecting to the running container using the AMQ **client** utility, as follows:

- Open a new command prompt.
- Change directory to the AMQ **InstallDir/bin** directory.
- Enter the following command to log on to the running container instance using the identity **jdoue**:

```
client -u jdoue -p secret
```

You should successfully log into the container's remote console. At the command console, type **jaas:** followed by the [Tab] key (to activate content completion):

```
JBossFuse:jdoue@root> jaas:
jaas:cancel      jaas:groupadd   jaas:groupcreate
jaas:groupdel   jaas:grouproleadd jaas:grouproledele
jaas:groups     jaas:manage    jaas:pending
jaas:realms     jaas:roleadd   jaas:roledel
jaas:update     jaas:useradd   jaas:userdel
jaas:users
```

You should see that **jdoue** has access to all of the **jaas** commands (which is consistent with the **Administrator** role).

- Log off the remote console by entering the **logout** command.
- Enter the following command to log on to the running container instance using the identity **janedoue**:

```
client -u janedoue -p secret
```

You should successfully log into the container's remote console. At the command console, type **jaas:** followed by the [Tab] key (to activate content completion):

```
JBossFuse:janedoue@root> jaas:
```

```
jaas:cancel      jaas:groupadd   jaas:groupcreate
jaas:groupdel   jaas:grouproleadd jaas:grouproledel
jaas:groups     jaas:manage    jaas:pending
jaas:realms    jaas:roleadd   jaas:roledel
jaas:useradd   jaas:userdel   jaas:users
```

You should see that **janedoe** has access to almost all of the **jaas** commands, except for **jaas:update** (which is consistent with the **Deployer** role).

- Log off the remote console by entering the **logout** command.
- Enter the following command to log on to the running container instance using the identity **crider**:

```
client -u crider -p secret
```

You should successfully log into the container's remote console. At the command console, type **jaas:** followed by the [Tab] key (to activate content completion):

```
JBossFuse:janedoe@root> jaas:
jaas:groupcreate jaas:groups jaas:realms
```

You should see that **crider** has access to only three of the **jaas** commands (which is consistent with the **Monitor** role).

- Log off the remote console by entering the **logout** command.

9.5. ADD BROKER AUTHORIZATION ENTRIES

Overview

Before enabling LDAP authorization in the broker, you need to create a suitable tree of entries in the directory server to represent permissions. You need to create the following kinds of entry:

Queue entries

Each queue entry has a Common Name (**cn**), which can be the name of a specific queue or a wildcard pattern that matches multiple queues. Under each queue entry, you must create sub-entries for the admin, read, and write permissions.

Topic entries

Each topic entry has a Common Name (**cn**), which can be the name of a specific topic or a wildcard pattern that matches multiple topics. Under each topic entry, you must create sub-entries for the admin, read, and write permissions.

Advisory topics entry

In particular, you must define one topic entry with the Common Name, **ActiveMQ.Advisory.\$**, which is a wildcard pattern that matches all advisory topics.

Temporary queues entry

A single **Temp** entry contains the admin, read, and write permissions that apply to *all* temporary queues.

Using wildcards in queue and topic entries

When setting the common name of queue and topic entries in the directory server, you can use any of the wildcards shown in [Table 9.1, “Destination Name Wildcards in LDAP”](#) to match one or more segments of a destination name.

Table 9.1. Destination Name Wildcards in LDAP

Wildcard	Description
.	Separates segments in a path name.
*	Matches any single segment in a path name.
\$	Matches any number of segments in a path name.

For example, the pattern, **FOO.***, will match **FOO.BAR**, but not **FOO.BAR.LONG**; whereas the pattern, **FOO.\$**, will match **FOO.BAR** and **FOO.BAR.LONG**.



NOTE

In the context of LDAP entries, the **\$** character is used instead of the usual **>** character to match multiple destination name segments.

Steps to add authorization entries

Perform the following steps to add authorization entries to the directory server:

1. The next few steps describe how to create the **ou=ActiveMQ** node.
 - a. Right-click the **YourDomain** node, and select **New → Organizational Unit** from the context menu. The **Create New Organizational Unit** dialog appears.
 - b. Select the **Unit** tab in the left-hand pane of the **Create New Organizational Unit** dialog.
 - c. Enter **ActiveMQ** in the **Name** field.
 - d. Click **OK**, to close the **Create New Organizational Unit** dialog.
2. The next few steps describe how to create the **ou=Destination** node.
 - a. Right-click on the **ActiveMQ** node and select **New → Organizational Unit** from the context menu. The **Create New Organizational Unit** dialog appears.
 - b. Select the **Unit** tab in the left-hand pane of the **Create New Organizational Unit** dialog.
 - c. Enter **Destination** in the **Name** field.
 - d. Click **OK**, to close the **Create New Organizational Unit** dialog.
3. In a similar manner to the preceding steps, by right-clicking on the **Destination** node and invoking the **New → Organizational Unit** context menu option, create the following **organisationalUnit** nodes as children of the **ou=Destination** node:

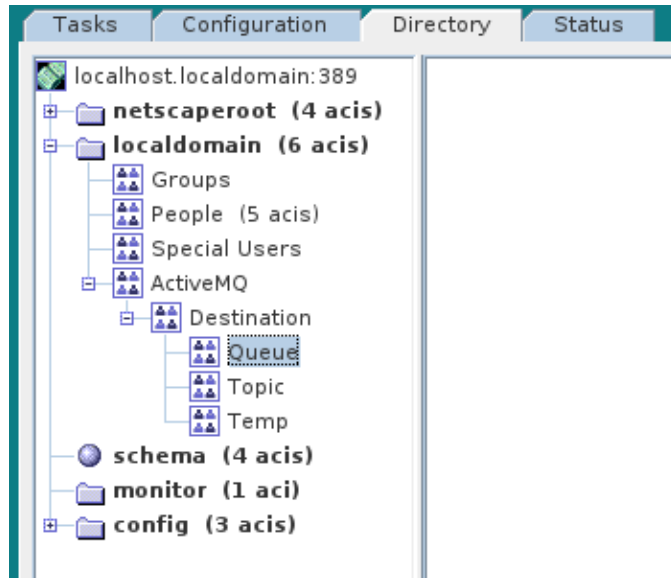
```

ou=Queue,ou=Destination,ou=ActiveMQ,dc=YourDomain
ou=Topic,ou=Destination,ou=ActiveMQ,dc=YourDomain
ou=Temp,ou=Destination,ou=ActiveMQ,dc=YourDomain

