Red Hat JBoss A-MQ 6.1

Security Guide

Making Red Hat JBoss A-MQ secure

Last Updated: 2017-10-13

Red Hat JBoss A-MQ 6.1 Security Guide

Making Red Hat JBoss A-MQ secure

JBoss A-MQ Docs Team
Content Services
fuse-docs-support@redhat.com

Legal Notice

Abstract

This guide describes how to configure the Red Hat JBoss A-MQ subsystem.

Table of Contents

CHAPTER 1. SECURITY ARCHITECTURE

Abstract

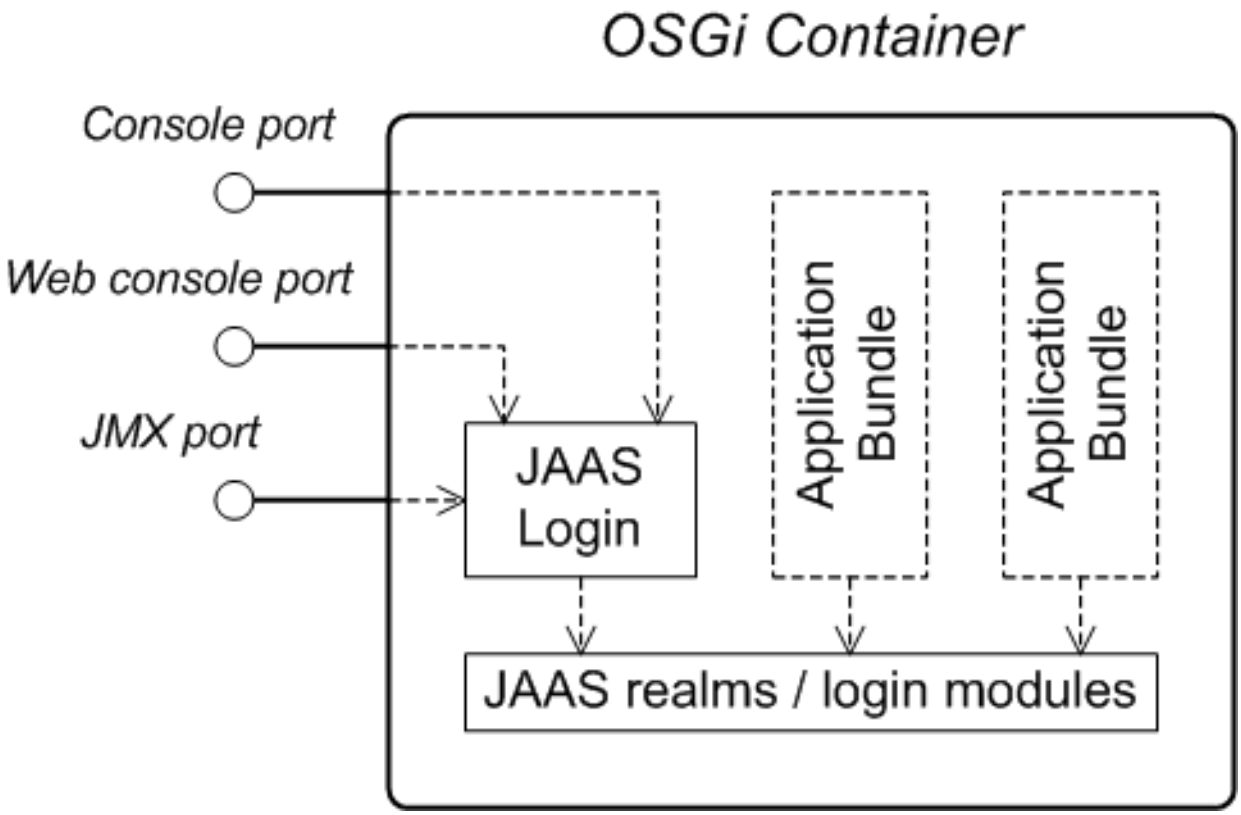
In the OSGi container, it is possible to deploy applications supporting a variety of security features. Currently, only the Java Authentication and Authorization Service (JAAS) is based on a common, container-wide infrastructure. Other security features are provided separately by the individual products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

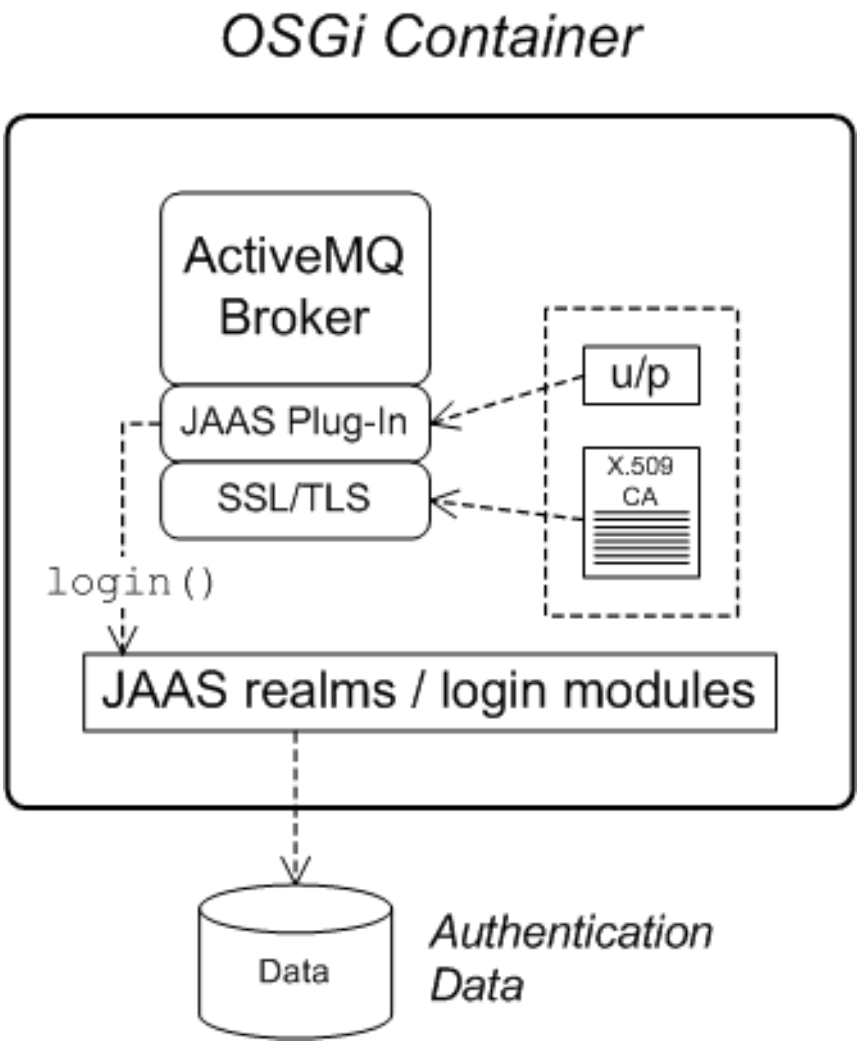
It is essential, however, that the JAAS realms are defined using the OSGi container's own login configuration mechanism—by default, Java provides a simple file-based login configuration implementation, but you *cannot* use this implementation in the context of the OSGi container.

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 6, *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 RED HAT JBOSS A-MQ CONTAINER

Abstract

The Red Hat JBoss A-MQ 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 in a for the default JAAS realm in a standalone container.

Default JAAS realm

The Red Hat JBoss A-MQ 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 JBoss A-MQ 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 two 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
```

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.

Configuring 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[,Role1][,Role2]...
```

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

```
jdoe=topsecret,admin
```

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

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,Role1,Role2,...
```

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

```
jdoe=AAAAB3NzaC1kc3MAAACBAP1/U4EddRIpUt9KnC7s5Of2EbdSPO9EAMMeP4C2USZpRV1AI
lH7WT2NWPq/xfW6MPbLm1Vs14E7
gB00b/JmYLdrmVClpJ+f6AR7ECLCT7up1/63xhv4O1fnfqimFQ8E+4P208UewwI1VBNaFpEy9n
Xzrith1yrv8iIDGZ3RSAHHAAAAFQCX
YFCPFSMLzLKSuYKi64QL8Fgc9QAAAnEA9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0Hgm
dRWVeOutRZT+ZxBxCBgLRJFnEj6Ewo
FhO3zwkyjMim4TwWeotifI0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhRkImog9/hWu
WfBpKLZl6Ae1UlZAFMO/7PSSoAAACB
AKKSU2PFl/qOLxIwmBZPPIcJshVe7bVUpFvyl3BbJDow8rXfskl8wO63OzP/qLmcJM0+JbcRU/
53Jj7uyk31drV2qxhIOsLDC9dGCWj4
7Y7TyhPdXh/0dthTRBy6bqGtRPxGa7gJov1xm/UuYYXPIUR/3x9MAZvZ5xvE0kYXO+rx,admin
```

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.

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. The 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 JBoss A-MQ 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.

jaas:module

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

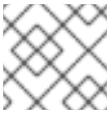
- **className**—the fully-qualified class name of a JAAS login module. The specified class must be available from the bundle classloader.
- **flags**—determines what happens upon success or failure of the login operation. Table 2.1, “Flags for Defining a JAAS Module” describes the valid values.

Table 2.1. Flags for Defining a JAAS Module

Value	Description

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 JBoss A-MQ uses the same properties as a standard Java login configuration file, however JBoss A-MQ requires that they are specified slightly differently. To see how the JBoss A-MQ 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 Apache ActiveMQ 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
className="org.apache.activemq.jaas.PropertiesLoginModule"
                     flags="required">
            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 JBoss A-MQ 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 JBoss A-MQ'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="1">
    <jaas:module
className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule"
                 flags="required">
      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
      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 2.2, “Enabling LDAP Authentication”.

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[,Role][,Role]...
```

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 **2**:

```
<?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="2">
        <jaas:module
className="org.apache.karaf.jaas.modules.properties.PropertiesLoginModule"
                     flags="required">
            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.PropertiesBackingEngineFac
tory"/>
    </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

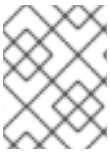
The JAAS OSGi config login module authenticates username/password credentials, returning the list of roles associated with the authenticated user.

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]...
```

This is the same format that is used in a users property file.

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 **2**:

```
<?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="2">
        <jaas:module
className="org.apache.karaf.jaas.modules.osgi.OsgiConfigLoginModule"
                     flags="required">
            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

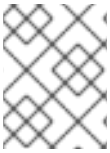
The JAAS public key login module authenticates SSH key credentials. When a user tries to log in, the SSH protocol uses the stored public key to challenge the user. The user must possess the corresponding private key in order to answer the challenge. If login is successful, the login module returns the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

The user 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 user properties file, where a line has the following form:

```
Username=PublicKey[,Role][,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 **`admin`** role, you would create an entry like the following:

```
jdoe=AAAAB3NzaC1kc3MAAACBAP1/U4EddRIpUt9KnC7s5Of2EbdSPO9EAMMeP4C2USZpRV1AI
lH7WT2NWPq/xfW6MPbLm1Vs14E7
gB00b/JmYLdrmVClpJ+f6AR7ECLCT7up1/63xhv4O1fnfqimFQ8E+4P208UewwI1VBNaFpEy9n
Xzrith1yrv8iIDGZ3RSAHHAAAAFQCX
YFCPFSMLzLKSuYKi64QL8Fgc9QAAAnEA9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0Hgm
dRWVeOutRZT+ZxBxCBgLRJFnEj6Ewo
FhO3zwkyjMim4TwWeotifI0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhRkImog9/hWu
WfBpKLZl6Ae1UlZAFMO/7PSSoAAACB
AKKSU2PFl/qOLxIwmBZPPIcJshVe7bVUpFvyl3BbJDow8rXfskl8wO63OzP/qLmcJM0+JbcRU/
53Jj7uyk31drV2qxhIOsLDC9dGCWj4
7Y7TyhPdXh/0dthTRBy6bqGtRPxGa7gJov1xm/UuYYXPIUR/3x9MAZvZ5xvE0kYXO+rx,admin
```

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.

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 **`2`**:

```
<?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="2">
        <jaas:module
className="org.apache.karaf.jaas.modules.publickey.PublickeyLoginModule"
                     flags="required">
            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).

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 class="com.mysql.jdbc.jdbc2.optional.MysqlDataSource"
id="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

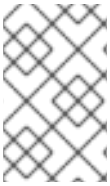
After the data source has been instantiated and exported as an OSGi service, you are ready to configure the JDBC login module. In particular, the **`datasource`** option of the JDBC login module can reference the data source's OSGi service using the following syntax:

```
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="2">
        <jaas:module
className="org.apache.karaf.jaas.modules.jdbc.JDBCLoginModule"
                     flags="required">
            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 JBoss A-MQ, see Chapter 8, *LDAP Authentication Tutorial*.

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.ldap.LDAPLoginModule`

Implements the JAAS login module.

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:

`connection.url`

The LDAP connection URL—for example, **`ldap://hostname`**.

