Red Hat Fuse 7.4

Apache Karaf Security Guide

Security for the Apache Karaf container

Last Updated: 2020-03-27

Red Hat Fuse 7.4 Apache Karaf Security Guide

Security for the Apache Karaf container

Legal Notice

Copyright © 2020 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

Abstract

This guide describes how to secure the Red Hat Fuse container, the web console, message brokers, routing and integration components, web and RESTful services, and it provides a tutorial on LDAP authentication.

Table of Contents

CHAPTER 1. SECURITY ARCHITECTURE 8
1.1. OSGI CONTAINER SECURITY 8
Overview 8
JAAS realms 8
karaf realm 9
Console port 9
JMX port 9
Application bundles and JAAS security 9
1.2. APACHE CAMEL SECURITY 9
Overview 9
Alternatives for Apache Camel security 10
Endpoint security 10
Payload security 11
XMLSecurity data format 11
Crypto data format 11

CHAPTER 2. SECURING THE APACHE KARAF CONTAINER 12
2.1. JAAS AUTHENTICATION 12
2.1.1. Default JAAS Realm 12
Default JAAS realm 12
How to integrate an application with JAAS 12
Default JAAS login modules 12
2.1.1.1. Authentication audit logging modules 13
Configuring users in the properties login module 13
Configuring user groups in the properties login module 13
Configuring the public key login module 14
Configuring user groups in the public key login module 14
Encrypting the stored passwords 15
Overriding the default realm 15
2.1.2. Defining JAAS Realms 15
Namespace 15
Configuring a JAAS realm 16
Converting standard JAAS login properties to XML 17
Example 18
2.1.3. JAAS Properties Login Module 19
Supported credentials 19
Implementation classes 19
Options 19
Format of the user properties file 19
Sample Blueprint configuration 20
2.1.4. JAAS OSGi Config Login Module 20
Overview 20
Supported credentials 21
Implementation classes 21
Options 21
Location of the configuration file 21
Format of the configuration file 21
Sample Blueprint configuration 21
2.1.5. JAAS Public Key Login Module 22
Supported credentials 22
Implementation classes 22

Options 22
Format of the keys properties file 23
Sample Blueprint configuration 23
2.1.6. JAAS JDBC Login Module 24
Overview 24
Supported credentials 24
Implementation classes 24
Options 24
Example of setting up a JDBC login module 25
Create the database tables 25
Create the data source 26
Specify the data source as an OSGi service 26
2.1.7. JAAS LDAP Login Module 27
Overview 27
Supported credentials 28
Implementation classes 28
Options 28
Sample configuration for Apache DS 31
Filter settings for different directory servers 32
2.1.8. JAAS Log Audit Login Module 33
Appender configuration 33
Logger configuration 34
2.1.8.1. Enabling Authentication Audit Logging 34
Apache Karaf shell commands for updating configuration files 35
2.1.9. JAAS File Audit Login Module 35
2.1.10. Encrypting Stored Passwords 36
Options 36
Encryption services 37
Basic encryption service 37
Jasypt encryption 38
Example of a login module with Jasypt encryption 38
2.2. ROLE-BASED ACCESS CONTROL 40
2.2.1. Overview of Role-Based Access Control 40
Mechanisms 41
Types of protection 41
Adding roles to users 42
Standard roles 42
ACL files 42
Customizing role-based access control 43
Additional properties for controlling access 43
2.2.2. Customizing the JMX ACLs 43
Architecture 43
How it works 44
Location of JMX ACL files 44
Mapping MBeans to ACL file names 45
ACL file format 45
ACL file hierarchy 45
Root ACL definitions 46
Package ACL definitions 46
ACL for custom MBeans 46
Dynamic configuration at run time 46
2.2.3. Customizing the Command Console ACLs 47
Architecture 47

How it works 47
Configuring default security roles 48
Location of command console ACL files 48
Mapping command scopes to ACL file names 48
ACL file format 48
Dynamic configuration at run time 49
2.2.4. Defining ACLs for OSGi Services 49
ACL file format 49
How to define an ACL for a custom OSGi service 50
How to invoke an OSGi service secured with RBAC 51
How to discover the roles required by an OSGi service 52
2.3. HOW TO USE ENCRYPTED PROPERTY PLACEHOLDERS 52
2.3.1. About the master password for encrypting values 52
2.3.2. Using encrypted property placeholders 53
2.3.3. Invoking the jasypt:digest command 57
2.3.4. Invoking the jasypt:decrypt command 59
2.4. ENABLING REMOTE JMX SSL 61
Overview 61
Prerequisites 62
Create the jbossweb.keystore file 62
Create and deploy the keystore.xml file 64
Add the required properties to org.apache.karaf.management.cfg 64
Restart the Karaf container 64
Testing the Secure JMX connection 64
2.5. USING AN ELYTRON CREDENTIAL STORE 65
2.5.1. Putting a credential store into use 66
2.5.2. Behavior when system properties hold credential store configuration 67
2.5.3. Description of credential store system properties and environment variables 69
2.5.4. credential-store:create command reference 70
2.5.5. credential-store:store command reference 72
2.5.6. credential-store:list command reference 73
2.5.7. credential-store:remove command reference 73
2.5.8. Example of Configuration Admin properties enabling credential store use 74

CHAPTER 3. SECURING THE UNDERTOW HTTP SERVER 76
3.1. UNDERTOW SERVER 76
3.2. CREATE X.509 CERTIFICATE AND PRIVATE KEY 76
3.3. ENABLING SSL/TLS FOR UNDERTOW IN AN APACHE KARAF CONTAINER 76
3.4. CUSTOMIZING ALLOWED TLS PROTOCOLS AND CIPHER SUITES 78
3.5. CONNECT TO THE SECURE CONSOLE 78

CHAPTER 4. SECURING THE CAMEL ACTIVEMQ COMPONENT 79
4.1. SECURE ACTIVEMQ CONNECTION FACTORY 79
Overview 79
Programming the security properties 79
Defining a secure connection factory 79
4.2. EXAMPLE CAMEL ACTIVEMQ COMPONENT CONFIGURATION 80
Overview 80
Prerequisites 80
Sample Camel ActiveMQ component 80
Sample Camel route 81