```

4. In the LDAP Browser window, you should now see the following tree:

Figure 9.1. DIT after Creating Destination, Queue, Topic and Temp Nodes



5. The next few steps describe how to create the following nodes:

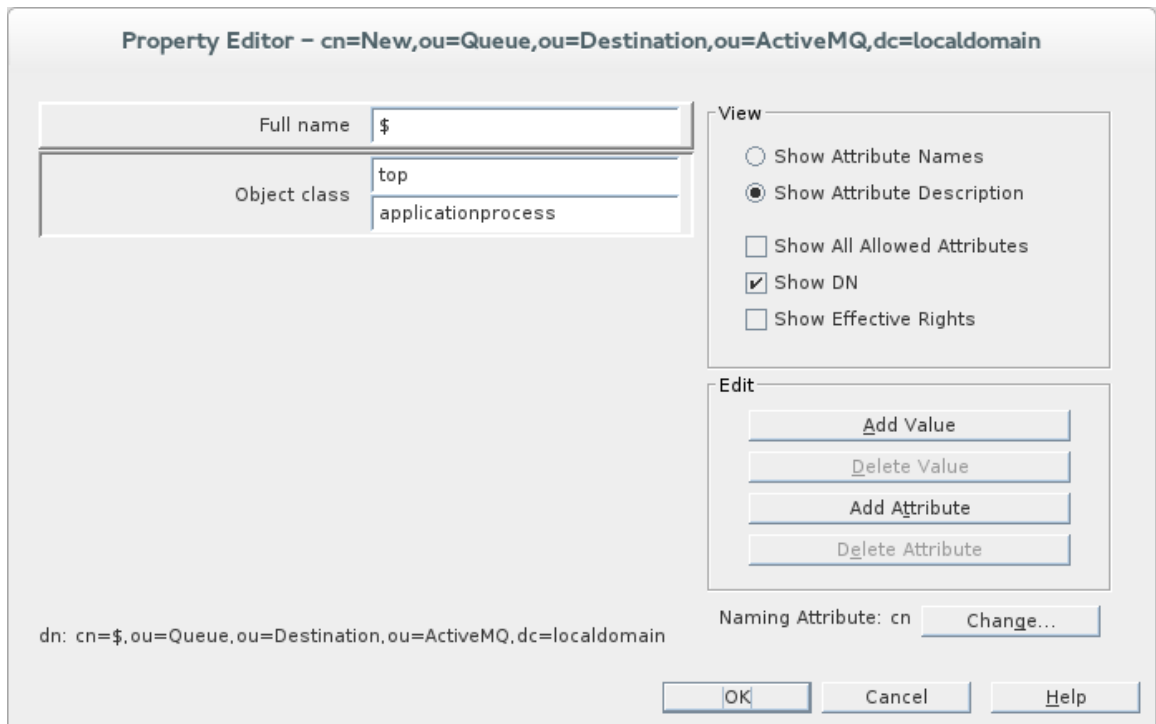
```

cn=$,ou=Queue,ou=Destination,ou=ActiveMQ,dc=YourDomain
cn=ActiveMQ.Advisory.$,ou=Topic,ou=Destination,ou=ActiveMQ,dc=YourDomain

```

These nodes represent name patterns that match queue names and topic names, respectively. The **cn=\$** queue node defines an entry that matches *all* queue names, so it can be used to define access rights for all queues. The **cn=ActiveMQ.Advisory.\$** node defines a topic entry that matches all advisory topics.

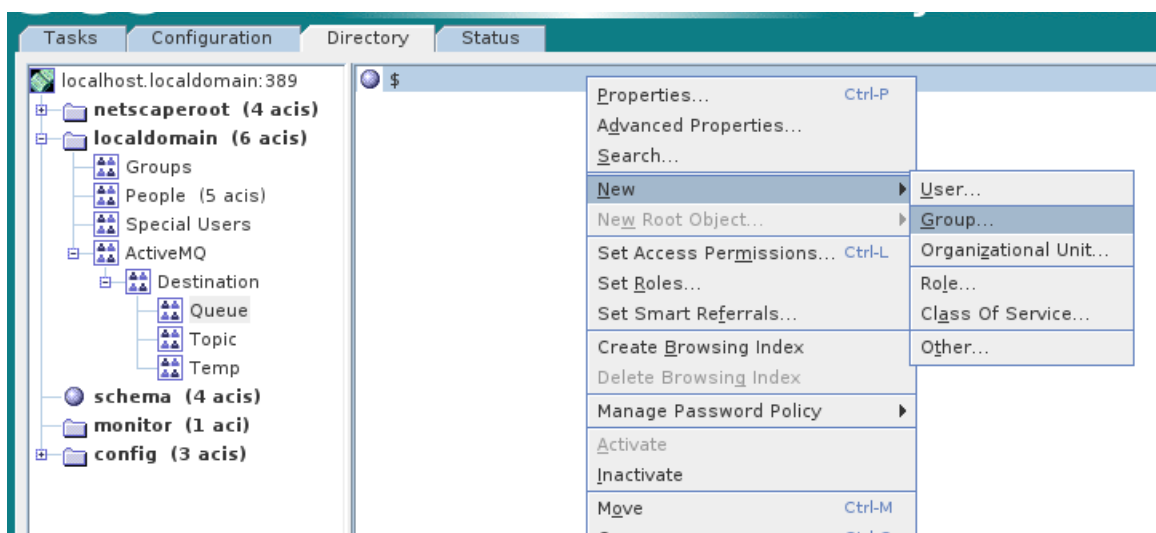
- a. Right-click on the **ou=Queue** node and select **New** → **Other**. The **New Object** dialog appears.
- b. Select **applicationprocess**. Click **OK**.
- c. The **Property Editor** dialog now appears. In the **Full name** field, enter **\$** (where **\$** represents the wildcard that matches any queue name). Click **OK**.



- d. In a similar manner to the preceding steps, by right-clicking on the **ou=Topic** node and selecting the **New → Other** context menu option, create the following **applicationProcess** node as a child of the **ou=Topic** node:

`cn=ActiveMQ.Advisory.$,ou=Topic,ou=Destination,ou=ActiveMQ,dc=YourDomain`

6. The next few steps describe how to create the permission group nodes, which represent **admin**, **read**, and **write** permissions, for the **ou=Queue** node.
- a. Right-click on the **cn=\$** node (initially depicted as a spherical icon in the console) and select **New → Group** from the context menu.



- b. The **Create New Group** dialog appears. Select the **General** tab in the left-hand pane of the **Create New Group** dialog.
- c. Set the **Group Name** field to **admin**.

Create New Group

* Group Name:
 Description:

* Indicates a required field

- d. Select the **Members** tab in the left-hand pane of the **Create New Group** dialog.

Create New Group

Static Group **Dynamic Group** Certificates Group

Member Name	Member User ID

- e. Click **Add** to open the **Search users and groups** dialog.
- f. In the **Search** field, select **Groups** from the drop-down menu, and click the **Search** button.
- g. From the list of groups that is now displayed, select **Administrator**.
- h. Click **OK**, to close the **Search users and groups** dialog.
- i. Click **OK**, to close the **Create New Group** dialog.
- j. In a similar manner to the preceding steps, by right-clicking on the **cn=\$** node and opening the **New** → **Group** dialog, create the following additional **groupOfUniqueNames** nodes as children of the **cn=\$** node:

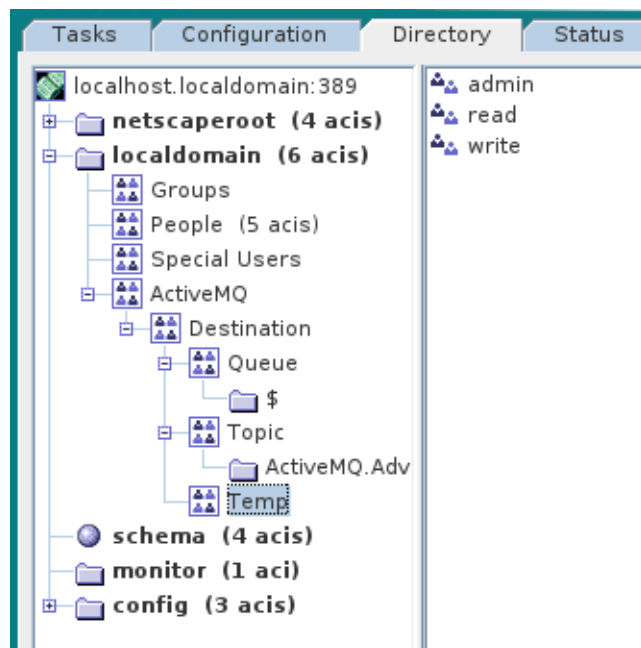
```
cn=read,cn=$,ou=Queue,ou=Destination,ou=ActiveMQ,dc= YourDomain
cn=write,cn=$,ou=Queue,ou=Destination,ou=ActiveMQ,dc= YourDomain
```

7. Copy the **cn=admin**, **cn=read**, and **cn=write** permission nodes and paste them as children of the **cn=ActiveMQ.Advisory.\$** node, as follows.