`connection.username`

Admin username to connect to the LDAP server. This parameter is optional: if it is not provided, the LDAP connection will be anonymous.

`connection.password`

Admin password to connect to the LDAP server. Used only if the **connection.username** is also specified.

user.base.dn

The LDAP base DN used to look up roles—for example, **ou=role,dc=apache,dc=org**.

user.filter

The LDAP filter used to look up a user's role—for example, **(member:=uid=%u)**.

user.search.subtree

If **true**, the user lookup is recursive (**SUBTREE**). If **false**, the user lookup is performed only at the first level (**ONELEVEL**).

role.base.dn

The LDAP base DN used to look up roles—for example, **ou=role,dc=apache,dc=org**.

role.filter

The LDAP filter used to look up a user's role—for example, **(member:=uid=%u)**.

role.name.attribute

The LDAP role attribute containing the role value used by Apache Karaf—for example, **cn**.

role.search.subtree

If **true**, the role lookup is recursive (**SUBTREE**). If **false**, the role lookup is performed only at the first level (**ONELEVEL**).

authentication

Define the authentication back-end used on the LDAP server. The default is **simple**.

initial.context.factory

Define the initial context factory used to connect to the LDAP server. The default is **com.sun.jndi.ldap.LdapCtxFactory**.

ssl

If **true** or if the protocol on the **connection.url** is **ldaps**, an SSL connection will be used.

ssl.provider

Specifies the SSL provider.

ssl.protocol

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

ssl.algorithm

The algorithm to use for the **KeyManagerFactory** and the **TrustManagerFactory**—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 Blueprint configuration”).

`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.

Sample Blueprint configuration

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 **`2`**:

```
<?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, for example the karaf.home
property -->
    <ext:property-placeholder placeholder-prefix="${" placeholder-
suffix="}"/>

    <jaas:config name="karaf" rank="2">
        <jaas:module
className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule"
                     flags="required">
            connection.url = ldaps://localhost:10636
            user.base.dn = ou=users,ou=system
            user.filter = (uid=%u)
            user.search.subtree = true
            role.base.dn = ou=groups,ou=system
            role.filter = (uniqueMember=uid=%u)
            role.name.attribute = cn
            role.search.subtree = true
            authentication = simple
            ssl.protocol=TLSv1
            ssl.truststore=ks
            ssl.algorithm=PKIX
        </jaas:module>
    </jaas:config>

    <jaas:keystore name="ks"
                path="file:///${karaf.home}/etc/trusted.ks"
```

```
                keystorePassword="secret" />

</blueprint>
```

IMPORTANT

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

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

IMPORTANT

Although message digest algorithms are not easy 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

Password encryption for JAAS login modules can optionally be enabled by setting the following login module properties:

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

An encryption service can be defined by inheriting from the **org.apache.karaf.jaas.modules.EncryptionService** interface and exporting an instance of the encryption service as an OSGi service. Two alternative implementations of the encryption service are provided:

- the section called “Basic encryption service”.
- the section called “Jasypt encryption”.

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

The Jasypt encryption service can be installed in the standalone container by installing the **`jasypt-encryption`** feature. For example, you can install Jasypt encryption by entering the following console command:

```
JBossFuse: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="2">
        <jaas:module
className="org.apache.karaf.jaas.modules.properties.PropertiesLoginModule"
                     flags="required">
            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
        </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.PropertiesBackingEngineFac
tory"/>
    </service>

</blueprint>
```

2.2. ENABLING LDAP AUTHENTICATION

Overview

Red Hat JBoss A-MQ supplies a JAAS login module that enables it to use LDAP to authenticate users. The JBoss A-MQ JAAS LDAP login module is implemented by the **`org.apache.karaf.jaas.modules.ldap.LDAPLoginModule`** class. It is preloaded in the container, so you do not need to install its bundle.

Procedure

To enable JBoss A-MQ to use LDAP for user authentication you need to create a JAAS realm that includes the JBoss A-MQ LDAP login module. As shown in Example 2.6, “Red Hat JBoss A-MQ LDAP JAAS Login Module”, this is done by adding a **`jaas:module`** element to the realm and setting its **`className`** attribute to **`org.apache.karaf.jaas.modules.ldap.LDAPLoginModule`**.

Example 2.6. Red Hat JBoss A-MQ LDAP JAAS Login Module

```
<jaas:config ... >
  <jaas:module
className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule"
               flags="required">
    ...
  </jaas:module>
</jaas:config>
```

You will also need to provide values for the properties described in Table 2.2, “Properties for the Red Hat JBoss A-MQ LDAP Login Module”.

LDAP properties

Table 2.2, “Properties for the Red Hat JBoss A-MQ LDAP Login Module” describes the properties used to configure the JBoss A-MQ JAAS LDAP login module.

Table 2.2. Properties for the Red Hat JBoss A-MQ LDAP Login Module

Property	Description

Property	Description
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.
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.
user.base.dn	Specifies the DN of the subtree of the DIT to search for user entries.
user.filter	Specifies the LDAP search filter used to locate user credentials. It is applied to the subtree selected by user.base.dn. Before being passed to the LDAP search operation, the value is subjected to string substitution such that all occurrences of **%u** are replaced by the user name extracted from the incoming credentials.
user.search.subtree	Specifies if the user entry search's scope includes the subtrees of the tree selected by user.base.dn.
role.base.dn	Specifies the DN of the subtree of the DIT to search for role entries.
role.filter	Specifies the LDAP search filter used to locate roles. It is applied to the subtree selected by role.base.dn. Before being passed to the LDAP search operation, the value is subjected to string substitution such that all occurrences of **%u** are replaced by the user name extracted from the incoming credentials.
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.
role.search.subtree	Specifies if the role entry search's scope includes the subtrees of the tree selected by role.base.dn.

Property	Description
authentication	Specifies the authentication method used when binding to the LDAP server. Valid values are • **`simple`**—bind with user name and password authentication • **`none`**—bind anonymously
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`**.
ssl	Specifies if the connection to the LDAP server is secured via 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.
ssl.keystore	Specifies the keystore name.
ssl.keyalias	Specifies the name of the private key in the keystore.
ssl.truststore	Specifies the trust keystore name.

All of the properties are mandatory except the SSL properties.

Example

Example 2.7, “Configuring a JAAS Realm that Uses LDAP Authentication” defines a JAAS realm that uses the LDAP server located at ldap://localhost:10389.

Example 2.7. Configuring a JAAS Realm that Uses LDAP Authentication

```
<?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="1">
    <jaas:module
classNameName="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule"
                 flags="sufficient">
      initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
      connection.username=uid=admin,ou=system
      connection.password=secret
      connection.protocol=
      connection.url = ldaps://localhost:10636
      user.base.dn = ou=users,ou=system
      user.filter = (uid=%u)
      user.search.subtree = true
      role.base.dn = ou=roles,ou=system,dc=jbossfuse
      role.filter = (uid=%u)
      role.name.attribute = cn
      role.search.subtree = true
      authentication = simple
      ssl.protocol=TLSv1
      ssl.truststore=truststore
      ssl.algorithm=PKIX
    </jaas:module>
    ...
  </jaas:config>
</blueprint>
```

IMPORTANT

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

2.3. CONFIGURING ROLES FOR THE ADMINISTRATIVE PROTOCOLS

Overview

By configuring each of the administrative functions to use a different role for authorization, you can provide fine grained control over who can monitor and manipulate running containers.

Administration protocols

You can independently configure roles for the following different administrative protocols:

- SSH (remote console login)
- JMX management

Default role

The default role name for all of the administration protocols is set by the **`karaf.admin.role`** property in the Red Hat JBoss A-MQ's **`etc/system.properties`** file. For example, the default setting of **`karaf.admin.role`** is:

```
karaf.admin.role=admin
```

You have the option of overriding the default **admin** role set by **karaf.admin.role** for each of the administrative protocols.

Changing the remote console's role

To override the default role for the remote console add a **sshRole** property to the **org.apache.karaf.shell** PID. The following sets the role to **admin**:

```
sshRole=admin
```

Changing the JMX role

To override the default role for JMX add a **jmxRole** property to the **org.apache.karaf.management** PID. The following sets the role to **jmx**:

```
jmxRole=jmx
```

2.4. 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 in Blueprint XML files.

How to use encrypted property placeholders

To use encrypted property placeholders in a Blueprint 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**. JBoss A-MQ 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 2.8. Property File with an Encrypted Property

```
#ldap.properties
ldap.password=ENC(EncryptedPassword)
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. For example, in the preceding property file example you would replace the ***EncryptedPassword*** value with the output of the **encrypt.sh** Jasypt utility.

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

Example 2.9. 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 2.10, “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 2.10. 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.StandardPBEStringEncryptor">
      <property name="config">
        <bean
class="org.jasypt.encryption.pbe.config.EnvironmentStringPBECon
fig">
          <property name="algorithm" value="PBEWithMD5AndDES"
/>
          <property name="password"
value="JASYPT_ENCRYPTION_PASSWORD" />
        </bean>
      </property>
    </enc:encryptor>
  </enc:property-placeholder>
...
</blueprint>
```

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. Before starting up the JBoss A-MQ container, make sure that you set the **JASYPT_ENCRYPTION_PASSWORD** environment variable to the value of the master password. For example, on a Linux or UNIX system with the bash shell, you would set the environment variable as follows:

   ```
   export JASYPT_ENCRYPTION_PASSWORD=MasterPass
   ```

9. 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
   ```

Blueprint XML example

Example 2.11, “Jasypt Example in Blueprint XML” shows an example of an LDAP JAAS realm configured in Blueprint XML, using Jasypt encrypted property placeholders.

Example 2.11. 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.StandardPBEStringEncryptor">
      <property name="config">
        <bean
class="org.jasypt.encryption.pbe.config.EnvironmentStringPBEConfig">
          <property name="algorithm" value="PBEWithMD5AndDES" />
          <property name="password" value="JASYPT_ENCRYPTION_PASSWORD"
/>
        </bean>
      </property>
    </enc:encryptor>
  </enc:property-placeholder>

  <jaas:config name="karaf" rank="1">
    <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.

2.5. 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 **admin** role

  Edit the **<*installDir*>/jboss-fuse-6.1.0.redhat-379/etc/users.properties** file and add the following entry, on a single line:

  ```
  admin=YourPassword,admin
  ```

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

Create the `jbossweb.keystore` file

Open a command prompt and make sure you are in the **etc/** directory of your JBoss A-MQ installation:

```
cd <installDir>/jboss-fuse-6.1.0.redhat-379/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:
112308602579056704360496299050191816946109837286427320179534244008039380
8

159410077607500864745991099141380637280072294767016640781490175445910072027
9046

394462181373817732403106426038265948319382617744876203043766931839107261986
7218

036972335210839062722456085328301058362052369248473659880488338711351959835
357
  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.l.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-6.1.0.redhat-379/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-6.1.0.redhat-379/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-6.1.0.redhat-379/deploy** directory (the hot deploy directory).

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

Edit the **<*installDir*>/jboss-fuse-6.1.0.redhat-379/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 JBoss A-MQ container

You must restart the JBoss A-MQ 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 JBoss A-MQ installation:

   ```
   cd <installDir>/jboss-fuse-6.1.0.redhat-379/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 JBoss A-MQ 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 (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

To enable SSL/TLS:

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 as shown in Example 3.1, “Pax Web Property for Disabling the HTTP Port”.

   Example 3.1. Pax Web Property for Disabling the HTTP Port

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

3. Enable the secure HTTPS port by adding the org.osgi.service.http.secure.enabled and setting it to **true** as shown in Example 3.2, “Pax Web Property for Enabling the HTTPS Port”.

   Example 3.2. Pax Web Property for Enabling the HTTPS Port

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

4. If you followed the preceding instructions, 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.xml

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

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

5. Edit the **etc/jetty.xml** file and add the following **Call** element to configure the SSL connector for Jetty:

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//
DTD Configure//EN" "http://jetty.mortbay.org/configure.dtd">

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

    <!-- =============================================================
-->
    <!-- Set connectors
-->
    <!-- =============================================================
-->
    <!-- One of each type!
-->
    <!-- =============================================================
-->
    ...
    <Call name="addConnector">
        <Arg>
            <!-- The SslSelectChannelConnector class uses the Java
NIO SslEngine -->
            <New
class="org.eclipse.jetty.server.ssl.SslSelectChannelConnector">
                <Arg>
                    <New
class="org.eclipse.jetty.http.ssl.SslContextFactory">
                        <!-- Protect against the POODLE security
vulnerability -->
                        <Set name="ExcludeProtocols">
                            <Array type="java.lang.String">
                                <Item>SSLv3</Item>
                            </Array>
                        </Set>
                        <Set
name="keyStore">/home/jdoe/Documents/server.keystore</Set>
                        <Set
name="keyStorePassword">mykeystorepass</Set>
                        <Set
name="keyManagerPassword">mykeypass</Set>
                    </New>
                </Arg>
```

```
                <Set name="port">8183</Set>
                <Set name="maxIdleTime">30000</Set>
            </New>
        </Arg>
    </Call>
    <Call name="addConnector">
        ...
    </Call>

    <Call name="addBean">
        ...
    </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.

6. Customize the properties of the **`SslSocketConnector`** instance defined in the **`etc/jetty.xml`** file, as follows:

   `port`

   The secure HTTPS port number.

   `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).

   `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).

7. Restart the JBoss A-MQ container, in order for the configuration changes to take effect.

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://localhost:8183

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

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

- Jetty/Howto/Configure SSL
- Jetty/Reference/SSL Connectors
- SslContextFactory
- API documentation (all Jetty versions)

CHAPTER 4. 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 JBoss A-MQ OSGi container.

4.1. PROGRAMMING CLIENT CREDENTIALS

Overview

Currently, for Java clients of Red Hat JBoss A-MQ, 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);
  ...
  }
```

4.2. 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&amp;waitForStart=10000" />
 <property name="userName" value="Username"/> <property name="password"
value="Password"/>
      </bean>
    </property>
  </bean>
  ...
</beans>
```

4.3. 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.

4.4. TUTORIAL I: JAAS AUTHENTICATION

Overview

This tutorial shows you how to communicate with the JBoss A-MQ 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 JBoss A-MQ. 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 JBoss A-MQ, 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 JBoss A-MQ 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 **admin** role. For example:

```
Username=Password,admin
```

Start the JBoss A-MQ 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)
```

4.5. 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 keystore files”.
3. the section called “Configure the broker”.
4. the section called “Start the JBoss A-MQ container”.
5. the section called “Configure the consumer and the producer clients”.
6. the section called “Run the consumer with the SSL protocol”.
7. 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 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

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">
        ...
        <sslContext>
            <sslContext
                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&amp;maximumConnections=10
00"/>
        </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.