CHAPTER 5. SECURING THE CAMEL CXF COMPONENT 82
5.1. THE CAMEL CXF PROXY DEMONSTRATION 82

Overview 82
Modifications 82
Obtaining the demonstration code 83
Obtaining the sample certificates 83
Physical part of the WSDL contract 83
WSDL addressing details 84
5.2. SECURING THE WEB SERVICES PROXY 84
Overview 84
Implicit configuration 85
Steps to add SSL/TLS security to the Jetty container 85
Add certificates to the bundle resources 86
Modify POM to switch off resource filtering 86
Instantiate the CXF Bus 86
Add the httpj:engine-factory element to Spring 87
Define the cxfcore:, sec: and httpj: prefixes 88
Modify proxy address URL to use HTTPS 88
5.3. DEPLOYING THE APACHE CAMEL ROUTE 89
Overview 89
Prerequisites 89
Steps to deploy the Camel route 89
Build the demonstration 89
Start the OSGi container 90
Install the required features 90
Deploy the bundle 90
Check the console output 90
5.4. SECURING THE WEB SERVICES CLIENT 90
Overview 91
Implicit configuration 91
Certificates needed on the client side 91
Loading Spring definitions into the client 92
Creating the client proxy 92
Steps to add SSL/TLS security to the client 93
Create the Java client as a test case 93
Add the http:conduit element to Spring configuration 95
Run the client 96

CHAPTER 6. SECURING THE FUSE CONSOLE **98**

CHAPTER 7. INTEGRATION WITH RED HAT SINGLE SIGN-ON **99**
7.1. ADAPTER FOR SPRING BOOT CONTAINER 99
7.2. ADAPTER FOR APACHE KARAF CONTAINER 99
7.3. ADAPTER FOR JBOSS EAP CONTAINER 99

CHAPTER 8. LDAP AUTHENTICATION TUTORIAL **101**
8.1. TUTORIAL OVERVIEW 101
Goals 101
8.2. SET-UP A DIRECTORY SERVER AND CONSOLE 101
Prerequisites 101
Install 389 Directory Server 101
Install 389 Management Console 102
Connect the console to the server 102
8.3. ADD USER ENTRIES TO THE DIRECTORY SERVER 104
Alternative to adding user entries 104
Goals 104

Adding user entries 104
Adding groups for the roles 106
8.4. ENABLE LDAP AUTHENTICATION IN THE OSGI CONTAINER 108
References 108
Procedure for standalone OSGi container 108
Test the LDAP authentication 110
Troubleshooting 111

APPENDIX A. MANAGING CERTIFICATES **112**
A.1. WHAT IS AN X.509 CERTIFICATE? 112
Role of certificates 112
Integrity of the public key 112
Digital signatures 112
Contents of an X.509 certificate 112
Distinguished names 113
A.2. CERTIFICATION AUTHORITIES 113
A.2.1. Introduction to Certificate Authorities 113
A.2.2. Commercial Certification Authorities 113
Signing certificates 113
Advantages of commercial CAs 113
Criteria for choosing a CA 113
A.2.3. Private Certification Authorities 114
Choosing a CA software package 114
OpenSSL software package 114
Setting up a private CA using OpenSSL 114
Choosing a host for a private certification authority 114
Security precautions 114
A.3. CERTIFICATE CHAINING 114
Certificate chain 114
Self-signed certificate 115
Chain of trust 115
Certificates signed by multiple CAs 115
Trusted CAs 115
A.4. SPECIAL REQUIREMENTS ON HTTPS CERTIFICATES 115
Overview 115
HTTPS URL integrity check 116
Reference 116
How to specify the certificate identity 116
Using commonName 116
Using subjectAltName (multi-homed hosts) 116
A.5. CREATING YOUR OWN CERTIFICATES 117
A.5.1. Install the OpenSSL Utilities 118
Installing OpenSSL on RHEL and Fedora platforms 118
Source code distribution 118
A.5.2. Set Up a Private Certificate Authority 118
Overview 118
Steps to set up a private Certificate Authority 118
A.5.3. Create a CA Trust Store File 121
Overview 121
Steps to create a CA trust store 121
A.5.4. Generate and Sign a New Certificate 121
Overview 121
Steps to generate and sign a new certificate 122

APPENDIX B. ASN.1 AND DISTINGUISHED NAMES **124**
B.1. ASN.1 124
Overview 124
BER 124
DER 124
References 124
B.2. DISTINGUISHED NAMES 124
Overview 124
String representation of DN 125
DN string example 125
Structure of a DN string 125
OID 125
Attribute types 125
AVA 126
RDN 126

CHAPTER 1. SECURITY ARCHITECTURE

Abstract

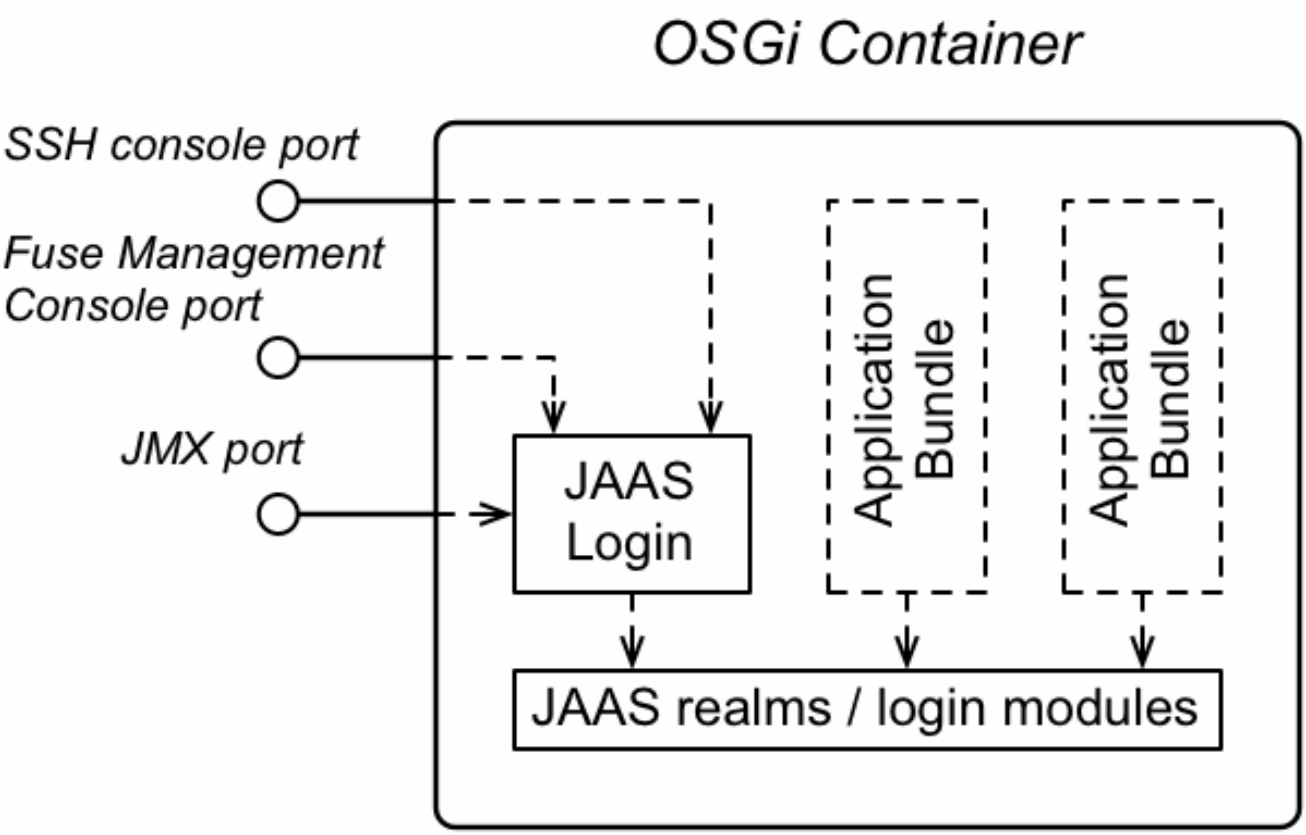
In the OSGi container, it is possible to deploy applications supporting a variety of security features. Currently, only the Java Authentication and Authorization Service (JAAS) is based on a common, container-wide infrastructure. Other security features are provided separately by the individual products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