Using a combination of mouse and keyboard, select the three nodes, **cn=admin**, **cn=read**, and **cn=write**, and type **Ctrl-C** to copy them. Select the **cn=ActiveMQ.Advisory.\$** node and type **Ctrl-V** to paste the copied nodes as children.

8. Similarly, copy the **cn=admin**, **cn=read**, and **cn=write** permission nodes and paste them as children of the **ou=Temp** node.
9. In the LDAP Browser window, you should now see the following tree:

Figure 9.2. DIT after Creating Children of Queue, Topic and Temp Nodes



9.6. ENABLE LDAP AUTHORIZATION IN THE BROKER

Overview

This section explains how to enable LDAP authorization in the broker, so that the broker obtains its authorization data from the directory server.

Compatibility with Apache Karaf principals

To avoid unnecessary duplication of user data, this LDAP authorization example reuses the user and role data already created for the Apache Karaf JAAS authentication plug-in (as described in [Section 9.3, "Add User Entries to the Directory Server"](#)). This affects the broker's LDAP authorization plug-in configuration, as follows:

- When you create authorization entries in the LDAP server (as described in [Section 9.5, "Add Broker Authorization Entries"](#)), you must specify the *full DN* of the roles that are being authorized. This enables you to specify roles from *any* location in the LDAP tree (previously, the LDAP authorization plug-in could read roles only from a fixed location under the **ou=ActiveMQ,ou=system** node).
- To enable the use of full DN's when specifying roles, you must set the **legacyGroupMapping** property to **false** in the LDAP authorization plug-in (the default is **true**).
- Because the Apache Karaf roles are a different type than the roles natively supported by the LDAP authorization plug-in, you must also specify the type of the Karaf roles, by setting the **groupClass** property.

Enable broker LDAP authorization in a standalone OSGi container

Perform the following steps to enable broker LDAP authorization in a standalone OSGi container:

1. Shut down the AMQ container, if it is currently running. In the console window, enter the following command:

```
JBossA-MQ:karaf@root> shutdown
```

2. Make a backup copy of the broker configuration file, *InstallDir/etc/activemq.xml*.
3. Replace the LDAP authorization plug-in in the broker configuration. Open the broker configuration file, *InstallDir/etc/activemq.xml*, with a text editor and replace the default **authorizationMap** element by the **cachedLDAPAuthorizationMap** element, as follows:

```
<beans ...>
  <broker ...>
    ...
    <plugins>
      ...
      <!-- Check user credentials and get roles, using JAAS authentication plug-in -->
      <jaasAuthenticationPlugin configuration = "karaf"/>

      <!-- Check destination permissions, using authorization plug-in -->
      <authorizationPlugin>
        <map>
          <cachedLDAPAuthorizationMap
            connectionURL="ldap://Hostname:Port"
            connectionUsername="cn=Directory Manager"
            connectionPassword="LDAPPassword"
            queueSearchBase="ou=Queue,ou=Destination,ou=ActiveMQ,dc=YourDomain"
            topicSearchBase="ou=Topic,ou=Destination,ou=ActiveMQ,dc=YourDomain"
            tempSearchBase="ou=Temp,ou=Destination,ou=ActiveMQ,dc=YourDomain"
            groupObjectClass="groupOfUniqueNames"
            permissionGroupMemberAttribute="uniquemember"
            refreshInterval="300000"
            legacyGroupMapping="false"
            groupClass="org.apache.karaf.jaas.boot.principal.RolePrincipal"
          />
        </map>
      </authorizationPlugin>
    </plugins>
    ...
  </broker>
</beans>
```

You must customize the following settings in the **activemq.xml** file:

connectionURL

Set this URL to the actual location of your directory server instance. Normally, this URL has the format, **ldap://Hostname:Port**. For example, the default port for the 389 Directory Server is IP port **389**.

connectionUsername

Specifies the username that is used to authenticate the connection to the directory server. For 389 Directory Server, the default is usually **cn=Directory Manager**.

connectionPassword

Specifies the password part of the credentials for connecting to the directory server.

queueSearchBase

Replace **YourDomain** with the name of the root node on your directory server.

topicSearchBase

Replace **YourDomain** with the name of the root node on your directory server.

tempSearchBase

Replace **YourDomain** with the name of the root node on your directory server.

**NOTE**

For more details about the options available on the **cachedLDAPAuthorizationMap** element, see [Section 8.2, “Cached LDAP Authorization Plug-In”](#).

4. Ensure that the X.500 directory server is running. If necessary, manually restart the X.500 directory server—see [Section 9.2, “Set-up a Directory Server and Console”](#). If the server is not running, all broker connections will fail.
5. Restart the AMQ container. Open a new command prompt and start the broker by entering the following command:

```
amq
```

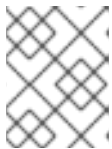
Enable broker LDAP authorization in a Fabric

Perform the following steps to enable broker LDAP authorization in a fabric:

1. Create a new version of the Fabric profile data, by entering the following console command:

```
JBossFuse:karaf@root> version-create  
Created version: 1.2 as copy of: 1.1
```

Where we have assumed that the current version is **1.1**.

**NOTE**

In effect, this command creates a new branch named **1.2** in the Git repository underlying the ZooKeeper registry.

2. Edit the **broker.xml** resource in version **1.2** of the **mq-base** profile, as follows:

```
JBossFuse:karaf@root> profile-edit --resource broker.xml mq-base 1.2
```

The built-in profile editor opens automatically, which you can use to edit the contents of the **broker.xml** resource.