Start the JBoss A-MQ container

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

```
./amq
```

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. Edit the Ant build file, ***ActiveMQInstallDir*/examples/openwire/swissarmy/build.xml**, and 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
```

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 shutdow
```

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
```

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)
```

CHAPTER 5. 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.

5.1. 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.

Defining a secure connection factory

Example 5.1, “Defining a Secure Connection Factory Bean” shows how to create a secure connection factory bean in Spring XML.

Example 5.1. Defining a Secure Connection Factory Bean

```
<bean id="jmsConnectionFactory"
      class="org.apache.activemq.ActiveMQSslConnectionFactory">
  <property name="brokerURL" value="ssl://localhost:61001" />
  <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.

`trustStorePassword`

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

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

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

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

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&amp;period=5000"/>
      <transform><constant>Hello world!</constant></transform>
      <to uri="activemqssl:security.test"/>
    </route>
  </camelContext>
  ...
</beans>
```

CHAPTER 6. 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.

6.1. 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 featues:

- *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. />.

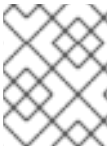
Public key cryptography

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.

6.2. SECURE TRANSPORT PROTOCOLS

Overview

Red Hat JBoss A-MQ 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 6.1, “Secure Transport Protocols” shows the transport protocols that can be secured using SSL/TLS.

Table 6.1. 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.

6.3. JAVA KEYSTORES

Overview

Java keystores provide a convenient mechanism for storing and deploying X.509 certificates and private keys. Red Hat JBoss A-MQ 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 ***JavaInstallDir*/lib/security** or ***JavaInstallDir*/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.

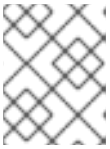
NOTE

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.

6.4. HOW TO USE X.509 CERTIFICATES

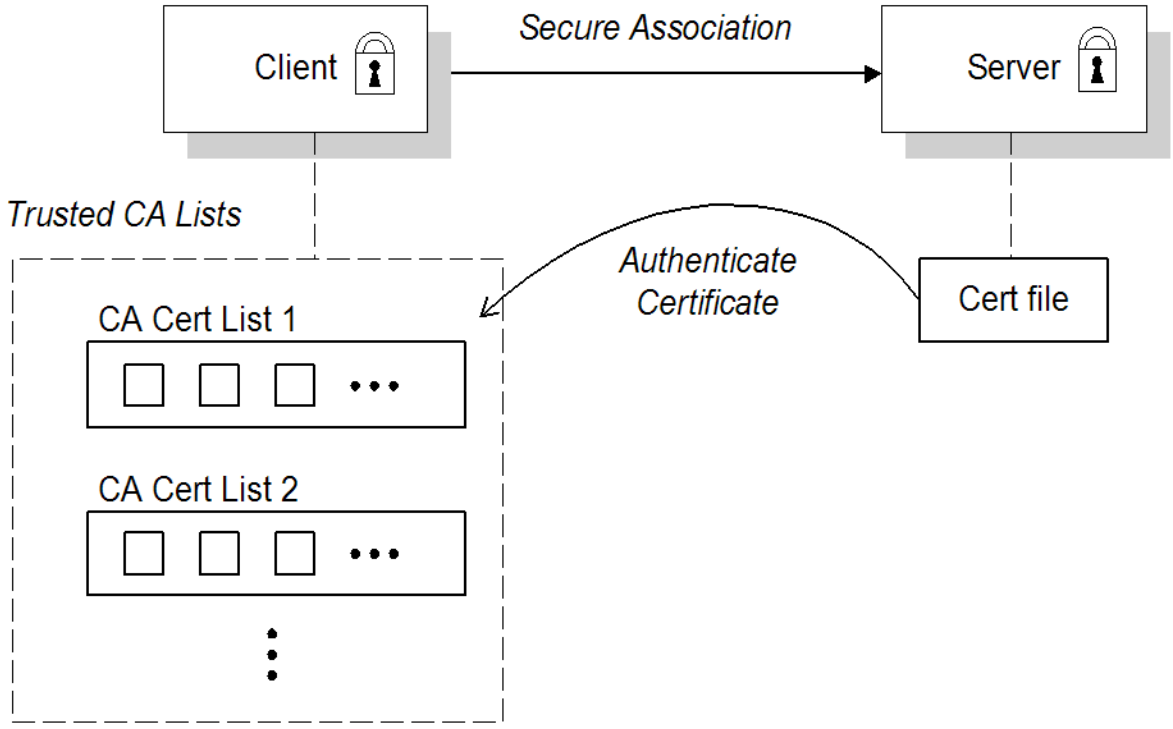
Overview

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 6.1, “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 6.1. Target-Only Authentication Scenario

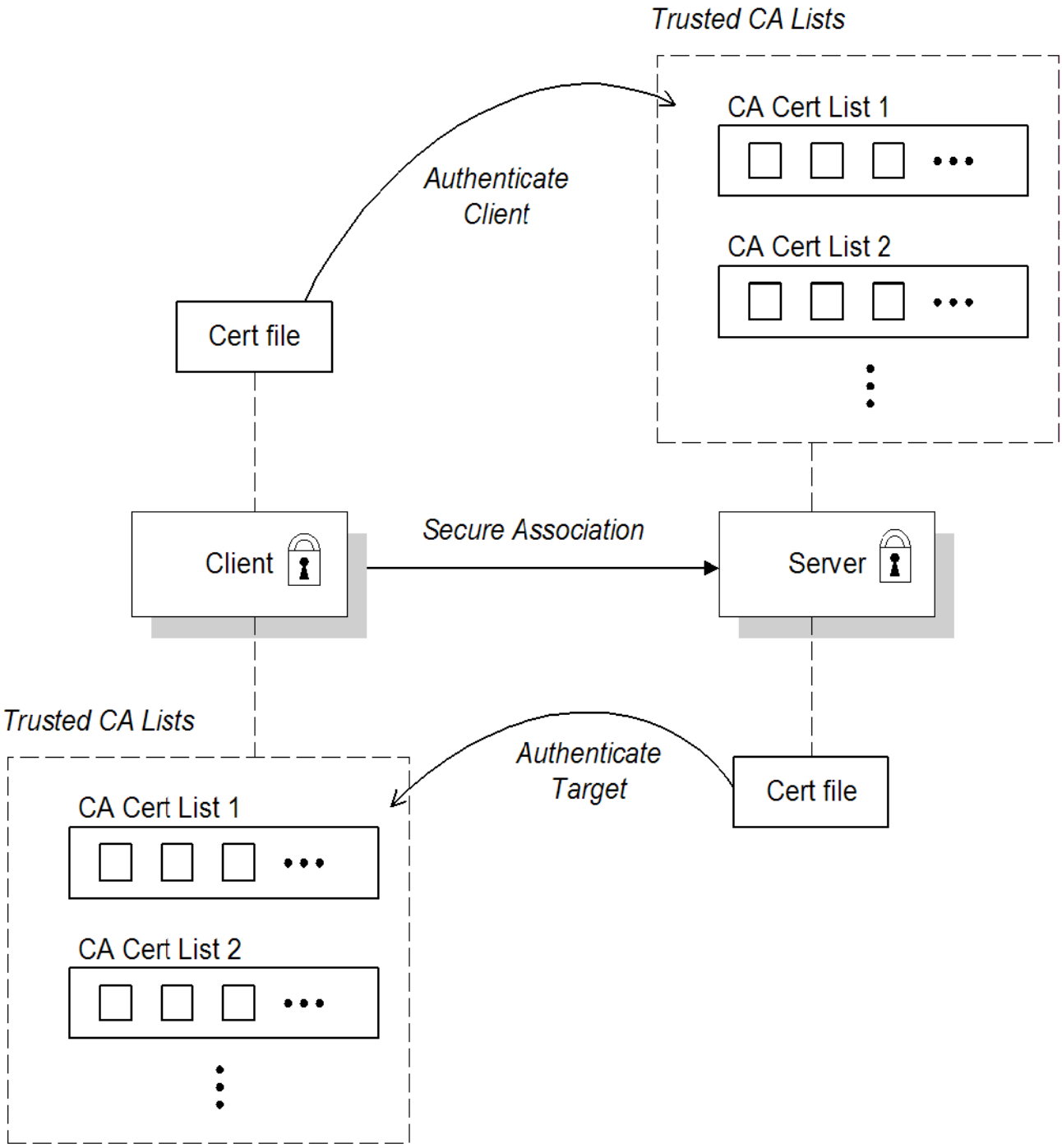


The broker is configured to have its own certificate and private key, which are both stored in the file, **`broker.ks`**. The client is configured to have a trust store, **`client.ts`**, that contains the certificate that originally signed the broker certificate. Normally, the trusted certificate is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

Mutual authentication

In the mutual authentication scenario, as shown in Figure 6.2, “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 6.2. 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.

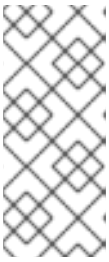
Selecting the authentication scenario

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*.

6.5. CONFIGURING JSSE SYSTEM PROPERTIES

Overview

Java Secure Socket Extension (JSSE) provides the underlying framework for the SSL/TLS implementation in Red Hat JBoss A-MQ. 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 6.2, “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 6.2. 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	Password to access the private key from the keystore file specified by **javax.net.ssl.keyStore**. This password is used twice: • To unlock the keystore file (store password), and • To decrypt the private key stored in the keystore (key password). In other words, the JSSE framework requires these passwords to be identical.
javax.net.ssl.keyStoreType	*(Optional)* 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**.
javax.net.ssl.trustStore	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, ****. 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): 1. **$JAVA_HOME/lib/security/jssecacerts** 2. **$JAVA_HOME/lib/security/cacerts**
javax.net.ssl.trustStorePassword	Password to unlock the keystore file (store password) specified by **javax.net.ssl.trustStore**.
javax.net.ssl.trustStoreType	*(Optional)* 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**.
javax.net.debug	To switch on logging for the SSL/TLS layer, set this property to **ssl**.

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, **-D*Property*=*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
```

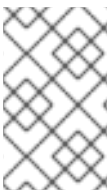
To configure a broker to use the demonstration broker keystore and demonstration broker trust store, you can set the **SSL_OPTS** environment variable as follows, on Windows:

```
set SSL_OPTS=-Djavax.net.ssl.keyStore=C:/Programs/FUSE/fuse-message-
broker-6.1.0.redhat-379/conf/broker.ks
         -Djavax.net.ssl.keyStorePassword=password
         -Djavax.net.ssl.trustStore=C:/Programs/FUSE/fuse-message-broker-
6.1.0.redhat-379/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-
6.1.0.redhat-379/conf/broker.ks
         -Djavax.net.ssl.keyStorePassword=password
         -Djavax.net.ssl.trustStore=/local/FUSE/fuse-message-broker-
6.1.0.redhat-379/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

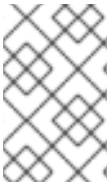
You can also set JSSE system properties using the standard Java API, as long as you set the properties before the relevant transport protocol is initialized. For example:

```
// Java
import java.util.Properties;
...
Properties systemProps = System.getProperties();
systemProps.put(
    "javax.net.ssl.trustStore",
    "C:/Programs/FUSE/fuse-message-broker-6.1.0.redhat-379/conf/client.ts"
);
System.setProperties(systemProps);
```

6.6. 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, **`conf/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**.