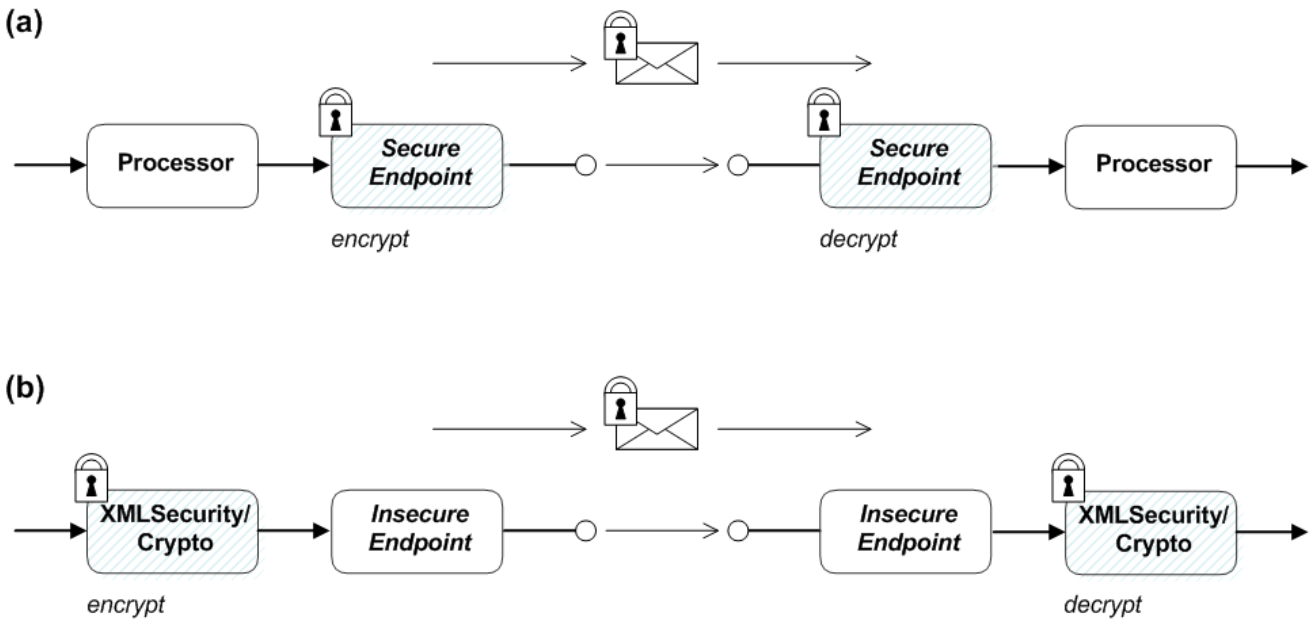
It is essential, however, that the JAAS realms are defined using the OSGi container's own login configuration mechanism—by default, Java provides a simple file-based login configuration implementation, but you **cannot** use this implementation in the context of the OSGi container.

1.2. APACHE CAMEL SECURITY

Overview

Figure 1.2, "Apache Camel Security Architecture" shows an overview of the basic options for securely routing messages between Apache Camel endpoints.

Figure 1.2. Apache Camel Security Architecture



Alternatives for Apache Camel security

As shown in Figure 1.2, "Apache Camel Security Architecture", you have the following options for securing messages:

- **Endpoint security**—part (a) shows a message sent between two routes with secure endpoints. The producer endpoint on the left opens a secure connection (typically using SSL/TLS) to the consumer endpoint on the right. Both of the endpoints support security in this scenario.
  With endpoint security, it is typically possible to perform some form of peer authentication (and sometimes authorization).

- **Payload security**—part (b) shows a message sent between two routes where the endpoints are both **insecure**. To protect the message from unauthorized snooping in this case, use a *payload processor* that encrypts the message before sending and decrypts the message after it is received.
  A limitation of payload security is that it does **not** provide any kind of authentication or authorization mechanisms.

Endpoint security

There are several Camel components that support security features. It is important to note, however, that these security features are implemented by the individual components, **not** by the Camel core. Hence, the kinds of security feature that are supported, and the details of their implementation, vary from component to component. Some of the Camel components that currently support security are, as follows:

- JMS and ActiveMQ—SSL/TLS security and JAAS security for client-to-broker and broker-to-broker communication.

- Jetty—HTTP Basic Authentication and SSL/TLS security.

- CXF—SSL/TLS security and WS-Security.

- Crypto—creates and verifies digital signatures in order to guarantee message integrity.

- Netty—SSL/TLS security.