3. Add the LDAP authorization plug-in to the broker configuration, **broker.xml**. Using the editor that opened in the previous step, add the default **authorizationPlugin** element as a child of the **plugins** element, as follows:

```

<beans ...>
  <broker ...>
    ...
    <plugins>
      ...
      <authorizationPlugin>
        <map>
          <cachedLDAPAuthorizationMap
            connectionURL="ldap://Hostname:Port"
            connectionUsername="cn=Directory Manager"
            connectionPassword="LDAPPASSWORD"
            queueSearchBase="ou=Queue,ou=Destination,ou=ActiveMQ,dc=YourDomain"
            topicSearchBase="ou=Topic,ou=Destination,ou=ActiveMQ,dc=YourDomain"
            tempSearchBase="ou=Temp,ou=Destination,ou=ActiveMQ,dc=YourDomain"
            groupObjectClass="groupOfUniqueNames"
            permissionGroupMemberAttribute="uniquemember"
            refreshInterval="300000"
            legacyGroupMapping="false"
            groupClass="org.apache.karaf.jaas.boot.principal.RolePrincipal"
          />
        </map>
      </authorizationPlugin>
    </plugins>
    ...
  </broker>
</beans>

```

You must customize the following settings in the **broker.xml** resource:

connectionURL

Set this URL to the actual location of your directory server instance. Normally, this URL has the format, **ldap://*Hostname:Port***. For example, the default port for the 389 Directory Server is IP port **389**.

connectionUsername

Specifies the username that is used to authenticate the connection to the directory server. For 389 Directory Server, the default is usually **cn=Directory Manager**.

connectionPassword

Specifies the password part of the credentials for connecting to the directory server.

queueSearchBase

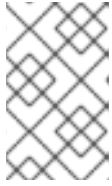
Replace ***YourDomain*** with the name of the root node on your directory server.

topicSearchBase

Replace ***YourDomain*** with the name of the root node on your directory server.

tempSearchBase

Replace ***YourDomain*** with the name of the root node on your directory server.

**NOTE**

For more details about the options available on the **cachedLDAPAuthorizationMap** element, see [Section 8.2, “Cached LDAP Authorization Plug-In”](#).

4. Save and close the **broker.xml** resource by typing Ctrl-S and Ctrl-X.
5. To check that you have edited the **broker.xml** resource correctly, you can print out the 1.2 version of the **mq-base** profile and its resources using the following console command:

```
JBossFuse:karaf@root> profile-display --version 1.2 -r mq-base
```

6. Ensure that the X.500 directory server is running. If necessary, manually restart the X.500 directory server—see [Section 9.2, “Set-up a Directory Server and Console”](#). If the server is not running, all broker connections will fail.
7. The broker LDAP authorization is not activated, until you upgrade a container to use the new version, **1.2**, of the **mq-base** profile. For example, to activate broker LDAP authorization on the **root** container, enter the following console command (assuming a broker profile is already deployed on the **root** container):

```
JBossFuse:karaf@root> container-upgrade 1.2 root
```

Install the Apache ActiveMQ kit

For testing purposes, it is useful to install the Apache ActiveMQ example producer and consumer clients. These example clients are *not* provided directly in the AMQ package. But you can obtain the sample clients by installing the Apache ActiveMQ kit, **apache-activemq-5.11.0.redhat-630187-bin.zip**, provided in the **extras/** directory of the AMQ installation.

Install the Apache ActiveMQ kit as follows:

1. Find the Apache ActiveMQ kit at the following location:

```
InstallDir/extras/apache-activemq-5.11.0.redhat-630187-bin.zip
```

2. Using a suitable archive utility on your platform, unzip the **apache-activemq-5.11.0.redhat-630187-bin.zip** file and extract it to a convenient location, **ActiveMQInstallDir**.

Test the new configuration

To test the new configuration, run the example consumer and producer clients as follows:

1. Run the consumer client with the **jdoue** user credentials. Open a new command prompt, change directory to **ActiveMQInstallDir/examples/openwire/swissarmy**, and enter the following Ant command:

```
ant consumer -Durl=tcp://localhost:61616 -Dmax=100 -Duser=jdoue -Dpassword=secret
```

**NOTE**

If testing against a Fabric container, you might need to change the broker port to **61617**.

2. Run the producer client with the **jdoe** user credentials. Open a new command prompt, change directory to **ActiveMQInstallDir/examples/openwire/swissarmy**, and enter the following Ant command:

```
ant producer -Durl=tcp://localhost:61616 -Dmax=100 -Duser=jdoe -Dpassword=secret
```

3. Run a negative test, to demonstrate that unauthorized users are blocked from accessing the broker queues.

Run the consumer client with the **janedoe** user credentials. Open a new command prompt, change directory to **ActiveMQInstallDir/examples/openwire/swissarmy**, and enter the following Ant command:

```
ant consumer -Durl=tcp://localhost:61616 -Dmax=100 -Duser=janedoe -Dpassword=secret
```

This time, the consumer client fails, because **janedoe** does not belong to the **Administrator** group.

CHAPTER 10. SECURING THE APACHE ACTIVEMQ STANDARD DISTRIBUTION

Abstract

There are significant differences in how you go about securing the Apache ActiveMQ standard distribution, as compared with the main AMQ product. This chapter explains some of those differences.

10.1. APACHE ACTIVEMQ STANDARD DISTRIBUTION

Overview

AMQ provides a standard distribution of Apache ActiveMQ, which is essentially a Red Hat version of the broker developed by the Apache ActiveMQ community at activemq.apache.org. Note the following points about this distribution:

- The broker does *not* run in a container (neither OSGi nor Java EE). It runs directly inside a JVM.
- The broker is launched by the **bin/activemq** script (extracted from the standard distribution archive).
- The libraries included with the Red Hat version of Apache ActiveMQ are identical to (and have identical versions as) the libraries deployed in the main AMQ product.
- The Red Hat version (and *only* the Red Hat version) of the Apache ActiveMQ standard distribution is supported and the libraries in this distribution can be patched from time to time.

Location of the standard distribution

The Apache ActiveMQ standard distribution is provided as an archive file, in the following location:

```
InstallDir/extras/apache-activemq-5.11.0.redhat-630187-bin.zip
```

After unpacking the archive to a convenient location on the file system, you will be able to access the commands and scripts referred to in this chapter.

10.2. CONFIGURE AND RUN ACTIVE-MQ USING ENCRYPTED PASSWORDS

Configure Password Encryption

ActiveMQ allows you to encrypt passwords and store them in configuration files. To encrypt the password, perform the following steps:

1. Run encrypt command.

```
$ bin/activemq encrypt --password encrypt123 --input password1
INFO: Using default configuration
...
Encrypted text: jkS8uzTLGwAoBzxEadnG6j7vkY1GD4Zt
```

Where **password** is a secret used by the encryptor and **input** is the password you want to encrypt.



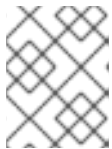
NOTE

Special characters, such as \$/^&, are not supported. Use only alphanumeric characters for passwords.

2. Add the password to the configuration file. By default, the credentials are added to the **\$ACTIVEMQ_HOME/conf/credentials-enc.properties**.

The contents of the **credentials-enc.properties** use the **ENC()** function to wrap encrypted passwords.

3. Instruct the property loader to encrypt variables while loading properties to the memory.



NOTE

The property loader used for encryption is **\$ACTIVEMQ_HOME/examples/conf/activemq-security.xml**.