6.7. SECURING JAVA CLIENTS

ActiveMQSslConnectionFactory class

To support SSL/TLS security in Java clients, Red Hat JBoss A-MQ 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 6.1, "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 6.1. 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 7. AUTHORIZATION

Abstract

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

7.1. 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 7.1, “Simple Authorization Plug-In Configuration”.

Example 7.1. 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" />
              <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 7.1, “Simple Authorization Plug-In Configuration”).

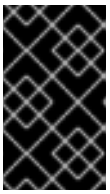
The default value is **org.apache.activemq.jaas.GroupPrincipal**.

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 the 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 JBoss A-MQ 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 7.2, “Setting Access Permissions for Advisory Destinations”.

Example 7.2. 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.

7.2. 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 7.3, “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 7.3. 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:

`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`**.

`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`**.

`authentication`

The authentication method to use when connecting to the LDAP server.

Default is **`simple`**.

`queueSearchBase`

The base DN of queue authorization entries.

Default is **`ou=Queue,ou=Destination,ou=ActiveMQ,ou=system`**.

`topicSearchBase`

The base DN of topic authorization entries.

Default is **ou=Topic,ou=Destination,ou=ActiveMQ,ou=system**.

tempSearchBase

The base DN of authorization entries for temporary destinations.

Default is **ou=Temp,ou=Destination,ou=ActiveMQ,ou=system**.

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**.

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**.

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**.

7.3. 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 7.4, “LDAP Authorization Plug-In Configuration”.

Example 7.4. 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.ldap.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=syste
m"/>
            <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`

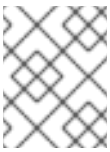
Must always be set to **`com.sun.jndi.ldap.LdapCtxFactory`**.

`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

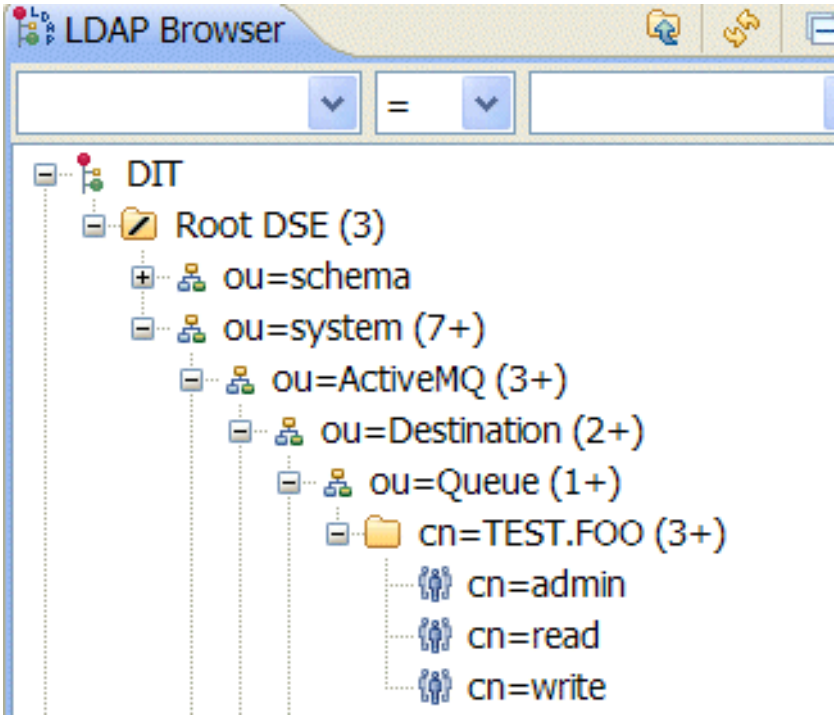
Specifies an LDAP search filter, which is used when looking up the *admin permissions* for any kind of queue or topic. The search filter attempts to match one of the children (or descendants, if **SUBTREE_SCOPE** is enabled) of the queue or topic node.

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
objectClass	*groupOfNames (structural)*
objectClass	*top (abstract)*
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.

7.4. 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 7.5, “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 7.5. 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 7.5, “Implementation of MessageAuthorizationPolicy”, add the following lines to the broker configuration file, **conf/activemq.xml**, inside the **broker** element:

```
<broker>
  ...
  <messageAuthorizationPolicy>
    <bean class="com.acme.MsgAuthzPolicy"
          xmlns="http://www.springframework.org/schema/beans"/>
  </messageAuthorizationPolicy>
  ...
</broker>
```

CHAPTER 8. 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.

8.1. TUTORIAL OVERVIEW

Goals

In this tutorial you will:

- install Apache Directory Studio
- add user entries into the LDAP server
- add groups to manage security roles
- configure Red Hat JBoss A-MQ to use LDAP authentication
- configure JBoss A-MQ to use roles for authorization
- configure SSL/TLS connections to the LDAP server

Tutorial stages

The tutorial consists of the following stages:

1. Section 8.2, “Set-up a Directory Server and Browser”.
2. Section 8.3, “Add User Entries to the Directory Server”.
3. Section 8.4, “Enable LDAP Authentication in the OSGi Container”.
4. Section 8.5, “Configuring Access to OSGi Administrative Functions”.
5. Section 8.8, “Enable SSL/TLS on the LDAP Connection”.
6. Section 8.6, “Add Authorization Entries”.
7. Section 8.7, “Enable LDAP Authorization in the Broker”.

8.2. SET-UP A DIRECTORY SERVER AND BROWSER

Overview

In this stage of the tutorial you will install an X.500 directory server and browser client from the *Apache Directory* project. These applications will be used throughout the rest of this tutorial.

Install Apache Directory Studio

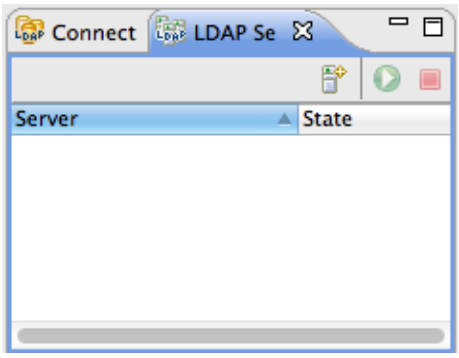
To install Apache Directory Studio:

1. Download the Apache Directory Studio RCP Application (Eclipse-based, standalone executable) for your platform:
   - Download for Linux
   - Download for Mac OS X
   - Download for Windows
2. Follow the *Installation* instructions on the relevant download page.
3. Start Apache Directory Studio by double-clicking the relevant icon to launch the application.

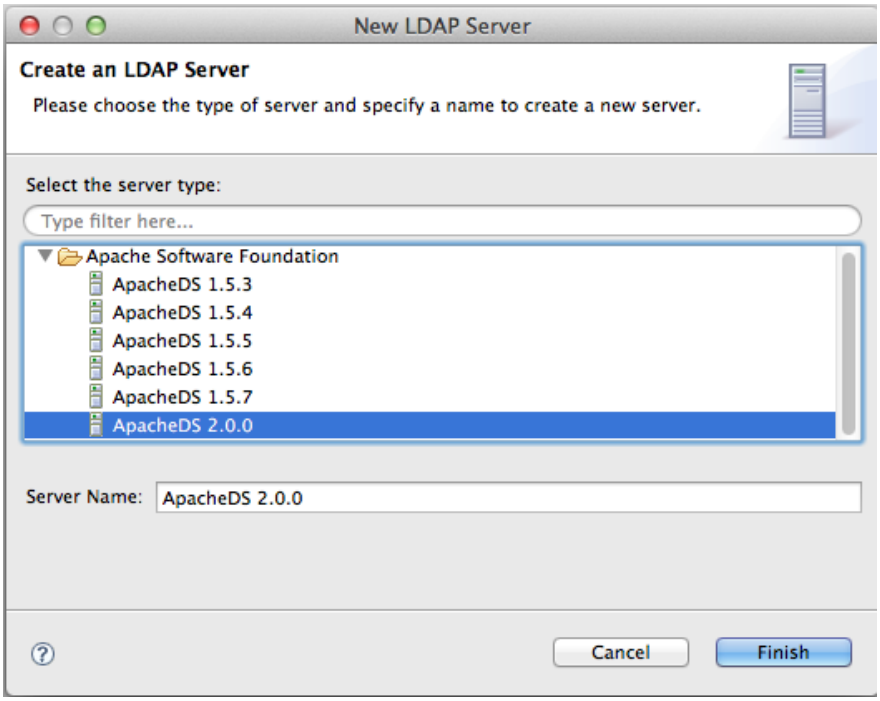
Create an embedded Directory Server instance

Apache Directory Studio is able to create embedded Directory Server instances. This is a convenient way to access a Directory Server for the purpose of running examples and tutorials. To create an embedded Directory Server instance:

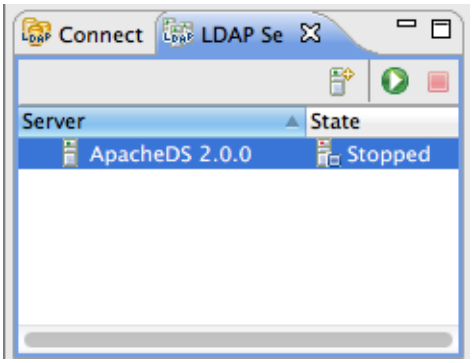
1. Start up Apache Directory Studio.
2. In the lower left corner of the screen, click on the **LDAP Server** tab to access the **LDAP Server** view.



3. While hovering your mouse over the **LDAP Server** view, open the context menu and select **New | New Server**. The **Create an LDAP Server** dialog opens, as shown. Select **Apache 2.0.0** and click **Finish**.



4. To start the new Directory Server instance, select **`Apache 2.0.0`** in the **`LDAP Server`** view and click the **Run** icon  (or select **Run** from the context menu).

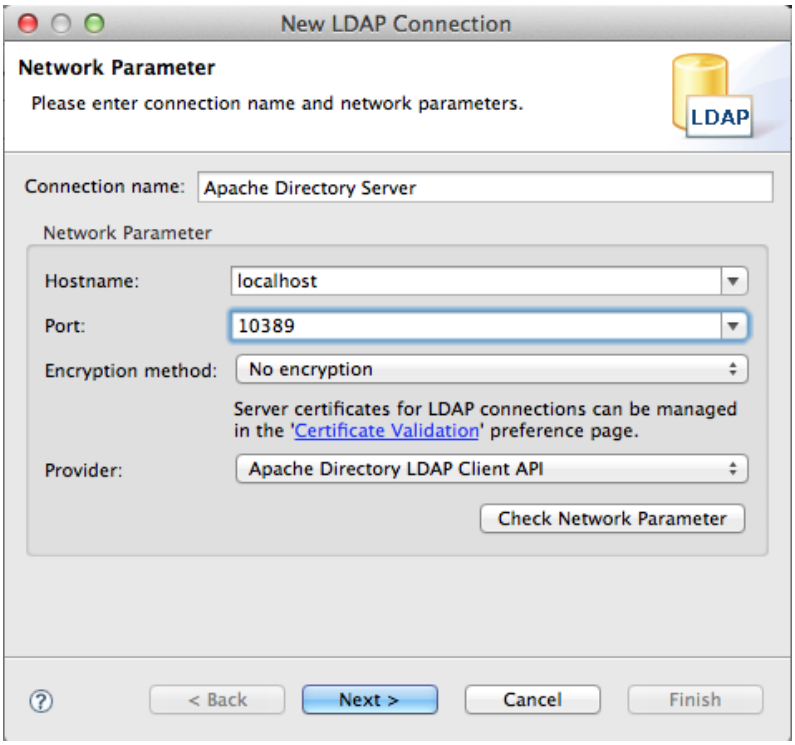


Connect the LDAP browser to the server

To connect the LDAP browser to the LDAP server:

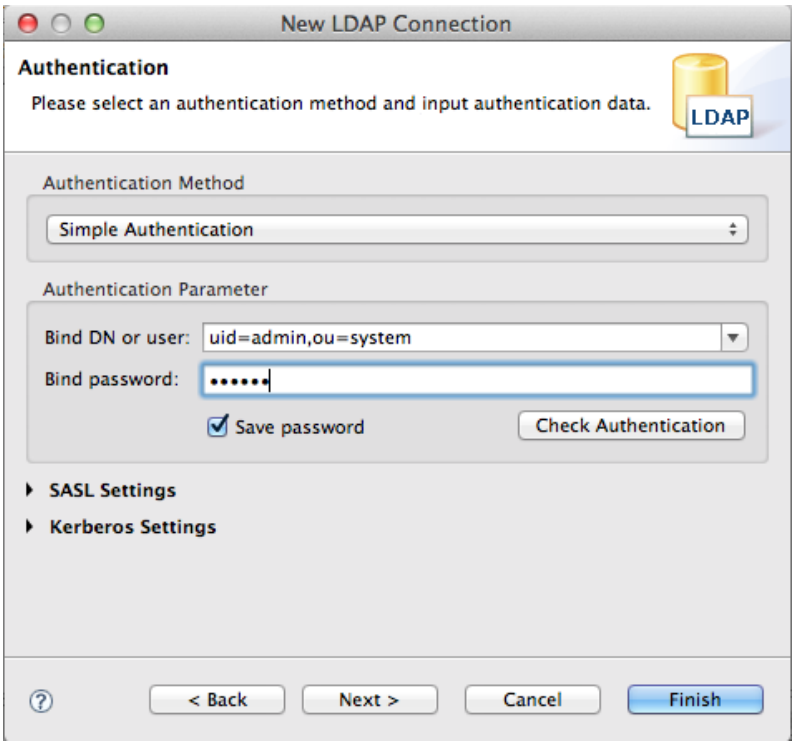
1. Right-click inside the **`Connections`** view.
2. Select **New Connection...** to open the **`New LDAP Connection`** wizard on the **`Network Parameter`** page.