- MINA–SSL/TLS security.
- Cometd–SSL/TLS security.
- glogin and gauth–authorization in the context of Google applications.

Payload security

Apache Camel provides the following payload security implementations, where the encryption and decryption steps are exposed as data formats on the **marshal()** and **unmarshal()** operations

- the section called "XMLSecurity data format" .
- the section called "Crypto data format" .

XMLSecurity data format

The XMLSecurity data format is specifically designed to encrypt XML payloads. When using this data format, you can specify which XML element to encrypt. The default behavior is to encrypt **all** XML elements. This feature uses a symmetric encryption algorithm.

For more details, see http://camel.apache.org/xmlsecurity-dataformat.html.

Crypto data format

The crypto data format is a general purpose encryption feature that can encrypt any kind of payload. It is based on the Java Cryptographic Extension and implements only symmetric (shared-key) encryption and decryption.

For more details, see http://camel.apache.org/crypto.html.

CHAPTER 2. SECURING THE APACHE KARAF CONTAINER

Abstract

The Apache Karaf 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

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

Default JAAS realm

The Karaf 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

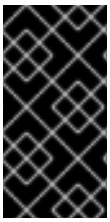
When you start the Karaf container for the first time, it is configured to use the **karaf** default realm. In this default configuration, the **karaf** realm deploys five JAAS login modules, which are enabled simultaneously. To see the deployed login modules, enter the **jaas:realms** console command, as follows:

```
Index │ Realm Name │ Login Module Class Name
──────┼────────────┼────────────────────────────────────────────────────────────
──────────────────────────────────────────────────────────────────────────────
1     │ karaf      │ org.apache.karaf.jaas.modules.properties.PropertiesLoginModule
2     │ karaf      │ org.apache.karaf.jaas.modules.publickey.PublickeyLoginModule
3     │ karaf      │ org.apache.karaf.jaas.modules.audit.FileAuditLoginModule
4     │ karaf      │ org.apache.karaf.jaas.modules.audit.LogAuditLoginModule
5     │ karaf      │ org.apache.karaf.jaas.modules.audit.EventAdminAuditLoginModule
```

Whenever a user attempts to log in, authentication proceeds through the five modules in list order. A flag value for each module specifies whether the module must complete successfully for authentication to succeed. Flag values also specify whether the authentication process stops after a module completes, or whether it proceeds to the next module.

The **Optional** flag is set for all five authentication modules. The **Optional** flag setting causes

authentication process to always pass from one module to the next, regardless of whether the current module completes successfully. The flag values in the Karaf JAAS realm are hard-coded, and cannot be changed. For more information about flags, see Table 2.1, "Flags for Defining a JAAS Module" .



IMPORTANT

In a Karaf 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.

2.1.1.1. Authentication audit logging modules

Within the list of default modules in a Karaf container, only the first two modules are used to verify user identity. The remaining modules are used to log the audit trail of successful and failed login attempts. The default realm includes the following audit logging modules:

org.apache.karaf.jaas.modules.audit.LogAuditLoginModule

This module records information about authentication attempts by using the loggers that are configured for the Pax logging infrastructure in the file **etc/org.ops4j.pax.logging.cfg**. For more information, see JAAS Log Audit Login Module.

org.apache.karaf.jaas.modules.audit.FileAuditLoginModule

This module records information about authentication attempts directly to a file that you specify. It does not use the logging infrastructure. For more information, see JAAS File Audit Login Module .

org.apache.karaf.jaas.modules.audit.EventAdminAuditLoginModule

This module tracks authentication attempts using the OSGi Event Admin service.

Configuring users in the properties login module

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

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

For example, to create the **jdoe** user with password, **topsecret**, and role, **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 user groups in the properties login module

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

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

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

```
_g_\:admingroup=group,admin
```

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

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

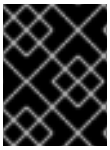
Configuring the public key login module

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

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

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

```
jdoe=AAAAB3NzaC1kc3MAAACBAP1/U4EddRIpUt9KnC7s5Of2EbdSPO9EAMMeP4C2USZpRV1AIlH
7WT2NWPq/xfW6MPbLm1Vs14E7gB00b/JmYLdrmVClpJ+f6AR7ECLCT7up1/63xhv4O1fnfqimFQ8E+4
P208UewwI1VBNaFpEy9nXzrith1yrv8iIDGZ3RSAHHAAAAFQCXYFCPFSMLzLKSuYKi64QL8Fgc9QA
AAnEA9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0HgmdRWVeOutRZT+ZxBxCBgLRJFnEj6E
woFhO3zwkyjMim4TwWeotifI0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhRkImog9/hWuWfB
pKLZl6Ae1UlZAFMO/7PSSoAAACBAKKSU2PFl/qOLxIwmBZPPIcJshVe7bVUpFvyl3BbJDow8rXfskl8w
O63OzP/qLmcJM0+JbcRU/53Jj7uyk31drV2qxhIOsLDC9dGCWj47Y7TyhPdXh/0dthTRBy6bqGtRPxGa
7gJov1xm/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.

Configuring user groups in the public key login module

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

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

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

```
_g_\:admingroup=group,admin
```

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

```
jdoe=AAAAB3NzaC1kc3MAAACBAP1/U4EddRIpUt9KnC7s5Of2EbdSPO9EAMMeP4C2USZpRV1AIlH
7WT2NWPq/xfW6MPbLm1Vs14E7gB00b/JmYLdrmVClpJ+f6AR7ECLCT7up1/63xhv4O1fnfqimFQ8E+4
P208UewwI1VBNaFpEy9nXzrith1yrv8iIDGZ3RSAHHAAAAFQCXYFCPFSMLzLKSuYKi64QL8Fgc9QA
AAnEA9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0HgmdRWVeOutRZT+ZxBxCBgLRJFnEj6E
woFhO3zwkyjMim4TwWeotifI0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhRkImog9/hWuWfB
```