The contents of the **activemq-security.xml** shows the configuration that ActiveMQ uses to load encrypted passwords. The **ACTIVEMQ_ENCRYPTION_PASSWORD** environment variable is used to load the encryptor password. The property loader then de-encrypts the password from the **credential-enc.properties** file.

```
<bean id="environmentVariablesConfiguration"
class="org.jasypt.encryption.pbe.config.EnvironmentStringPBEConfig">
  <property name="algorithm" value="PBEWithMD5AndDES" />
  <property name="passwordEnvName" value="ACTIVEMQ_ENCRYPTION_PASSWORD" />
</bean>

<bean id="configurationEncryptor"
class="org.jasypt.encryption.pbe.StandardPBEStrngEncryptor">
  <property name="config" ref="environmentVariablesConfiguration" />
</bean>

<bean id="propertyConfigurer"
class="org.jasypt.spring31.properties.EncryptablePropertyPlaceholderConfigurer">
  <constructor-arg ref="configurationEncryptor" />
  <property name="location" value="file:${activemq.base}/conf/credentials-enc.properties"/>
</bean>
```

4. Add the property to the **activemq-security.xml** as shown here:

```
<simpleAuthenticationPlugin>
  <users>
    <authenticationUser username="system"
      password="{activemq.password}"
      groups="users,admins"/>
    <authenticationUser username="user"
      password="{guest.password}"
      groups="users"/>
    <authenticationUser username="guest"
```

```

    password="{guest.password}"
    groups="guests"/>
  </users>
</simpleAuthenticationPlugin>

```

Run Active-MQ using Encrypted Passwords

To run the Active-MQ broker with encrypted password configuration, follow the following steps:

1. Set environment variable for encryption

```
$ export ACTIVEMQ_ENCRYPTION_PASSWORD=encrypt123
```

2. Set the AMQ broker

```
$ bin/activemq start xbean:examples/conf/activemq-security.xml
```

3. Reset the environment variable for encryption

```
$ unset ACTIVEMQ_ENCRYPTION_PASSWORD
```

Resetting the environment is important to avoid saving passwords on your system.

Configuring the network connector

Given two brokers, Broker A and Broker B, where Broker A is configured to perform authentication, you can configure Broker B to log on to Broker A by setting the **userName** attribute and the **password** attribute in the **networkConnector** element, as follows:

```

<beans ...>
  <broker ...>
    ...
    <networkConnectors>
      <networkConnector name="BrokerABridge"
        userName="Username"
        password="Password"
        uri="static://(ssl://brokerA:61616)"/>
      ...
    </networkConnectors>
    ...
  </broker>
</beans>

```

If Broker A is configured to connect to Broker B, Broker A's **networkConnector** element must also be configured with username/password credentials, even if Broker B is not configured to perform authentication. This is because Broker A's authentication plug-in checks for Broker A's username.

APPENDIX A. MANAGING CERTIFICATES

Abstract

TLS authentication uses X.509 certificates—a common, secure and reliable method of authenticating your application objects. You can create X.509 certificates that identify your Red Hat AMQ applications.

A.1. WHAT IS AN X.509 CERTIFICATE?

Role of certificates

An X.509 certificate binds a name to a public key value. The role of the certificate is to associate a public key with the identity contained in the X.509 certificate.

Integrity of the public key

Authentication of a secure application depends on the integrity of the public key value in the application's certificate. If an impostor replaces the public key with its own public key, it can impersonate the true application and gain access to secure data.

To prevent this type of attack, all certificates must be signed by a *certification authority* (CA). A CA is a trusted node that confirms the integrity of the public key value in a certificate.

Digital signatures

A CA signs a certificate by adding its *digital signature* to the certificate. A digital signature is a message encoded with the CA's private key. The CA's public key is made available to applications by distributing a certificate for the CA. Applications verify that certificates are validly signed by decoding the CA's digital signature with the CA's public key.



WARNING

The supplied demonstration certificates are self-signed certificates. These certificates are insecure because anyone can access their private key. To secure your system, you must create new certificates signed by a trusted CA.

Contents of an X.509 certificate

An X.509 certificate contains information about the certificate subject and the certificate issuer (the CA that issued the certificate). A certificate is encoded in Abstract Syntax Notation One (ASN.1), a standard syntax for describing messages that can be sent or received on a network.

The role of a certificate is to associate an identity with a public key value. In more detail, a certificate includes:

- A *subject distinguished name (DN)* that identifies the certificate owner.
- The *public key* associated with the subject.

- X.509 version information.
- A *serial number* that uniquely identifies the certificate.
- An *issuer DN* that identifies the CA that issued the certificate.
- The digital signature of the issuer.
- Information about the algorithm used to sign the certificate.
- Some optional X.509 v.3 extensions; for example, an extension exists that distinguishes between CA certificates and end-entity certificates.

Distinguished names

A DN is a general purpose X.500 identifier that is often used in the context of security.

See [Appendix B, ASN.1 and Distinguished Names](#) for more details about DNs.

A.2. CERTIFICATION AUTHORITIES

A.2.1. Introduction to Certificate Authorities

A CA consists of a set of tools for generating and managing certificates and a database that contains all of the generated certificates. When setting up a system, it is important to choose a suitable CA that is sufficiently secure for your requirements.

There are two types of CA you can use:

- [commercial CAs](#) are companies that sign certificates for many systems.
- [private CAs](#) are trusted nodes that you set up and use to sign certificates for your system only.

A.2.2. Commercial Certification Authorities

Signing certificates

There are several commercial CAs available. The mechanism for signing a certificate using a commercial CA depends on which CA you choose.

Advantages of commercial CAs

An advantage of commercial CAs is that they are often trusted by a large number of people. If your applications are designed to be available to systems external to your organization, use a commercial CA to sign your certificates. If your applications are for use within an internal network, a private CA might be appropriate.

Criteria for choosing a CA

Before choosing a commercial CA, consider the following criteria:

- What are the certificate-signing policies of the commercial CAs?
- Are your applications designed to be available on an internal network only?

- What are the potential costs of setting up a private CA compared to the costs of subscribing to a commercial CA?

A.2.3. Private Certification Authorities

Choosing a CA software package

If you want to take responsibility for signing certificates for your system, set up a private CA. To set up a private CA, you require access to a software package that provides utilities for creating and signing certificates. Several packages of this type are available.

OpenSSL software package

One software package that allows you to set up a private CA is OpenSSL, <http://www.openssl.org>. The OpenSSL package includes basic command line utilities for generating and signing certificates. Complete documentation for the OpenSSL command line utilities is available at <http://www.openssl.org/docs>.

Setting up a private CA using OpenSSL

To set up a private CA, see the instructions in [Section A.5, "Creating Your Own Certificates"](#).

Choosing a host for a private certification authority

Choosing a host is an important step in setting up a private CA. The level of security associated with the CA host determines the level of trust associated with certificates signed by the CA.

If you are setting up a CA for use in the development and testing of Red Hat AMQ applications, use any host that the application developers can access. However, when you create the CA certificate and private key, do not make the CA private key available on any hosts where security-critical applications run.

Security precautions

If you are setting up a CA to sign certificates for applications that you are going to deploy, make the CA host as secure as possible. For example, take the following precautions to secure your CA:

- Do not connect the CA to a network.
- Restrict all access to the CA to a limited set of trusted users.
- Use an RF-shield to protect the CA from radio-frequency surveillance.

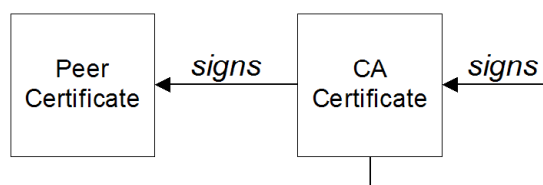
A.3. CERTIFICATE CHAINING

Certificate chain

A *certificate chain* is a sequence of certificates, where each certificate in the chain is signed by the subsequent certificate.