Figure 8.1. Network Parameter



3. In the **`Connection name`** field, enter **`Apache Directory Server`**.
4. In the **`Hostname`** field enter **`localhost`**.
5. In the **`Port`** field, enter **`10389`**.
6. Click **`Next`** to open the **`Authentication`** page.

Figure 8.2. Authentication Step of New LDAP Connection



7. In the **`Bind DN or user`** field, enter **`uid=admin,ou=system`**).

8. In the **Bind password** field, enter **secret**).

9. Click **Finish**.

If the connection is successfully established, an outline of the Directory Information Tree (DIT) appears in the **LDAP Browser** view.

8.3. 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.

Goals

In this portion of the tutorial you will

- add three user entries to the LDAP server
- add four groups to the LDAP server

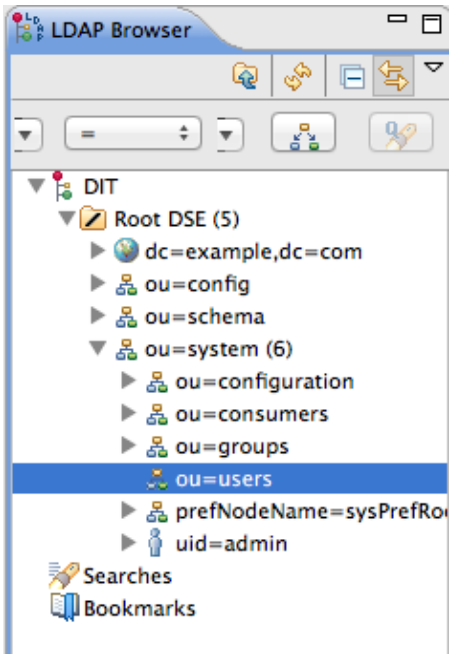
Adding user entries

Perform the following steps to add user entries to the directory server:

1. Ensure that the LDAP server and browser are running.

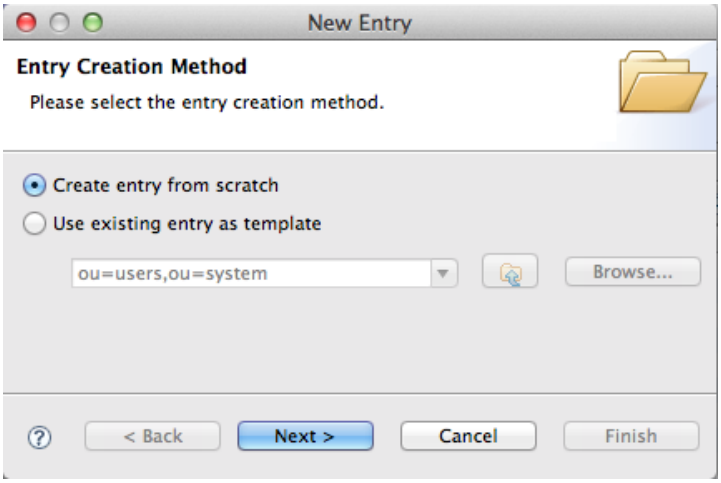
   See Section 8.2, “Set-up a Directory Server and Browser”.

2. In the **LDAP Browser** view, drill down to the **ou=users** node.

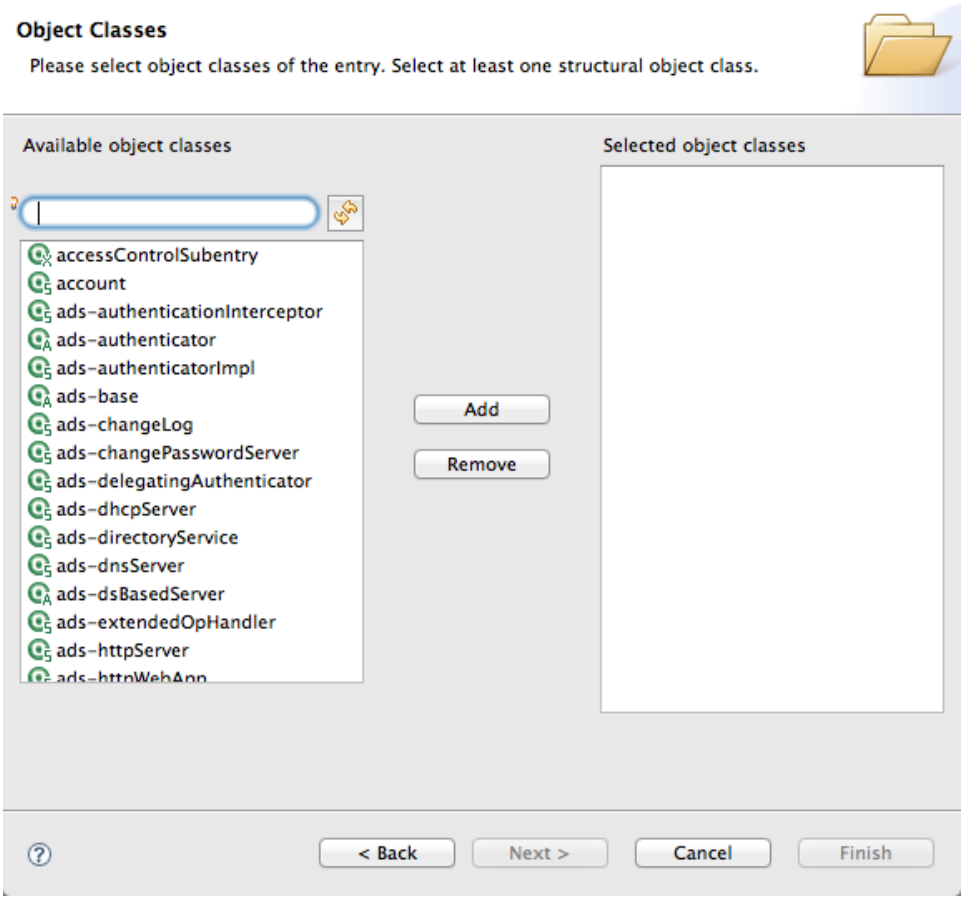


3. Select the **ou=users** node.

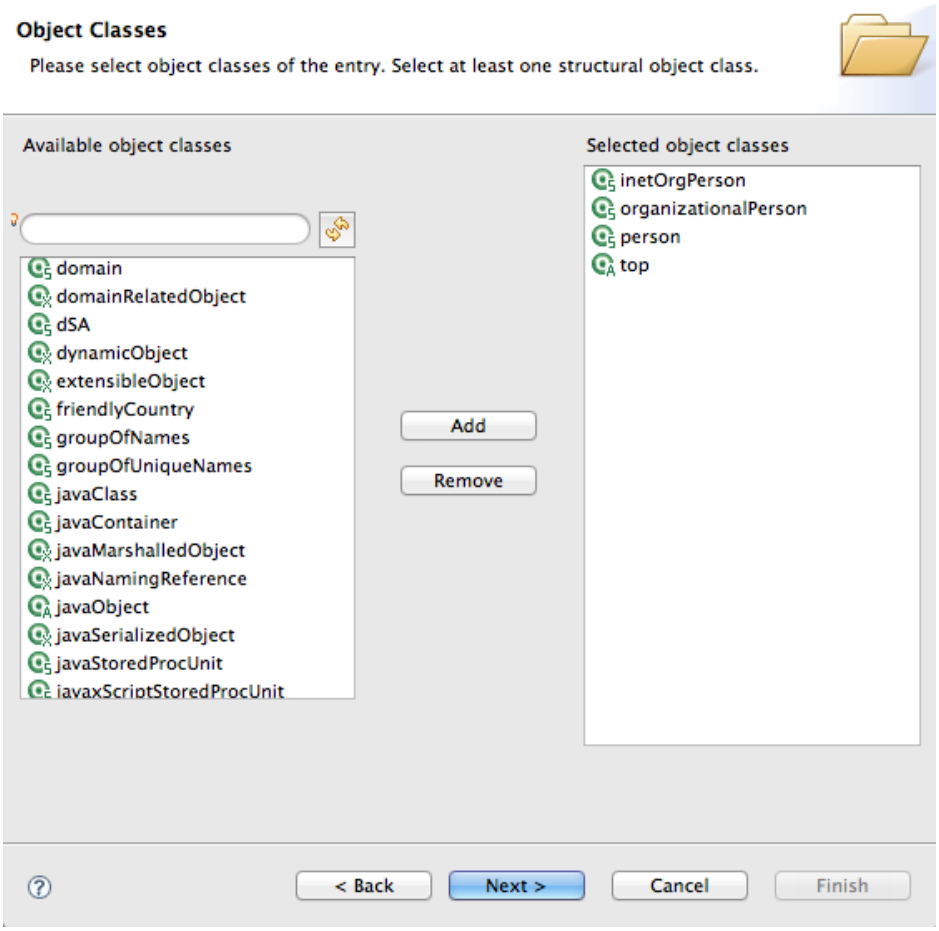
4. Open the context menu.
5. Select **New** → **New Entry** to open the **`Entry Creation Method`** pane.



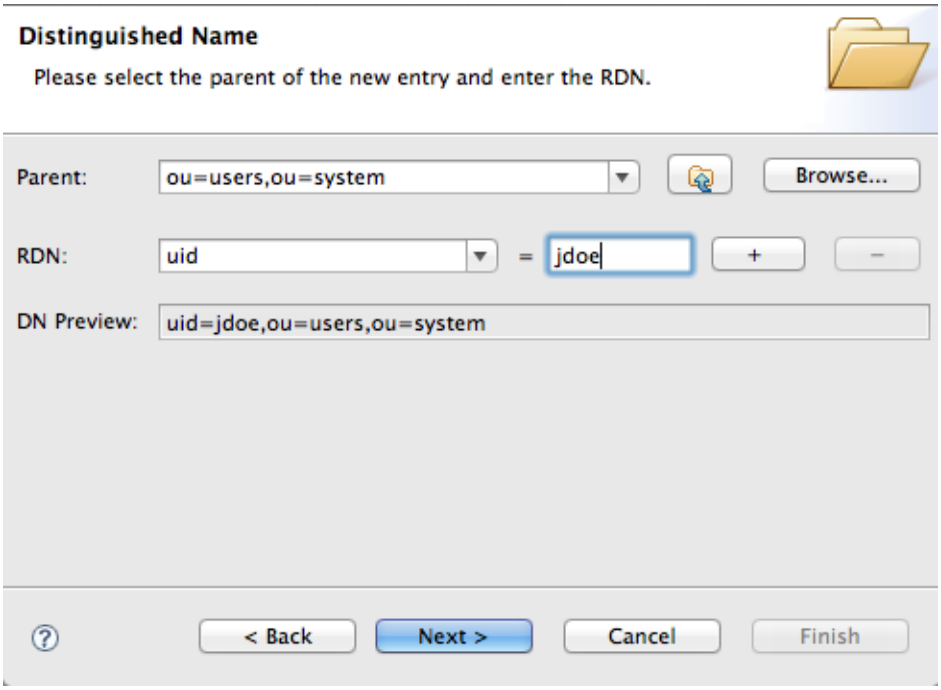
6. Check **`Create entry from scratch`**.
7. Click **`Next`** to open the **`Object Classes`** pane.



8. Select **`inetOrgPerson`** from the list of **`Available object classes`**.
9. Click **`Add`** to populate the list of **`Selected object classes`**.