```
pKLZl6Ae1UlZAFMO/7PSSoAAACBAKKSU2PFl/qOLxIwmBZPPIcJshVe7bVUpFvyl3BbJDow8rXfskl8w
O63OzP/qLmcJM0+JbcRU/53Jj7uyk31drV2qxhIOsLDC9dGCWj47Y7TyhPdXh/0dthTRBy6bqGtRPxGa
7gJov1xm/UuYYXPIUR/3x9MAZvZ5xvE0kYXO+rx,_g_:admingroup
```

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 Karaf container. They have no effect on a custom realm.

For more details about password encryption, see Section 2.1.10, “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 Fuse 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

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 need to include the line shown in Example 2.1, “JAAS Blueprint Namespace”.

Example 2.1. JAAS Blueprint Namespace

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

Configuring a JAAS realm

The syntax for the **jaas:config** element is shown in Example 2.2, "Defining a JAAS Realm in Blueprint XML".

Example 2.2. Defining a JAAS Realm in Blueprint XML

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

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

</blueprint>
```

The elements are used as follows:

jaas:config

Defines the JAAS realm. It has the following attributes:

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

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

Example 2.3. Standard JAAS Properties

```
PropertiesLogin {
   org.apache.activemq.jaas.PropertiesLoginModule required
      org.apache.activemq.jaas.properties.user="users.properties"
      org.apache.activemq.jaas.properties.group="groups.properties";
};
```

The equivalent JAAS realm definition, using the **jaas:config** element in a blueprint file, is shown in Example 2.4, “Blueprint JAAS Properties”.

Example 2.4. Blueprint JAAS Properties

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

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

</blueprint>
```

IMPORTANT

Do not use double quotes for JAAS properties in the blueprint configuration.

Example

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

Example 2.5. Configuring a JAAS Realm

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

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

-

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

2.1.3. JAAS Properties Login Module

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 Karaf 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 Apache Karaf Console Reference).

Options

The JAAS properties login module supports the following options:

users

Location of the user properties file.

Format of the user properties file

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

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

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

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

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

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

# Groups
_g_\:admingroup=group,admin
_g_\:guestgroup=viewer
```

Sample Blueprint configuration

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

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

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

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

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

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

</blueprint>
```

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

2.1.4. JAAS OSGi Config Login Module

Overview

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

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

Supported credentials

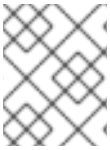
The JAAS OSGi config login module authenticates username/password credentials, returning the list of roles associated with the authenticated user.

Implementation classes

The following classes implement the JAAS OSGi config login module:

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

Implements the JAAS login module.



NOTE

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

Options

The JAAS OSGi config login module supports the following options:

pid

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

Location of the configuration file

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

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

Format of the configuration file

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

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

NOTE

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

Sample Blueprint configuration

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

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

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

</blueprint>
```

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

2.1.5. JAAS Public Key Login Module

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

Supported credentials

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

Implementation classes

The following classes implement the JAAS public key login module:

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

Implements the JAAS login module.

NOTE

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

Options

The JAAS public key login module supports the following options:

users

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

Format of the keys properties file

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

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

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

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


```
jdoe=AAAAB3NzaC1kc3MAAACBAP1/U4EddRIpUt9KnC7s5Of2EbdSPO9EAMMeP4C2USZpRV1AIlH
7WT2NWPq/xfW6MPbLm1Vs14E7gB00b/JmYLdrmVClpJ+f6AR7ECLCT7up1/63xhv4O1fnfqimFQ8E+4
P208UewwI1VBNaFpEy9nXzrith1yrv8iIDGZ3RSAHHAAAAFQCXYFCPFSMLzLKSuYKi64QL8Fgc9QA
AAnEA9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0HgmdRWVeOutRZT+ZxBxCBgLRJFnEj6E
woFhO3zwkyjMim4TwWeotifI0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhRkImog9/hWuWfB
pKLZl6Ae1UlZAFMO/7PSSoAAACBAKKSU2PFl/qOLxIwmBZPPIcJshVe7bVUpFvyl3BbJDow8rXfskl8w
O63OzP/qLmcJM0+JbcRU/53Jj7uyk31drV2qxhIOsLDC9dGCWj47Y7TyhPdXh/0dthTRBy6bqGtRPxGa
7gJov1xm/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.

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

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

Sample Blueprint configuration

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

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

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

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

```
  </jaas:config>

</blueprint>
```

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

2.1.6. JAAS JDBC Login Module

Overview

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

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

NOTE

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

Supported credentials

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

Implementation classes

The following classes implement the JAAS JDBC Login Module:

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

Implements the JAAS login module.

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

Must be exposed as an OSGi service. This service makes it possible for you to manage the user data using the **jaas:*** console commands from the Apache Karaf shell (see olink:FMQCommandRef/Consolejaas).

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

query.password

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

query.role

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

insert.user

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

insert.role

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

delete.user

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

delete.role

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

delete.roles

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

Example of setting up a JDBC login module

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

1. the section called "Create the database tables"
2. the section called "Create the data source"
3. the section called "Specify the data source as an OSGi service"

Create the database tables

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

```
CREATE TABLE users (
  username VARCHAR(255) NOT NULL,
```

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

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

Create the data source

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

```
<blueprint xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
  <bean id="mysqlDatasource"
       class="com.mysql.jdbc.jdbc2.optional.MysqlDataSource">
    <property name="serverName" value="localhost"></property>
    <property name="databaseName" value="DBName"></property>
    <property name="port" value="3306"></property>
    <property name="user" value="DBUser"></property>
    <property name="password" value="DBPassword"></property>
  </bean>

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

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

Specify the data source as an OSGi service

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

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

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

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

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

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

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

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

</blueprint>
```

NOTE

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

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

2.1.7. JAAS LDAP Login Module

Overview

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

For more details about using LDAP with Red Hat Fuse see LDAP Authentication Tutorial.

NOTE

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

Supported credentials

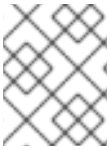
The JAAS LDAP Login Module authenticates username/password credentials, returning the list of roles associated with the authenticated user.

Implementation classes

The following classes implement the JAAS LDAP Login Module:

org.apache.karaf.jaas.modules.ldap.LDAPLoginModule

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



NOTE

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

Options

The JAAS LDAP login module supports the following options:

authentication

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

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

NOTE

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

connection.url

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

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

connection.username

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

connection.password

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

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

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

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

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

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

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

context.java.naming.referral

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

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

disableCache

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

initial.context.factory

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

role.base.dn

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

role.filter

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

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

role.mapping

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

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

Where each LDAP group, ***ldap-group***, is specified by its Common Name (CN).

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