[Figure A.1, "A Certificate Chain of Depth 2"](#) shows an example of a simple certificate chain.

Figure A.1. A Certificate Chain of Depth 2



Self-signed certificate

The last certificate in the chain is normally a *self-signed certificate*—a certificate that signs itself.

Chain of trust

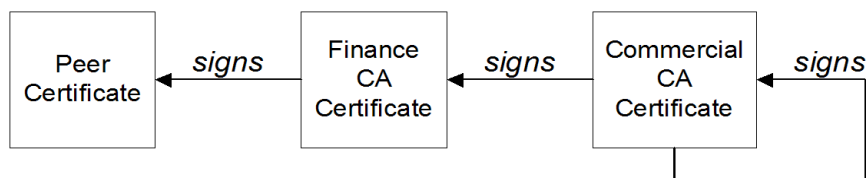
The purpose of a certificate chain is to establish a chain of trust from a peer certificate to a trusted CA certificate. The CA vouches for the identity in the peer certificate by signing it. If the CA is one that you trust (indicated by the presence of a copy of the CA certificate in your root certificate directory), this implies you can trust the signed peer certificate as well.

Certificates signed by multiple CAs

A CA certificate can be signed by another CA. For example, an application certificate could be signed by the CA for the finance department of Progress Software, which in turn is signed by a self-signed commercial CA.

Figure A.2, “A Certificate Chain of Depth 3” shows what this certificate chain looks like.

Figure A.2. A Certificate Chain of Depth 3



Trusted CAs

An application can accept a peer certificate, provided it trusts at least one of the CA certificates in the signing chain.

A.4. SPECIAL REQUIREMENTS ON HTTPS CERTIFICATES

Overview

The HTTPS specification mandates that HTTPS clients must be capable of verifying the identity of the server. This can potentially affect how you generate your X.509 certificates. The mechanism for verifying the server identity depends on the type of client. Some clients might verify the server identity by accepting only those server certificates signed by a particular trusted CA. In addition, clients can inspect the contents of a server certificate and accept only the certificates that satisfy specific constraints.

In the absence of an application-specific mechanism, the HTTPS specification defines a generic mechanism, known as the *HTTPS URL integrity check*, for verifying the server identity. This is the standard mechanism used by Web browsers.

HTTPS URL integrity check

The basic idea of the URL integrity check is that the server certificate's identity must match the server host name. This integrity check has an important impact on how you generate X.509 certificates for HTTPS: *the certificate identity (usually the certificate subject DN's common name) must match the host name on which the HTTPS server is deployed.*

The URL integrity check is designed to prevent *man-in-the-middle* attacks.

Reference

The HTTPS URL integrity check is specified by RFC 2818, published by the Internet Engineering Task Force (IETF) at <http://www.ietf.org/rfc/rfc2818.txt>.

How to specify the certificate identity

The certificate identity used in the URL integrity check can be specified in one of the following ways:

- [Using commonName](#)
- [Using subjectAltName](#)

Using commonName

The usual way to specify the certificate identity (for the purpose of the URL integrity check) is through the Common Name (CN) in the subject DN of the certificate.

For example, if a server supports secure TLS connections at the following URL:

```
https://www.redhat.com/secure
```

The corresponding server certificate would have the following subject DN:

```
C=IE,ST=Co. Dublin,L=Dublin,O=RedHat,  
OU=System,CN=www.redhat.com
```

Where the CN has been set to the host name, **www.redhat.com**.

For details of how to set the subject DN in a new certificate, see [Section A.5, "Creating Your Own Certificates"](#).

Using subjectAltName (multi-homed hosts)

Using the subject DN's Common Name for the certificate identity has the disadvantage that only *one* host name can be specified at a time. If you deploy a certificate on a multi-homed host, however, you might find it is practical to allow the certificate to be used with *any* of the multi-homed host names. In this case, it is necessary to define a certificate with multiple, alternative identities, and this is only possible using the **subjectAltName** certificate extension.

For example, if you have a multi-homed host that supports connections to either of the following host names:

```
www.redhat.com  
www.jboss.org
```


Then you can define a **subjectAltName** that explicitly lists both of these DNS host names. If you generate your certificates using the **openssl** utility, edit the relevant line of your **openssl.cnf** configuration file to specify the value of the **subjectAltName** extension, as follows:

```
subjectAltName=DNS:www.redhat.com,DNS:www.jboss.org
```

Where the HTTPS protocol matches the server host name against either of the DNS host names listed in the **subjectAltName** (the **subjectAltName** takes precedence over the Common Name).

The HTTPS protocol also supports the wildcard character, *, in host names. For example, you can define the **subjectAltName** as follows:

```
subjectAltName=DNS:*.jboss.org
```

This certificate identity matches any three-component host name in the domain jboss.org.



WARNING

You must *never* use the wildcard character in the domain name (and you must take care never to do this accidentally by forgetting to type the dot, ., delimiter in front of the domain name). For example, if you specified ***jboss.org**, your certificate could be used on *any* domain that ends in the letters **jboss**.

A.5. CREATING YOUR OWN CERTIFICATES

Abstract

This chapter describes the techniques and procedures to set up your own private Certificate Authority (CA) and to use this CA to generate and sign your own certificates.



WARNING

Creating and managing your own certificates requires an expert knowledge of security. While the procedures described in this chapter can be convenient for generating your own certificates for demonstration and testing environments, it is *not recommended* to use these certificates in a production environment.

A.5.1. Install the OpenSSL Utilities

Installing OpenSSL on RHEL and Fedora platforms

On Red Hat Enterprise Linux (RHEL) 5 and 6 and Fedora platforms, are made available as an RPM package. To install OpenSSL, enter the following command (executed with administrator privileges):

```
yum install openssl
```

Source code distribution

The source distribution of OpenSSL is available from <http://www.openssl.org/docs>. The OpenSSL project provides source code distributions *only*. You cannot download a binary install of the OpenSSL utilities from the OpenSSL Web site.

A.5.2. Set Up a Private Certificate Authority

Overview

If you choose to use a private CA you need to generate your own certificates for your applications to use. The OpenSSL project provides free command-line utilities for setting up a private CA, creating signed certificates, and adding the CA to your Java keystore.



WARNING

Setting up a private CA for a production environment requires a high level of expertise and extra care must be taken to protect the certificate store from external threats.