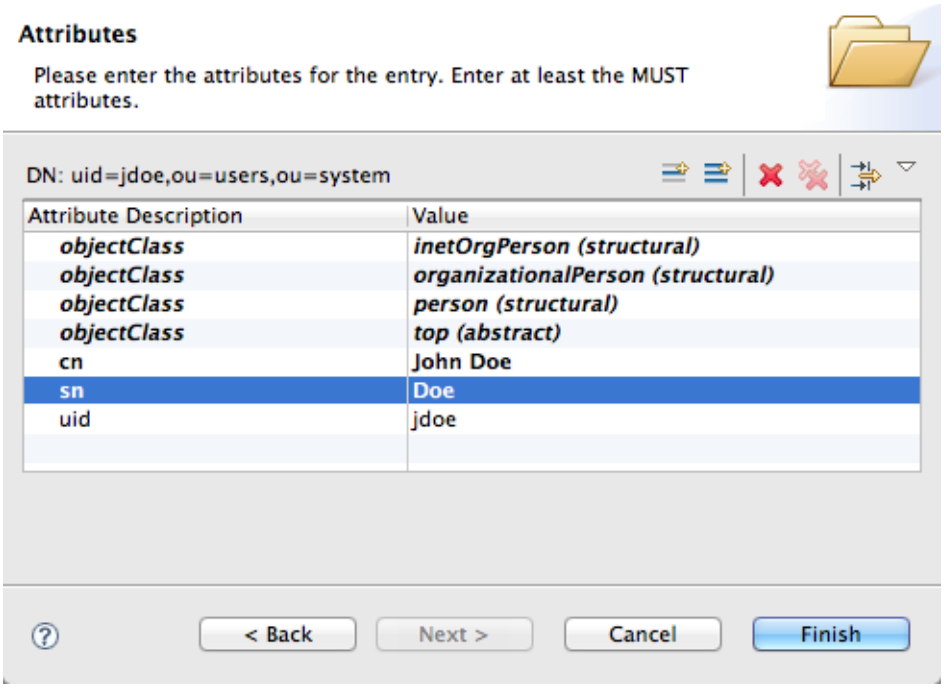


10. Click **`Next`** to open the **`Distinguished Name`** pane.



11. In the the **`RDN`** field, enter **`uid`** in front and **`jdoe`** after the equals sign.

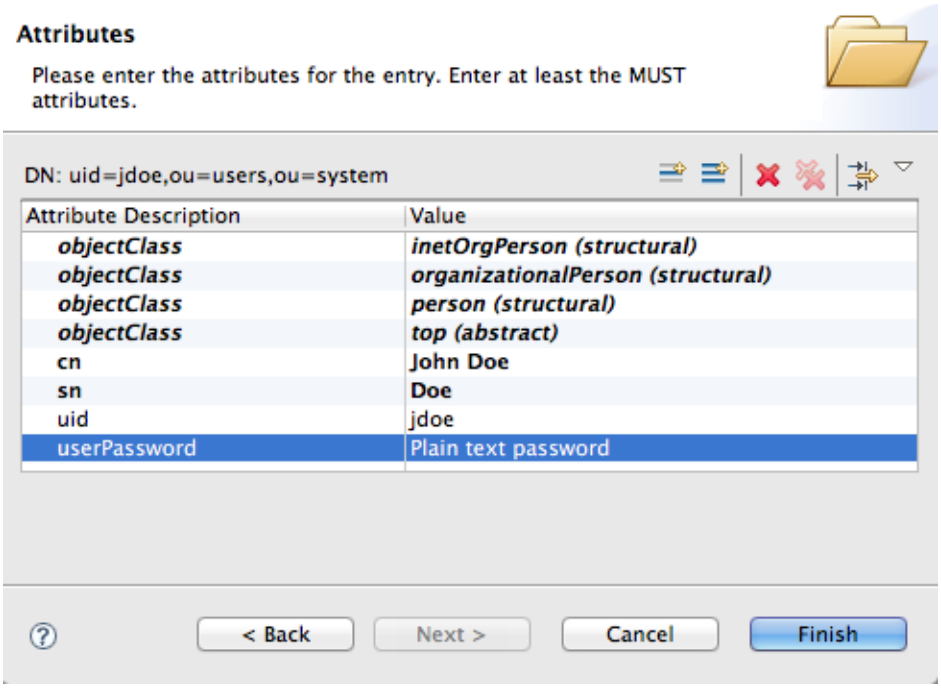
12. Click **`Next`** to open the **`Attributes`** pane.



13. Fill in the remaining mandatory attributes in the **Attributes** pane.
    a. Set the **cn** (common name) attribute to **John Doe**
    b. Set the **sn** (surname) attribute to **Doe**.
14. Add a **userPassword** attribute to the user entry.
    a. Open the context menu in the **Attributes** pane.
    b. Select **New Attribute** to open the **New Attribute** wizard.
    c. From the **Attribute type** drop-down list, select **userPassword**.
    d. Click **Finish**.

       The **Password Editor** dialog opens.
    e. In the **Enter New Password** field, enter the password, **secret**.
    f. Click **OK**.

       The **userPassword** attribute appears in the **Attributes** editor.



15. Click **`Finish`**.
16. Add a user **`Jane Doe`** by following Step 3 to Step 15.

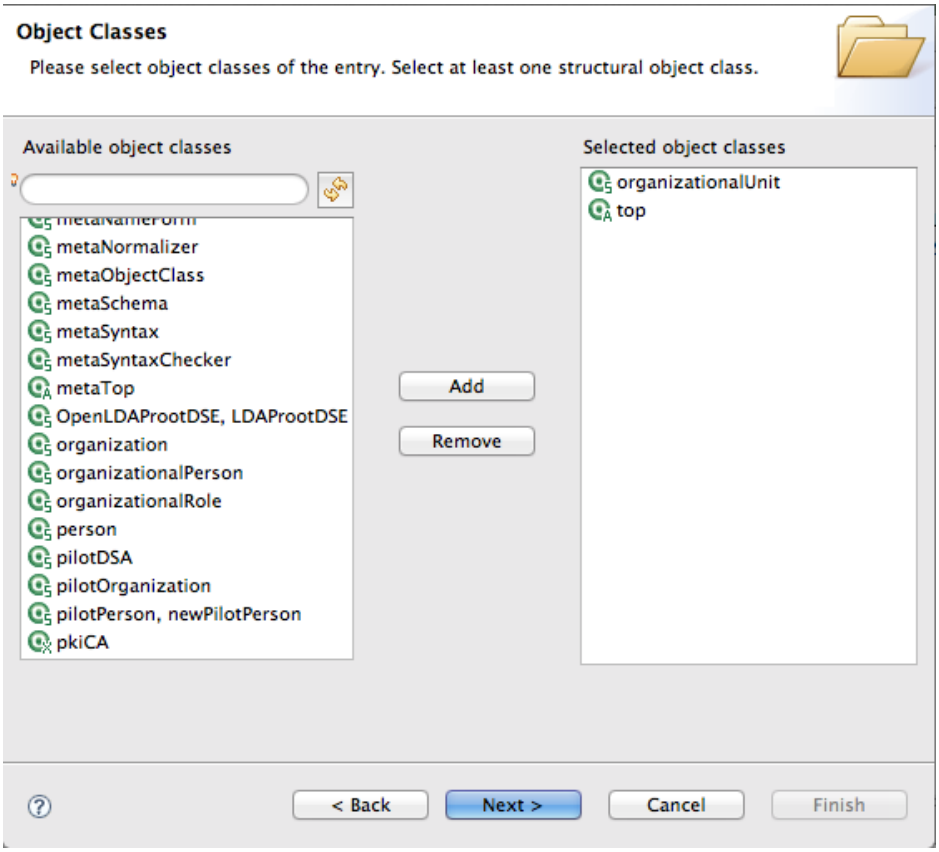
    In Step 11, use **`janedoe`** for the new user's **`uid`**.
17. Add a user **`Camel Rider`** by following Step 3 to Step 15.

    In Step 11, use **`crider`** for the new user's **`uid`**.

Adding groups for the roles

To add the groups that define the roles:

1. Create a new organizational unit to contain the role groups.
    a. In the **`LDAP Browser`** view, select the **`ou=system`** node.
    b. Open the context menu.
    c. Select **New** → **New Entry** to open the **`Entry Creation Method`** pane.
    d. Check **`Create entry from scratch`**.
    e. Click **`Next`** to open the **`Object Classes`** pane.
    f. Select **`organizationalUnit`** from the list of **`Available object classes`**.
    g. Click **`Add`** to populate the list of **`Selected object classes`**.



h. Click **Next>** to open the **Distinguished Name** pane.

i. In the the **RDN** field, enter **ou** in front and **roles** after the equals sign.

j. Click **Next>** to open the **Attributes** pane.

k. Click **Finish**.

NOTE

This step is required because Apache DS allows only administrators access to entries in **ou=system,ou=groups**.

2. In the **LDAP Browser** view, drill down to the **ou=roles** node.
3. Select the **ou=roles** node.
4. Open the context menu.
5. Select **New** → **New Entry** to open the **Entry Creation Method** pane.
6. Check **Create entry from scratch**.
7. Click **Next** to open the **Object Classes** pane.
8. Select **groupOfNames** from the list of **Available object classes**.
9. Click **Add** to populate the list of **Selected object classes**.
10. Click **Next** to open the **Distinguished Name** pane.

11. In the the **RDN** field, enter **cn** in front and **admin** after the equals sign.
12. Click **Next** to open the **Attributes** pane.

    The message **Attribute "member" has an empty value, please insert a valid value.** displays, and a DN Editor opens for you to enter a value.
13. Enter **uid=jdoe**.
14. Click **OK**.
15. Click **Finish**.
16. Add a **sshConsole** role by following Step 3 to Step 15.

    In Step 11, use **sshConsole** for the new group's **cn**.

    In Step 13, use **uid=janedoe**.
17. Add a **webconsole** role by following Step 3 to Step 15.

    In Step 11, use **webconsole** for the new group's **cn**.

    In Step 13, use **uid=janedoe**.
18. Add a **jmxUser** role by following Step 3 to Step 15.

    In Step 11, use **jmxUser** for the new group's **cn**.

    In Step 13, use **uid=crider**.

8.4. ENABLE LDAP AUTHENTICATION IN THE OSGI CONTAINER

Overview

In this part of the tutorial you will 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.

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 JBoss A-MQ by entering the following command in a terminal window:

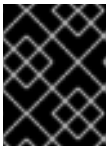
   ```
   amq
   ```
3. Create a Blueprint configuration file called **ldap-module.xml**.
4. Copy Example 8.1, “JAAS Realm for Standalone” into **ldap-module.xml**.

   Example 8.1. 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="1">
    <jaas:module
className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule"
                 flags="required">
      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
      role.base.dn=ou=roles,ou=system
      role.name.attribute=cn
      role.filter=(member=uid=%u)
      role.search.subtree=true
      authentication=simple
    </jaas:module>
  </jaas:config>
</blueprint>
```

This login module creates a JAAS realm called **karaf**, which is the same name as the default JAAS realm used by Red Hat JBoss A-MQ. By redefining this realm with a **rank** attribute value greater than **0**, it overrides the standard **karaf** realm which has the rank **0**. For more information on configuring a JAAS realm see Section 2.1.2, “Defining JAAS Realms”.

For a detailed description of configuring JBoss A-MQ to use LDAP see Section 2.2, “Enabling LDAP Authentication”.



IMPORTANT

When setting the JAAS properties above, do *not* enclose the property values in double quotes.

TIP

If you use OpenLDAP, the syntax of the role filter is **(member:=uid=%u)**.

5. To deploy the new LDAP module, copy the **ldap-module.xml** into the JBoss A-MQ **deploy/** directory.

   The LDAP module is automatically activated.

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:

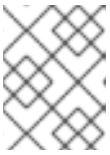
```
./amq
```

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 8.2, “JAAS Realm for Fabric” into the **ldap-module.xml** resource, customizing the value of the **rank** attribute and the **connection.url** property, as necessary.

Example 8.2. 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">

  <jaas:config name="karaf" rank="2">
    <jaas:module
className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule"
                 flags="required">
      initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
      connection.username=uid=admin,ou=system
      connection.password=secret
      connection.protocol=
      connection.url=ldap://LDAPHost:10389
```

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

Where ***LDAPHost*** is the name of the host where the LDAP server is running. You must be sure to use a hostname that is accessible to all of the containers in the fabric (for example, you cannot use **localhost** as the hostname here).

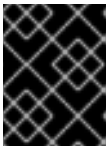
Save and close the **ldap-module.xml** resource by typing Ctrl-S and Ctrl-X.

This login module creates a JAAS realm called **karaf**, which is the same name as the default JAAS realm used by Red Hat JBoss A-MQ. By redefining this realm with a **rank** of **2**, it overrides all of the previously installed **karaf** realms. For more information on configuring a JAAS realm see Section 2.1.2, “Defining JAAS Realms”.

For a detailed description of configuring JBoss A-MQ to use LDAP see Section 2.2, “Enabling LDAP Authentication”.

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.

TIP

If you use OpenLDAP, the syntax of the role filter is **(member:=uid=%u)**.

6. 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.

7. 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.

Test the LDAP authentication

Test the new LDAP realm by connecting to the running container using the Red Hat JBoss A-MQ **client** utility, as follows:

1. Open a new command prompt.
2. change directory to the JBoss A-MQ ***InstallDir*/bin** directory.
3. Enter the following command to log on to the running container instance using the identity **janedoe**:

```
client -u janedoe -p secret
```

You should receive the following message:

```
Authentication failure
```

This fails because **janedoe** does not have the **admin** role which is required for using the remote console.