```
role.mapping=admin=admin;devop=admin,manager;tester=viewer
```

role.name.attribute

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

role.search.subtree

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

ssl

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

ssl.provider

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

ssl.protocol

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

ssl.algorithm

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

ssl.keystore

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

ssl.keyalias

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

ssl.truststore

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

user.base.dn

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

user.filter

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

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

user.search.subtree

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

Sample configuration for Apache DS

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

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

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

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

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

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

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

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

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

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

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

<!-- How are LDAP referrals handled?

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

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

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

NOTE

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

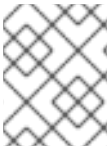
IMPORTANT

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

Filter settings for different directory servers

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

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



NOTE

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

2.1.8. JAAS Log Audit Login Module

The login module **org.apache.karaf.jaas.modules.audit.LogAuditLoginModule** provides robust logging of authentication attempts. It supports standard log management capabilities such as setting a maximum file size, log rotation, file compression, and filtering. You establish settings for these options in the logging configuration file.

By default, authentication audit logging is disabled. Enabling logging requires you to define a logging configuration and an audit configuration, and then link the two together. In the logging configuration you specify properties for a *file appender* process, and a *logger* process. The file appender publishes information about authentication events to a specified file. The logger is a mechanism that captures information about authentication events and makes it available to the appenders that you specify. You define the logging configuration in the standard Karaf Log4j logging configuration file, **etc/org.ops4j.pax.logging.cfg**.

The audit configuration enables audit logging and links to the logging infrastructure to be used. You define the audit configuration in the file **etc/org.apache.karaf.jaas.cfg**.

Appender configuration

By default, the standard Karaf Log4j configuration file (**etc/org.ops4j.pax.logging.cfg**) defines an audit logging appender with the name **AuditRollingFile**.

The following excerpt from a sample configuration file shows the properties of an appender that writes to an audit log file at **${karaf.data}/security/audit.log**:

-

```
# Audit file appender
log4j2.appender.audit.type = RollingRandomAccessFile
log4j2.appender.audit.name = AuditRollingFile
log4j2.appender.audit.fileName = ${karaf.data}/security/audit.log
log4j2.appender.audit.filePattern = ${karaf.data}/security/audit.log.%i
log4j2.appender.audit.append = true
log4j2.appender.audit.layout.type = PatternLayout
log4j2.appender.audit.layout.pattern = ${log4j2.pattern}
log4j2.appender.audit.policies.type = Policies
log4j2.appender.audit.policies.size.type = SizeBasedTriggeringPolicy
log4j2.appender.audit.policies.size.size = 8MB
```

To use the appender, you must configure a logger that provides the information for the appender to publish to a log file.

Logger configuration

By default, the Karaf Log4j configuration file **etc/org.ops4j.pax.logging.cfg** establishes an audit logger with the name **org.apache.karaf.jaas.modules.audit**. In the following excerpt from a sample configuration file, the default logger is configured to provide information about authentication events to an appender with the name **AuditRollingFile**:

```
log4j2.logger.audit.name = org.apache.karaf.jaas.modules.audit
log4j2.logger.audit.level = INFO
log4j2.logger.audit.additivity = false
log4j2.logger.audit.appenderRef.AuditRollingFile.ref = AuditRollingFile
```

The value of **log4j2.logger.audit.appenderRef.AuditRollingFile.ref** must match the value of **log4j2.appender.audit.name** in the **Audit file appender** section of **etc/org.ops4j.pax.logging.cfg**.

2.1.8.1. Enabling Authentication Audit Logging

After you establish the logging configuration, you can turn on audit logging and connect the logging configuration to the audit configuration.

To enable audit logging, insert the following lines in **etc/org.apache.karaf.jaas.cfg**:

```
audit.log.enabled = true
audit.log.logger = <logger.name>
audit.log.level = <level>
```

The **<logger.name>** represents in dot-separated format any standard logger (category) name that is established by the Apache Log4J and Log4J2 libraries, for example, **org.jboss.fuse.audit** or **com.example.audit**. The **<level>**` represents a log level setting, such as **WARN**, **INFO**, **TRACE**, or **DEBUG**.

For example, in the following excerpt from a sample audit configuration file, the audit log is enabled and it is configured to use the audit logger with the name **org.apache.karaf.jaas.modules.audit**:

```
audit.log.enabled = true
audit.log.logger = org.apache.karaf.jaas.modules.audit
audit.log.level = INFO
```

The value for **audit.log.logger** must match the value of **log4j2.logger.audit.name** in the Karaf Log4j configuration file (**etc/org.ops4j.pax.logging.cfg**).

After you update a file, the Apache Felix File Install bundle detects the change and updates the configuration in the Apache Felix Configuration Administration Service (**Config Admin**). The settings from the Config Admin are then passed to the logging infrastructure.

Apache Karaf shell commands for updating configuration files

You can edit configuration files in **<FUSE_HOME>/etc** directly, or you can run Apache Karaf **config:*** commands to update the Config Admin.

When you use the **config*** commands to update the configuration, the Apache Felix File Install bundle is notified about the changes and automatically updates the relevant **etc/*.cfg** files.

Example: Using a config command to list the properties for the JAAS realm

To list the properties in the JAAS realm, from a shell prompt, type the following command:

config:property-list --pid org.apache.karaf.jaas

The command returns the current properties for the realm, for example:

```
audit.log.enabled = true
audit.log.level = INFO
audit.log.logger = org.apache.karaf.jaas.modules.audit
encryption.algorithm = MD5
encryption.enabled = false
encryption.encoding = hexadecimal
encryption.name =
encryption.prefix = {CRYPT}
encryption.suffix = {CRYPT}
```

Example: Using a config command to change the audit log level

To change the audit log level for the realm to **DEBUG**, from a shell prompt, type the following command: **config:property-set --pid org.apache.karaf.jaas audit.log.level DEBUG**

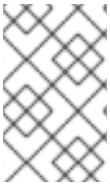
To verify that the change is effective, list the properties again to check the value for **audit.log.level**.

2.1.9. JAAS File Audit Login Module

The authentication module **org.apache.karaf.jaas.modules.audit.FileAuditLoginModule** provides basic logging of authentication attempts. The File Audit Login module writes directly to a specified file. Configuration is simple, because it does not rely on the Pax logging infrastructure. But unlike the Log Audit Login Module, it does not support log management features, such as pattern filtering, log file rotation, and so forth.

To enable audit logging with the **FileAuditLoginModule**, insert the following lines in **etc/org.apache.karaf.jaas.cfg**:

```
audit.file.enabled = true
audit.file.file = ${karaf.data}/security/audit.log
```



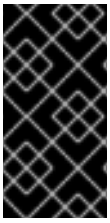
NOTE

Typically, you would not configure audit logging through both the **File Audit Login Module** and the **Log Audit Login Module**. If you enable logging through both modules, you can avoid loss of data by configuring each module to use a unique target log file.

2.1.10. Encrypting Stored Passwords

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



IMPORTANT

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

Options

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

encryption.enabled

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

encryption.name

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

encryption.prefix

Prefix for encrypted passwords.

encryption.suffix

Suffix for encrypted passwords.

encryption.algorithm

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

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

encryption.encoding

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

encryption.providerName (Jasypt only)

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

encryption.providerClassName (Jasypt only)

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

encryption.iterations (Jasypt only)

Number of times to apply the hash function recursively.

encryption.saltSizeBytes (Jasypt only)

Size of the salt used to compute the digest.

encryption.saltGeneratorClassName (Jasypt only)

Class name of the salt generator.

role.policy

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

role.discriminator

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

Encryption services

There are two encryption services provided by Fuse:

- **encryption.name = basic**, described in the section called "Basic encryption service" ,
- **encryption.name = jasypt**, described in the section called "Jasypt encryption" .

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

- Implement the **org.apache.karaf.jaas.modules.EncryptionService** interface, and
- Expose your implementation as OSGI service.

The following listing shows how to expose a custom encryption service to the OSGI container:

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

  <service interface="org.apache.karaf.jaas.modules.EncryptionService">
    <service-properties>
      <entry key="name" value="jasypt" />
    </service-properties>
    <bean class="org.apache.karaf.jaas.jasypt.impl.JasyptEncryptionService"/>
  </service>
  ...
</blueprint>
```

Basic encryption service

The basic encryption service is installed in the Karaf 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 is normally installed by default on Karaf. If necessary, you can install it explicitly by installing the **jasypt-encryption** feature, as follows:

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

This command installs the requisite Jasypt bundles and exports Jasypt encryption as an OSGi service, so that it is available for use by JAAS login modules.

For more information about Jasypt encryption, see the Jasypt documentation.

Example of a login module with Jasypt encryption

By default, the passwords are stored in clear form in the **etc/users.properties** file. It is possible to enable encryption by installing **jasypt-encryption** feature and modifying the **etc/org.apache.karaf.jaas.cfg** configuration file.

1. Install feature **jasypt-encryption**. This will install the **jasypt** service.

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

   Now you can use the **jaas** commands to create users.

2. Open the **$FUSE_HOME/etc/org.apache.karaf.jaas.cfg** file and modify it as follows. Set the **encryption.enabled = true**, **encryption.name = jasypt**, and in this case **encryption.algorithm = SHA-256**. There are other **encryption.algorithm** options available, you can set it as per your requirement.

   ```
   #
   # Boolean enabling / disabling encrypted passwords
   #
   encryption.enabled = true

   #
   # Encryption Service name
   #   the default one is 'basic'
   #   a more powerful one named 'jasypt' is available
   #       when installing the encryption feature
   #
   encryption.name = jasypt

   #
   # Encryption prefix
   #
   encryption.prefix = {CRYPT}

   #
   # Encryption suffix
   #
   encryption.suffix = {CRYPT}

   #
   # Set the encryption algorithm to use in Karaf JAAS login module
   # Supported encryption algorithms follow:
   ```

```
#   MD2
#   MD5
#   SHA-1
#   SHA-256
#   SHA-384
#   SHA-512
#
encryption.algorithm = SHA-256
```

3. Enter **jaas:realms** command on the Karaf console to view the deployed login modules.

```
karaf@root()> jaas:realms

Index │ Realm Name │ Login Module Class Name

──────┼────────────┼──────────────────────────────────────────────────────────────────────────────

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

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

3     │ karaf      │ org.apache.karaf.jaas.modules.audit.FileAuditLoginModule

4     │ karaf      │ org.apache.karaf.jaas.modules.audit.LogAuditLoginModule

5     │ karaf      │ org.apache.karaf.jaas.modules.audit.EventAdminAuditLoginModule
```

4. Enter the following commands to create users.

```
karaf@root()> jaas:realm-manage --index 1

karaf@root()> jaas:user-list

User Name │ Group      │ Role

──────────┼────────────┼──────────────

admin     │ admingroup │ admin

admin     │ admingroup │ manager

admin     │ admingroup │ viewer

admin     │ admingroup │ systembundles

admin     │ admingroup │ ssh

karaf@root()> jaas:useradd usertest test123

karaf@root()> jaas:group-add usertest admingroup

karaf@root()> jaas:update

karaf@root()> jaas:realm-manage --index 1
```

```
karaf@root()> jaas:user-list
User Name │ Group      │ Role
──────────┼────────────┼──────────────
admin     │ admingroup │ admin
admin     │ admingroup │ manager
admin     │ admingroup │ viewer
admin     │ admingroup │ systembundles
admin     │ admingroup │ ssh
usertest  │ admingroup │ admin
usertest  │ admingroup │ manager
usertest  │ admingroup │ viewer
usertest  │ admingroup │ systembundles
usertest  │ admingroup │ ssh
```

5. Now if you look at the **$FUSE_HOME/etc/users.properties** file you can see that the user **usertest** is added to the file.

```
admin =
{CRYPT}WXX+4PM2G7nT045ly4iS0EANsv9H/VwmStGIb9bcbGhFH5RgMuL0D3H/GVTigpga
{CRYPT},_g_:admingroup

_g_\:admingroup = group,admin,manager,viewer,systembundles,ssh

usertest =
{CRYPT}33F5E76E5FF97F3D27D790AAA1BEE36057410CCDBDBE2C792239BB2853D176
54315354BB8B608AD5{CRYPT},_g_:admingroup
```

6. You can test the newly created login in a different terminal as you have already run a **jaas:update** command.

2.2. ROLE-BASED ACCESS CONTROL

Abstract

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

2.2.1. Overview of Role-Based Access Control

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

Mechanisms

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

JMX Guard

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

OSGi Service Guard

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

Types of protection

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

Fuse Console (Hawtio)

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

JMX

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

Karaf command console

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

NOTE

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

OSGi services

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

Adding roles to users

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

```
admin = secretpass,group,admin,manager,viewer,systembundles,ssh
```

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

```
admin = secretpass, _g_:admingroup

_g_\:admingroup = group,admin,manager,viewer,systembundles,ssh
```

NOTE

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

Standard roles

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

Table 2.2. Standard Roles for Access Control

Roles	Description
viewer	Grants read-only access to the Karaf container.
manager	Grants read-write access at the appropriate level for ordinary users, who want to deploy and run applications. But blocks access to sensitive Karaf container configuration settings.
admin	Grants unrestricted access to the Karaf container.
ssh	Grants users permission to connect to the Karaf command console (through the ssh port).