Steps to set up a private Certificate Authority

To set up your own private Certificate Authority:

1. Create the directory structure for the CA, as follows:

```
X509CA/demoCA
X509CA/demoCA/private
X509CA/demoCA/certs
X509CA/demoCA/newcerts
X509CA/demoCA/crl
```

2. Using a text editor, create the file, **X509CA/openssl.cfg**, and add the following contents to this file:

Example A.1. OpenSSL Configuration

```
#
# SSLey example configuration file.
# This is mostly being used for generation of certificate requests.
#
RANDFILE      = ./rnd
```

```
#####
[ req ]
default_bits      = 2048
default_keyfile   = keySS.pem
distinguished_name = req_distinguished_name
encrypt_rsa_key   = yes
default_md        = sha1

[ req_distinguished_name ]
countryName       = Country Name (2 letter code)

organizationName = Organization Name (eg, company)

commonName        = Common Name (eg, YOUR name)

#####
[ ca ]
default_ca        = CA_default      # The default ca section

#####
[ CA_default ]

dir               = ./demoCA        # Where everything is kept
certs             = $dir/certs      # Where the issued certs are kept
crl_dir           = $dir/crl        # Where the issued crl are kept
database         = $dir/index.txt   # database index file.
#unique_subject   = no              # Set to 'no' to allow creation of
                                   # several certificates with same subject.
new_certs_dir     = $dir/newcerts   # default place for new certs.

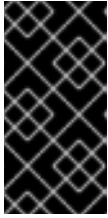
certificate       = $dir/cacert.pem # The CA certificate
serial           = $dir/serial      # The current serial number
crl              = $dir/crl.pem     # The current CRL
private_key       = $dir/private/cakey.pem # The private key
RANDFILE         = $dir/private/.rand # private random number file

name_opt         = ca_default       # Subject Name options
cert_opt         = ca_default       # Certificate field options

default_days     = 365              # how long to certify for
default_crl_days = 30               # how long before next CRL
default_md       = md5              # which md to use.
preserve         = no               # keep passed DN ordering

policy           = policy_anything

[ policy_anything ]
countryName      = optional
stateOrProvinceName = optional
localityName     = optional
organizationName = optional
organizationalUnitName = optional
commonName       = supplied
emailAddress     = optional
```

**IMPORTANT**

The preceding **openssl.cfg** configuration file is provided *as a demonstration only*. In a production environment, this configuration file would need to be carefully elaborated by an engineer with a high level of security expertise, and actively maintained to protect against evolving security threats.

- Initialize the **demoCA/serial** file, which must have the initial contents **01** (zero one). Enter the following command:

```
echo 01 > demoCA/serial
```

- Initialize the **demoCA/index.txt**, which *must* initially be completely empty. Enter the following command:

```
touch demoCA/index.txt
```

- Create a new self-signed CA certificate and private key with the command:

```
openssl req -x509 -new -config openssl.cfg -days 365 -out demoCA/cacert.pem -keyout demoCA/private/cakey.pem
```

You are prompted for a pass phrase for the CA private key and details of the CA distinguished name as shown in [Example A.2, "Creating a CA Certificate"](#).

Example A.2. Creating a CA Certificate

```
Generating a 2048 bit RSA private key
.....+++
.....+++
writing new private key to 'demoCA/private/cakey.pem'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) []:DE
Organization Name (eg, company) []:Red Hat
Common Name (eg, YOUR name) []:Scooby Doo
```

**NOTE**

The security of the CA depends on the security of the private key file and the private key pass phrase used in this step.

You must ensure that the file names and location of the CA certificate and private key, **cacert.pem** and **cakey.pem**, are the same as the values specified in **openssl.cfg**.

A.5.3. Create a CA Trust Store File

Overview

A trust store file is commonly required on the client side of an SSL/TLS connection, in order to verify a server's identity. A trust store file can also be used to check digital signatures (for example, to check that a signature was made using the private key corresponding to one of the trusted certificates in the trust store file).

Steps to create a CA trust store

To add one or more CA certificates to a trust store file:

1. Assemble the collection of trusted CA certificates that you want to deploy.

The trusted CA certificates can be obtained from public CAs or private CAs. The trusted CA certificates can be in any format that is compatible with the Java **keystore** utility; for example, PEM format. All you need are the certificates themselves—the private keys and passwords are *not* required.

2. Add a CA certificate to the trust store using the **keytool -import** command.

Enter the following command to add the CA certificate, **cacert.pem**, in PEM format, to a JKS trust store.

```
keytool -import -file cacert.pem -alias CAAlias -keystore truststore.ts -storepass StorePass
```

Where **truststore.ts** is a keystore file containing CA certificates. If this file does not already exist, the **keytool** command creates it. The **CAAlias** is a convenient identifier for the imported CA certificate and **StorePass** is the password required to access the keystore file.

3. Repeat the previous step to add all of the CA certificates to the trust store.

A.5.4. Generate and Sign a New Certificate

Overview

In order for a certificate to be useful in the real world, it must be signed by a CA, which vouches for the authenticity of the certificate. This facilitates a scalable solution for certificate verification, because it means that a single CA certificate can be used to verify a large collection of certificates.

Steps to generate and sign a new certificate

To generate and sign a new certificate, using your own private CA, perform the following steps:

1. Generate a certificate and private key pair using the **keytool -genkeypair** command, as follows:

```
keytool -genkeypair -keyalg RSA -dname "CN=Alice, OU=Engineering, O=Red Hat, ST=Dublin, C=IE" -validity 365 -alias alice -keypass KeyPass -keystore alice.ks -storepass StorePass
```

Because the specified keystore, **alice.ks**, did not exist prior to issuing the command implicitly creates a new keystore and sets its password to **StorePass**.

The **-dname** and **-validity** flags define the contents of the newly created X.509 certificate.



NOTE

When specifying the certificate's Distinguished Name (through the **-dname** parameter), you must be sure to observe any policy constraints specified in the **openssl.cfg** file. If those policy constraints are not heeded, you will not be able to sign the certificate using the CA (in the next steps).



NOTE

It is essential to generate the key pair with the **-keyalg RSA** option (or a key algorithm of similar strength). The default key algorithm uses a combination of DSA encryption and SHA-1 signature. But the SHA-1 algorithm is no longer regarded as sufficiently secure and modern Web browsers will reject certificates signed using SHA-1. When you select the RSA key algorithm, the **keytool** utility uses an SHA-2 algorithm instead.

2. Create a certificate signing request using the **keytool -certreq** command.

Create a new certificate signing request for the **alice.ks** certificate and export it to the **alice_csr.pem** file, as follows:

```
keytool -certreq -alias alice -file alice_csr.pem -keypass KeyPass -keystore alice.ks -storepass StorePass
```

3. Sign the CSR using the **openssl ca** command.

Sign the CSR for the Alice certificate, using your private CA, as follows:

```
openssl ca -config openssl.cfg -days 365 -in alice_csr.pem -out alice_signed.pem
```

You will be prompted to enter the CA private key pass phrase you used when creating the CA (in [Step 5](#)).

For more details about the **openssl ca** command see <http://www.openssl.org/docs/apps/ca.html#>.

4. Convert the signed certificate to PEM only format using the **openssl x509** command with the **-outform** option set to **PEM**. Enter the following command:

```
openssl x509 -in alice_signed.pem -out alice_signed.pem -outform PEM
```

5. Concatenate the CA certificate file and the converted, signed certificate file to form a certificate chain. For example, on Linux and UNIX platforms, you can concatenate the CA certificate file and the signed Alice certificate, **alice_signed.pem**, as follows:

```
cat demoCA/cacert.pem alice_signed.pem > alice.chain
```

6. Import the new certificate's full certificate chain into the Java keystore using the **keytool -import** command. Enter the following command:

```
keytool -import -file alice.chain -keypass KeyPass -keystore alice.ks -storepass StorePass
```

APPENDIX B. ASN.1 AND DISTINGUISHED NAMES

Abstract

The OSI Abstract Syntax Notation One (ASN.1) and X.500 Distinguished Names play an important role in the security standards that define X.509 certificates and LDAP directories.