4. Enter the following command to log on to the running container instance using the identity **jdoe**:

```
client -u jdoe -p secret
```

You should successfully log into the container's remote console because **jdoe** does have the **admin** role.

5. Log off the remote console by entering the **logout** command.

8.5. CONFIGURING ACCESS TO OSGI ADMINISTRATIVE FUNCTIONS

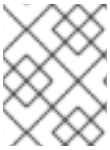
Overview

This tutorial explains how to configure the OSGi administrative functions to use specific roles for authorization. By configuring each of the administrative functions to use a different role for access, you can provide fine grained control over who can monitor and manipulate running containers.

When LDAP is enabled, the OSGi container expects the user role data to be stored along with the user authentication data in the LDAP directory server. The LDAP search query to extract the role data is specified by the **`role.*`** properties in the **`jaas:module`** element.

The JAAS LDAP login module used in this tutorial, shown in Example 8.1, “JAAS Realm for Standalone”, is configured to extract the role name from the **`cn`** property of all entries selected by the filter **`member=uid=%u`** which is run on the tree selected using the base DN **`ou=roles,ou=system`**. In the section called “Adding groups for the roles”, you added three groups to the **`ou=roles,ou=system`** tree. The filter will match with any group that has a member specified by **`uid=%u`**.

For example, when you attempted to connect to the remote console as user **`jdoe`** the filter searched for a group with a member **`uid=jdoe`** and matched on the group **`cn=admin,ou=roles,ou=system`**. The LDAP module extracted the **`cn`** property's value of **`admin`** and used it as the role for authorizing user **`jdoe`**.



NOTE

The instructions in this section are applicable only to a standalone OSGi container. They do not apply to Fabric containers.

Goals

You will change the role used for each of the administrative functions:

- SSH (remote console login)
- JMX management

Prerequisites

Before you can perfrom any of the following tutorials, you must ensure that the ApacheDS server is running.

Configure a role for the remote console

To configure a role for the remote console:

1. Open ***`InstallDir`*`/etc/org.apache.karaf.shell.cfg`** in a text editor.
2. Add the following line:

   ```
   sshRole=sshConsole
   ```

3. Save the changes.
4. Start Red Hat JBoss A-MQ by entering the following command in a terminal window:

   ```
   amq
   ```

5. Open a new command prompt.
6. Change directory to the JBoss A-MQ ***InstallDir*/bin** directory.
7. Enter the following command to log on to the running container instance using the identity **janedoe**:

```
client -u janedoe -p secret
```

   You should successfully log into the container's remote console because **janedoe** does have the **sshConsole** role.

Configure a role for JMX access

To configure a role for JMX access:

1. Open ***InstallDir*/etc/org.apache.karaf.management.cfg** in a text editor.
2. Add the following line:

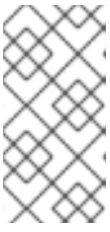
```
jmxRole=jmxUser
```

3. Save the changes.
4. Start JBoss A-MQ by entering the following command in a terminal window:

```
amq
```

5. Start JConsole or another JMX console.
6. Connect to JBoss A-MQ's JMX server using the following settings:
   - JMX URL: **service:jmx:rmi://localhost:44444/jndi/rmi://localhost:1099/karaf-root**
   - User: **jdoe**
   - Password: **secret**

   The connection will fail because **jdoe** user does not have the **jmxUser** role.



NOTE

To test security, you must log on using the **Remote process** option (even though you are logging on to a JVM instance that is running on your local machine). If you log on using the **Local process** option, JConsole bypasses the authentication check.

7. Connect to JBoss A-MQ's JMX server as using the following settings:

- JMX URL: **`service:jmx:rmi://localhost:44444/jndi/rmi://localhost:1099/karaf-root`**
- User: **`crider`**
- Password: **`secret`**

The connection will succeed because **`crider`** user does have the **`jmxUser`** role.

More information

For more information on configuring the JBoss A-MQ LDAP login module see Section 2.2, “Enabling LDAP Authentication”.

For more information on configuring the JBoss A-MQ administrative functions see Section 2.3, “Configuring Roles for the Administrative Protocols”.

8.6. ADD 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.

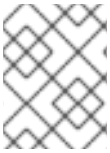
Using wildcards in queue and topic entries

When setting the common name of queue and topic entries in the directory server, you can use any of the wildcards shown in Table 8.1, “Destination Name Wildcards in LDAP” to match one or more segments of a destination name.

Table 8.1. 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:

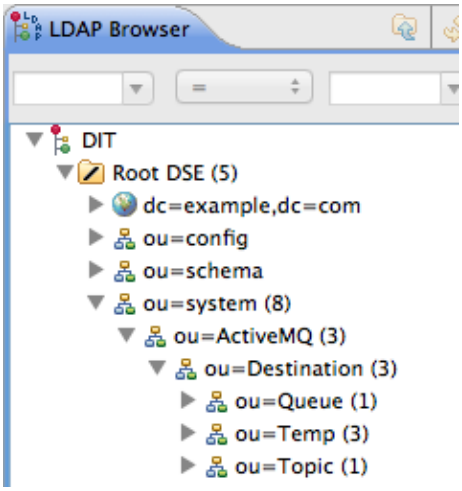
1. The next few steps describe how to create the **ou=ActiveMQ** node.
    a. Right-click on the **ou=system** node and select **New → New Entry**. The **New Entry** wizard appears.
    b. In the **Entry Creation Method** pane, select the **Create entry from scratch** radiobutton. Click **Next**.
    c. In the **Object Classes** pane, select **organisationalUnit** from the list of **Available object classes** on the left and then click **Add** to populate the list of **Selected object classes**. Click **Next**.
    d. In the **Distinguished Name** pane, complete the **RDN** field, putting **ou** in front and **ActiveMQ** after the equals sign. Click **Next** and then click **Finish**.
2. The next few steps describe how to create the **ou=Destination**, **ou=Queue**, **ou=Topic**, and **ou=Temp** nodes.
    a. Right-click on the **ou=ActiveMQ** node and select **New → New Entry**. The **New Entry** wizard appears.
    b. In the **Entry Creation Method** pane, select the **Create entry from scratch** radiobutton. Click **Next**.
    c. In the **Object Classes** pane, select **organisationalUnit** from the list of **Available object classes** on the left and then click **Add** to populate the list of **Selected object classes**. Click **Next**.
    d. In the **Distinguished Name** pane, complete the **RDN** field, putting **ou** in front and **Destination** after the equals sign. Click **Next** and then click **Finish**.

e. In a similar manner to the preceding steps, by right-clicking on the **ou=Destination** node and invoking the **New Entry** wizard, create the following **organisationalUnit** nodes as children of the **ou=Destination** node:

```
ou=Queue,ou=Destination,ou=ActiveMQ,ou=system
ou=Topic,ou=Destination,ou=ActiveMQ,ou=system
ou=Temp,ou=Destination,ou=ActiveMQ,ou=system
```

3. In the LDAP Browser window, you should now see the following tree:

Figure 8.3. DIT after Creating Destination, Queue, Topic and Temp Nodes



4. The next few steps describe how to create the following nodes:

```
cn=$,ou=Queue,ou=Destination,ou=ActiveMQ,ou=system
cn=ActiveMQ.Advisory.$,ou=Topic,ou=Destination,ou=ActiveMQ,ou=system
```

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 → New Entry**. The **New Entry** wizard appears.

b. In the **Entry Creation Method** pane, select the **Create entry from scratch** radiobutton. Click **Next**.

c. In the **Object Classes** pane, select **applicationProcess** from the list of **Available object classes** on the left and then click **Add** to populate the list of **Selected object classes**. Click **Next**.

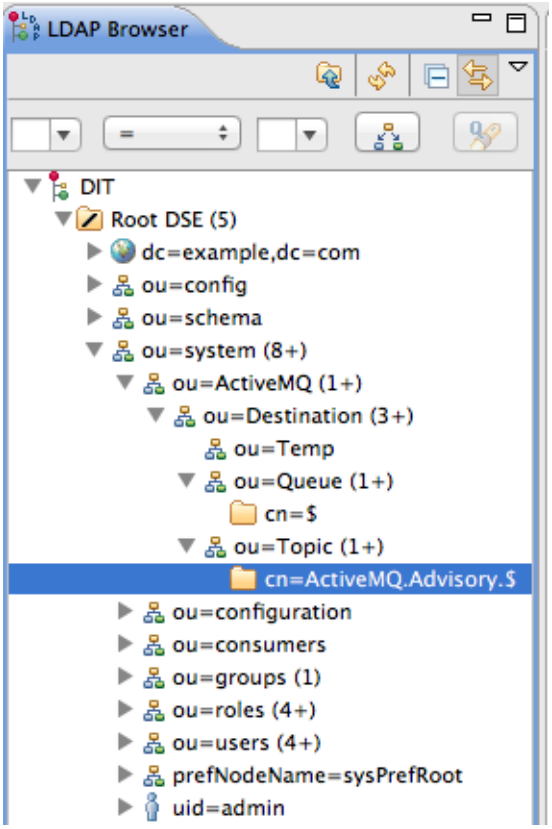
d. In the **Distinguished Name** pane, complete the **RDN** field, putting **cn** in front and **$** after the equals sign (where **$** represents the wildcard that matches any queue name). Click **Next** and then click **Finish**.

e. In a similar manner to the preceding steps, by right-clicking on the **ou=Topic** node and invoking the **New Entry** wizard, create the following **applicationProcess** node as a child of the **ou=Topic** node:

```
cn=ActiveMQ.Advisory.$,ou=Topic,ou=Destination,ou=ActiveMQ,ou=sys
tem
```

5. In the LDAP Browser window, you should now see the following tree:

Figure 8.4. DIT after Creating Children of Queue and Topic Nodes



6. The next few steps describe how to create nodes that represent **admin**, **read**, and **write** permissions for the queues and topics.

   a. Right-click on the **cn=$** node and select **New** → **New Entry**. The **New Entry** wizard appears.

   b. In the **Entry Creation Method** pane, select the **Create entry from scratch** radiobutton. Click **Next**.

   c. In the **Object Classes** pane, select **groupOfNames** from the list of **Available object classes** on the left and then click **Add** to populate the list of **Selected object classes**. Click **Next**.

   d. In the **Distinguished Name** pane, complete the **RDN** field, putting **cn** in front and **admin** after the equals sign. Click **Next**.

   e. You are now prompted to provide a value for the mandatory **member** attribute, through the **DN Editor** dialog. In the text field, enter **cn=admin,ou=roles,ou=system**. Click **Ok**.

NOTE

The **cn=admin,ou=roles,ou=system** references a role that was created for the Apache Karaf JAAS authentication plug-in in a previous tutorial. These roles can be reused by the Apache ActiveMQ authorization plug-in, under certain conditions. See the section called “Compatibility with Apache Karaf principals” for details.

f. Click **Finish**, to close the **New Entry** wizard.

g. In a similar manner to the preceding steps, by right-clicking on the **cn=$** node and invoking the **New Entry** wizard, create the following additional **groupOfNames** nodes as children of the **cn=$** node:

```
cn=read,cn=$,ou=Queue,ou=Destination,ou=ActiveMQ,ou=system
cn=write,cn=$,ou=Queue,ou=Destination,ou=ActiveMQ,ou=system
```

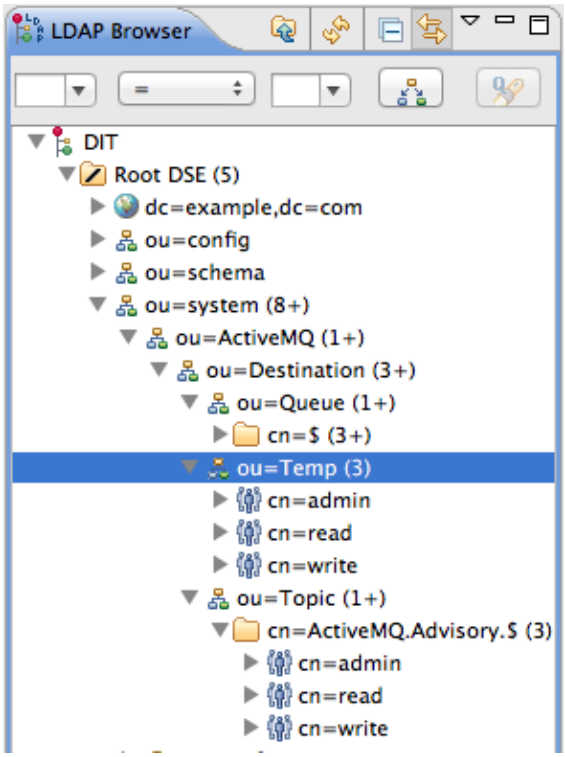
7. Copy the **cn=admin**, **cn=read**, and **cn=write** permission nodes and paste them as children of the **cn=ActiveMQ.Advisory.$** node, as follows.