B.1. ASN.1

Overview

The *Abstract Syntax Notation One* (ASN.1) was defined by the OSI standards body in the early 1980s to provide a way of defining data types and structures that are independent of any particular machine hardware or programming language. In many ways, ASN.1 can be considered a forerunner of modern interface definition languages, such as the [OMG's IDL](#) and WSDL, which are concerned with defining platform-independent data types.

ASN.1 is important, because it is widely used in the definition of standards (for example, SNMP, X.509, and LDAP). In particular, ASN.1 is ubiquitous in the field of security standards. The formal definitions of X.509 certificates and distinguished names are described using ASN.1 syntax. You're not required to have detailed knowledge of ASN.1 syntax to use these security standards, but you need to be aware that ASN.1 is used for the basic definitions of most security-related data types.

BER

The OSI's Basic Encoding Rules (BER) define how to translate an ASN.1 data type into a sequence of octets (binary representation). The role played by BER with respect to ASN.1 is, therefore, similar to the role played by GIOP with respect to the OMG IDL.

DER

The OSI's Distinguished Encoding Rules (DER) are a specialization of the BER. The DER consists of the BER plus some additional rules to ensure that the encoding is unique (BER encodings are not).

References

You can read more about ASN.1 in the following standards documents:

- ASN.1 is defined in the ITU [X.208](#) specification.
- BER is defined in the ITU [X.209](#) specification.

B.2. DISTINGUISHED NAMES

Overview

Historically, distinguished names (DN) are defined as the primary keys in an X.500 directory structure. However, DNs have come to be used in many other contexts as general purpose identifiers. In Apache CXF, DNs occur in the following contexts:

- X.509 certificates—for example, one of the DNs in a certificate identifies the owner of the certificate (the security principal).
- LDAP—DNs are used to locate objects in an LDAP directory tree.

String representation of DN

Although a DN is formally defined in ASN.1, there is also an LDAP standard that defines a UTF-8 string representation of a DN (see [RFC 2253](#)). The string representation provides a convenient basis for describing the structure of a DN.



NOTE

The string representation of a DN does *not* provide a unique representation of DER-encoded DN. Hence, a DN that is converted from string format back to DER format does not always recover the original DER encoding.

DN string example

The following string is a typical example of a DN:

```
C=US,O=IONA Technologies,OU=Engineering,CN=A. N. Other
```

Structure of a DN string

A DN string is built up from the following basic elements:

- [OID](#) .
- [Attribute Types](#) .
- [AVA](#) .
- [RDN](#) .

OID

An OBJECT IDENTIFIER (OID) is a sequence of bytes that uniquely identifies a grammatical construct in ASN.1.

Attribute types

The variety of attribute types that can appear in a DN is theoretically open-ended, but in practice only a small subset of attribute types are used. [Table B.1, “Commonly Used Attribute Types”](#) shows a selection of the attribute types that you are most likely to encounter:

Table B.1. Commonly Used Attribute Types

String Representation	X.500 Attribute Type	Size of Data	Equivalent OID
C	countryName	2	2.5.4.6
O	organizationName	1..64	2.5.4.10

String Representation	X.500 Attribute Type	Size of Data	Equivalent OID
OU	organizationalUnitName	1..64	2.5.4.11
CN	commonName	1..64	2.5.4.3
ST	stateOrProvinceName	1..64	2.5.4.8
L	localityName	1..64	2.5.4.7
STREET	streetAddress		
DC	domainComponent		
UID	userid		

AVA

An *attribute value assertion* (AVA) assigns an attribute value to an attribute type. In the string representation, it has the following syntax:

```
<attr-type>=<attr-value>
```

For example:

```
CN=A. N. Other
```

Alternatively, you can use the equivalent OID to identify the attribute type in the string representation (see [Table B.1, "Commonly Used Attribute Types"](#)). For example:

```
2.5.4.3=A. N. Other
```

RDN

A *relative distinguished name* (RDN) represents a single node of a DN (the bit that appears between the commas in the string representation). Technically, an RDN might contain more than one AVA (it is formally defined as a set of AVAs). However, this almost never occurs in practice. In the string representation, an RDN has the following syntax:

```
<attr-type>=<attr-value>[+<attr-type>=<attr-value> ...]
```

Here is an example of a (very unlikely) multiple-value RDN:

```
OU=Eng1+OU=Eng2+OU=Eng3
```

Here is an example of a single-value RDN:

```
OU=Engineering
```

INDEX

A

Abstract Syntax Notation One (see ASN.1)

ActiveMQSslConnectionFactory, [ActiveMQSslConnectionFactory class](#)

ASN.1, [Contents of an X.509 certificate](#), [ASN.1 and Distinguished Names attribute types](#), [Attribute types](#)

AVA, [AVA](#)

OID, [OID](#)

RDN, [RDN](#)

attribute value assertion (see AVA)

authorization

temporary destinations, [Temporary destinations](#)

authorizationEntries, [Configuring the simple authorization plug-in](#)

authorizationEntry, [Named destinations](#)

authorizationMap, [Configuring the simple authorization plug-in](#)

authorizationPlugin, [Configuring the simple authorization plug-in](#)

AVA, [AVA](#)

B

Basic Encoding Rules (see BER)

BER, [BER](#)

C

CA, [Integrity of the public key](#)

choosing a host, [Choosing a host for a private certification authority](#)

commercial CAs, [Commercial Certification Authorities](#)

list of trusted, [Trusted CAs](#)

multiple CAs, [Certificates signed by multiple CAs](#)

private CAs, [Private Certification Authorities](#)

security precautions, [Security precautions](#)

certificates

chaining, [Certificate chain](#)

peer, [Chain of trust](#)

public key, [Contents of an X.509 certificate](#)

self-signed, [Self-signed certificate](#)

signing, [Integrity of the public key](#)

X.509, [Role of certificates](#)

chaining of certificates, [Certificate chain](#)

D

DER, [DER](#)

Distinguished Encoding Rules (see DER)

distinguished names

definition, [Overview](#)

DN

definition, [Overview](#)

string representation, [String representation of DN](#)

J

JAAS

configuration syntax, [Configuring a JAAS realm](#)

converting to blueprint, [Converting standard JAAS login properties to XML](#)

namespace, [Namespace](#)

jaas:config, [Configuring a JAAS realm](#)

jaas:module, [Configuring a JAAS realm](#)

JMX SSL connection, enabling, [Enabling Remote JMX SSL](#)

M

multiple CAs, [Certificates signed by multiple CAs](#)

O

OpenSSL, [OpenSSL software package](#)

P

peer certificate, [Chain of trust](#)

public keys, [Contents of an X.509 certificate](#)

R

RDN, [RDN](#)

relative distinguished name (see [RDN](#))

root certificate directory, [Trusted CAs](#)

S

self-signed certificate, [Self-signed certificate](#)

signing certificates, [Integrity of the public key](#)

SSLey, [OpenSSL software package](#)

T

tempDestinationAuthorizationEntry, [Configuring the simple authorization plug-in](#), [Temporary destinations](#)

temporary destinations

authorization, [Temporary destinations](#)

trusted CAs, [Trusted CAs](#)

X

X.500, [ASN.1 and Distinguished Names](#)

X.509 certificate

definition, [Role of certificates](#)