   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 8.5. DIT after Creating Children of Queue, Topic and Temp Nodes



8.7. 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 8.3, “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 8.6, “Add 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 DNs 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 JBoss A-MQ container, if it is currently running. In the console window, enter the following command:

   ```
   JBossA-MQ:karaf@root> shutdown -f
   ```

2. Add the LDAP authorization plug-in to the broker configuration. Open the broker configuration file, ***`InstallDir`*`/etc/activemq.xml`**, with a text editor and add the **`authorizationPlugin`** element, as follows:

   ```
   <beans ...>
     <broker ...>
       ...
       <plugins>
         ...
         <authorizationPlugin>
           <map>
               <cachedLDAPAuthorizationMap
                   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="300000"
                legacyGroupMapping="false"

groupClass="org.apache.karaf.jaas.boot.principal.RolePrincipal"
            />
        </map>
      </authorizationPlugin>
    </plugins>
    ...
  </broker>
</beans>
```

3. Ensure that the X.500 directory server is running. If necessary, manually restart the X.500 directory server—see Section 8.2, “Set-up a Directory Server and Browser”. If the server is not running, all broker connections will fail.

4. Restart the JBoss A-MQ 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 **`authorizationPlugin`** element, as follows:

```
<beans ...>
  <broker ...>
    ...
    <plugins>
      ...
      <authorizationPlugin>
```

```
        <map>
            <cachedLDAPAuthorizationMap
                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="300000"
                legacyGroupMapping="false"

groupClass="org.apache.karaf.jaas.boot.principal.RolePrincipal"
            />
        </map>
      </authorizationPlugin>
    </plugins>
    ...
  </broker>
</beans>
```

Save and close the **broker.xml** resource by typing Ctrl-S and Ctrl-X.

4. Ensure that the X.500 directory server is running. If necessary, manually restart the X.500 directory server—see Section 8.2, “Set-up a Directory Server and Browser”. If the server is not running, all broker connections will fail.

5. 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 JBoss A-MQ package. But you can obtain the sample clients by installing the Apache ActiveMQ kit, **apache-activemq-5.9.0.redhat-610379-bin.zip**, provided in the **extras/** directory of the JBoss A-MQ installation.

Install the Apache ActiveMQ kit as follows:

1. Find the Apache ActiveMQ kit at the following location:

```
InstallDir/extras/apache-activemq-5.9.0.redhat-610379-bin.zip
```

2. Using a suitable archive utility on your platform, unzip the **apache-activemq-5.9.0.redhat-610379-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 **jdoe** 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=jdoe -
Dpassword=secret
```

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
```

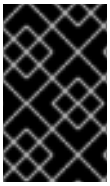
   This time, the consumer client fails, because **janedoe** does not belong to the **admin** group.

8.8. ENABLE SSL/TLS ON THE LDAP CONNECTION

Overview

This tutorial explains how to enable SSL/TLS security on the connection between the LDAP login module and the Apache Directory Server.

The Apache Directory Server is already configured with an SSL endpoint. The default configuration creates an LDAPS endpoint that listens on the IP port **10636**. The directory server automatically generates a self-signed X.509 certificate which it uses to identify itself during the SSL/TLS handshake.



IMPORTANT

You can use the default SSL configuration for simple demonstrations, but it is *not* suitable for real deployments. For advice on how to configure a real deployment, see the section called “Tightening up security”.

Procedure

To enable SSL/TLS security on the connection to the Apache Directory Server:

1. Obtain a copy of the server's self-signed certificate.

   a. Using a Web browser , navigate to the following URL:

   -

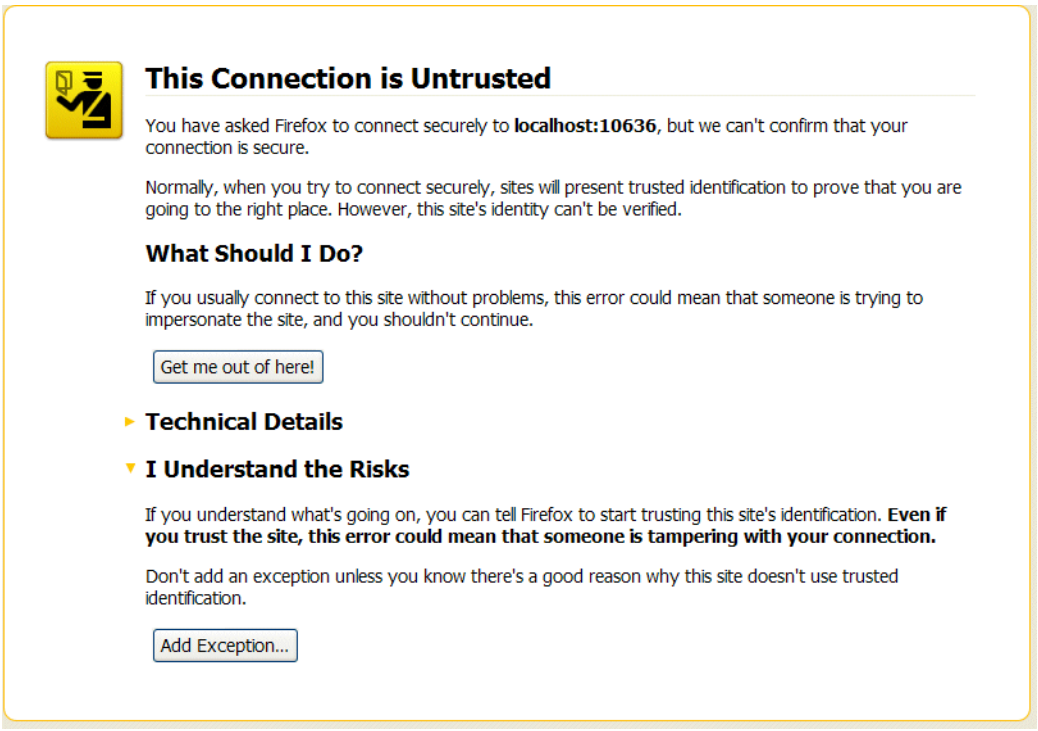
```
https://localhost:10636
```

IMPORTANT

Remember to specify the scheme as **`https`**, not just **`http`**.

The Web browser now signals an error, because the certificate it receives from the server is untrusted. In the case of Firefox, you will see the following error in the browser window:

Figure 8.6. Obtaining the Certificate



b. Click **`I Understand the Risks`**.

c. Click **`Add Exception`**.

   The **`Add Security Exception`** dialog opens.

d. In the **`Add Security Exception`** dialog, click **`Get Certificate`**.

e. Click **`View`**.

   The **`Certificate Viewer`** dialog opens.

f. In the **`Certificate Viewer`** dialog, select the **`Details`** tab.

g. Click **`Export`**.

   The **`Save Certificate To File`** dialog opens.

h. In the **`Save Certificate To File`** dialog, use the drop-down list to set the **`Save as type`** to X.509 Certificate (DER).

i. Save the certificate, **`ApacheDS.der`**, to a convenient location on the filesystem.

2. Convert the DER format certificate into a keystore.

   a. From a command prompt, change directory to the directory where you have stored the **ApacheDS.der** file.

   b. Enter the following **keytool** command:

```
keytool -import -file ApacheDS.der -alias server -keystore
truststore.ks -storepass secret
```

3. Copy the newly created keystore file, **truststore.ks**, into the Red Hat JBoss A-MQ **etc/** directory.

4. Open the **ldap-module.xml** file you created in Section 8.4, “Enable LDAP Authentication in the OSGi Container” in a text editor.

5. Edit the connection.url to use ldaps://localhost:10636.

6. Add the highlighted lines in Example 8.3, “LDAP Configuration for Using SSL/TLS”.

Example 8.3. LDAP Configuration for Using SSL/TLS

```
<?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">

  <!-- Example configuration for using LDAP based authentication.
      This example uses an JAAS LoginModule from Karaf.
      It supports authentication of users and also supports
      retrieving user roles for authorization.

       Note, this config overwrite the default karaf domain
    that is defined inside some JAR file
       by using a rank > 0 attribute.
  -->
  <jaas:config name="karaf" rank="1">
    <jaas:module
classname="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule"
flags="required">
      initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
      connection.username=uid=admin,ou=system
      connection.password=secret
      connection.protocol=
      connection.url = ldaps://localhost:10636
      user.base.dn = ou=users,ou=system
      user.filter = (uid=%u)
      user.search.subtree = true
      role.base.dn = ou=users,ou=system
      role.filter = (uid=%u)
      role.name.attribute = ou
      role.search.subtree = true
      authentication = simple
      ssl.protocol=TLSv1
```

```
        ssl.truststore=truststore
        ssl.algorithm=PKIX
      </jaas:module>
    </jaas:config>

    <jaas:keystore name="truststore"
          path="file:///InstallDir/etc/truststore.ks"
keystorePassword="secret" />
</blueprint>
```

7. Copy the **ldap-module.xml** file into the Red Hat JBoss A-MQ **deploy/** directory.

   The LDAP module is automatically activated.

8. Test the new LDAP realm by connecting to the running container using the JBoss A-MQ **client** utility.

   a. Open a new command prompt.

   b. Change to the JBoss A-MQ install directory.

   c. Enter the following command to log on to the running container instance using the identity **jdoe**:

   ```
   client -u jdoe -p secret
   ```

   You should successfully log into the container's remote console because **jdoe** does have the **admin** role.

Tightening up security

The SSL set-up described here is suitable *only* as a proof-of-concept demonstration. For a real deployment, you must make the following changes to tighten up security:

- Delete all entries from the Red Hat JBoss A-MQ's **etc/users.properties** file.

  If the **ldap-module.xml** bundle fails to start up properly, JAAS authentication reverts to the built-in file-based **karaf** realm, which takes its user data from the **users.properties** file.

- Disable the insecure LDAP endpoint on the Apache Directory Server.

- Create and deploy a properly signed X.509 certificate on the Apache Directory Server.

  See Appendix A, *Managing Certificates*.

- Make sure that the LDAP server is configured to use the TLSv1 protocol (POODLE vulnerability). Do *not* enable the SSLv3 protocol. For more information, see Poodle vulnerability (CVE-2014-3566).

Apache Directory Server Reference

For more details of how to configure SSL/TLS security on the Apache Directory Server, see How to enable SSL.

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 JBoss A-MQ 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 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.2.1. 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.2.2. 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. OpenSSL is derived from SSLeay, an implementation of SSL developed by Eric Young (`eay@cryptsoft.com`). 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 JBoss A-MQ 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.

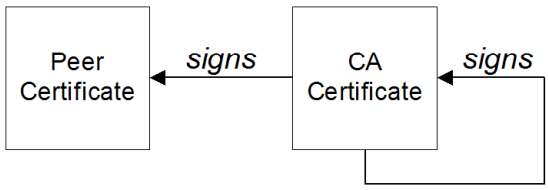
A.3. CERTIFICATE CHAINING

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

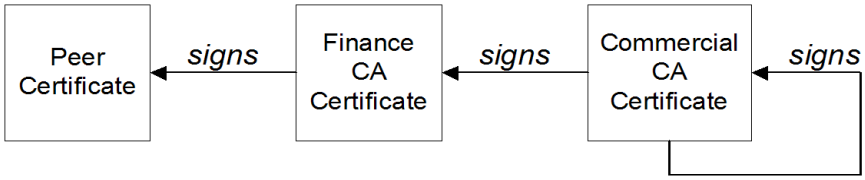
The purpose of a certificate chain is to establish a chain of trust from a peer certificate to a trusted CA certificate. The CA vouches for the identity in the peer certificate by signing it. If the CA is one that you trust (indicated by the presence of a copy of the CA certificate in your root certificate 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 subectAltName

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=Progress,
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 ???.

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
fusesource.com
```

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.cfg** configuration file to specify the value of the **subjectAltName** extension, as follows:

```
subjectAltName=DNS:www.redhat.com,DNS:fusesource.com
```

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:*.fusesource.com
```

This certificate identity matches any three-component host name in the domain fusesource.com.

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 ***fusesource.com**, your certificate could be used on *any* domain that ends in the letters **fusesource**.

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
```

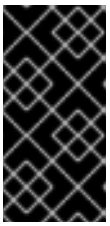
2. Using a text editor, create the file, ***X509CA/openssl.cfg***, and add the following contents to this file:

Example A.1. OpenSSL Configuration

```
#
# SSLeay 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.

3. Initialize the **`demoCA/serial`** file, which must have the initial contents **`01`** (zero one). Enter the following command:

```
echo 01 > demoCA/serial
```

4. Initialize the **`demoCA/index.txt`**, which *must* initially be completely empty. Enter the following command:

```
touch demoCA/index.txt
```

5. 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 of 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

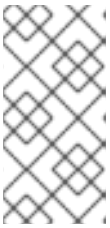
To generate and sign a new certificate, using your own private CA, perform the following steps:

1. Generate a certificate and private key pair using the **`keytool -genkeypair`** command, as follows:

   ```
   keytool -genkeypair -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).

2. Create a certificate signing request using the **`keystore -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 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 do not require 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 X.208.
- BER is defined in X.209.

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

R

S

T

U

X