ACL files

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

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

JMX ACL files.

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

Command console ACL files.

Customizing role-based access control

A complete set of JMX ACL files and command console ACL files are provided by default. You are free to customize these ACLs as required to suit the requirements of your system. Details of how to do this are given in the following sections.

Additional properties for controlling access

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

karaf.local.roles

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

hawtio.roles

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

karaf.secured.command.compulsory.roles

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

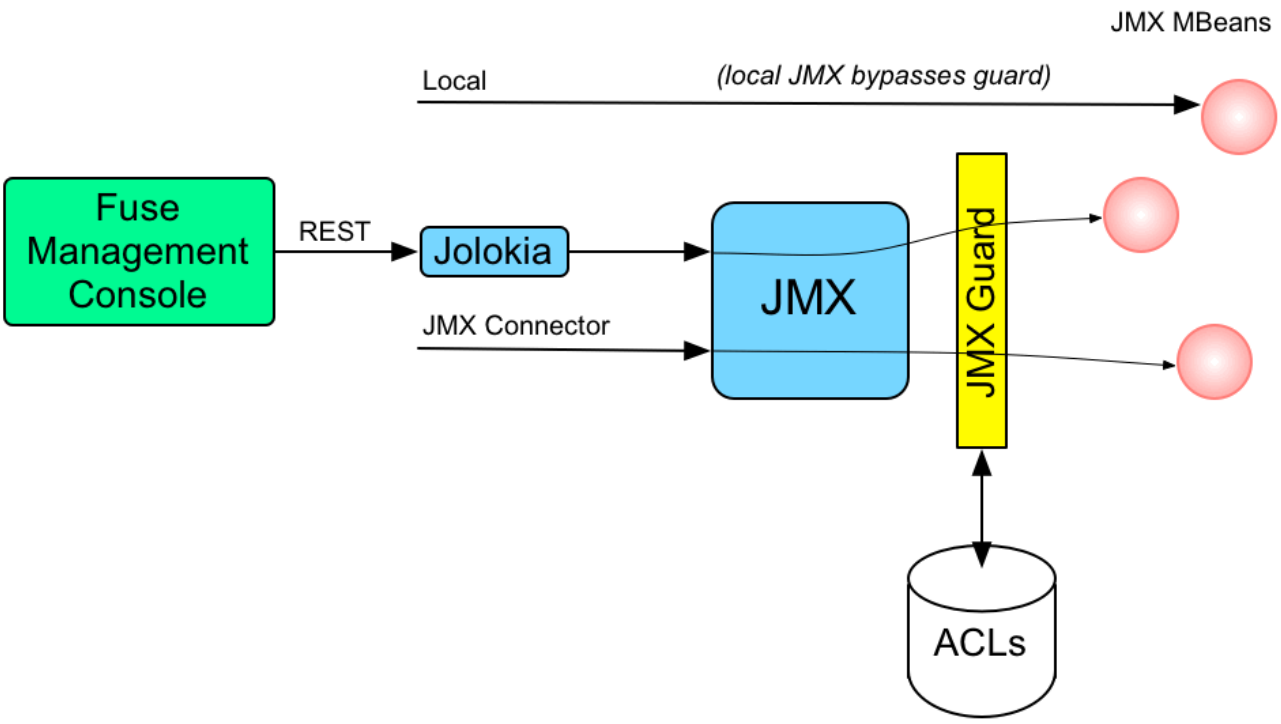
2.2.2. Customizing the JMX ACLs

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

Architecture

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

Figure 2.1. Access Control Mechanism for JMX



How it works

JMX access control works by providing remote access to JMX through a special **javax.management.MBeanServer** object. This object acts as a proxy by invoking an **org.apache.karaf.management.KarafMBeanServerGuard** object, which is referred to as JMX guard. JMX guard is available without special configuration in startup files.

JMX access control is applied as follows:

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

Location of JMX ACL files

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

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

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

Mapping MBeans to ACL file names

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

```
jmx.acl.org.apache.camel.context
```

The OSGi Config Admin service stores this PID data in the following file:

```
etc/auth/jmx.acl.org.apache.camel.context.cfg
```

ACL file format

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

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

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

```
getResourceURL = admin, manager, viewer
getLoadClassOrigin = admin, manager, viewer
```

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

```
set* = admin, manager, viewer
```

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

```
create(java.lang.String)[/jmx[.]acl.*/] = admin
create(java.lang.String)[/org[.]apache[.]karaf[.]command[.]acl.+/] = admin
create(java.lang.String)[/org[.]apache[.]karaf[.]service[.]acl.+/] = admin
create(java.lang.String) = admin, manager
```

In this case, the **manager** role generally has permission to invoke the **create** method, but only the **admin** role has permission to invoke **create** with a PID argument matching **jmx.acl.***, **org.apache.karaf.command.acl.***, or **org.apache.karaf.service.***.

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

ACL file hierarchy

Because it is often impractical to provide an ACL file for every single MBean, you have the option of specifying an ACL file at the level of a Java package, which provides default settings for **all** of the

MBeans in that package. For example, the **org.apache.cxf.Bus** MBean could be affected by ACL settings at **any** of the following PID levels:

```
jmx.acl.org.apache.cxf.Bus
jmx.acl.org.apache.cxf
jmx.acl.org.apache
jmx.acl.org
jmx.acl
```

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

Root ACL definitions

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

```
list* = admin, manager, viewer
get* = admin, manager, viewer
is* = admin, manager, viewer
set* = admin
* = admin
```

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

Package ACL definitions

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

```
is* = admin, manager, viewer
get* = admin, manager, viewer
set* = admin, manager
```

ACL for custom MBeans

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

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

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

Dynamic configuration at run time

Because the OSGi Config Admin service is dynamic, you can change ACL settings while the system is

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

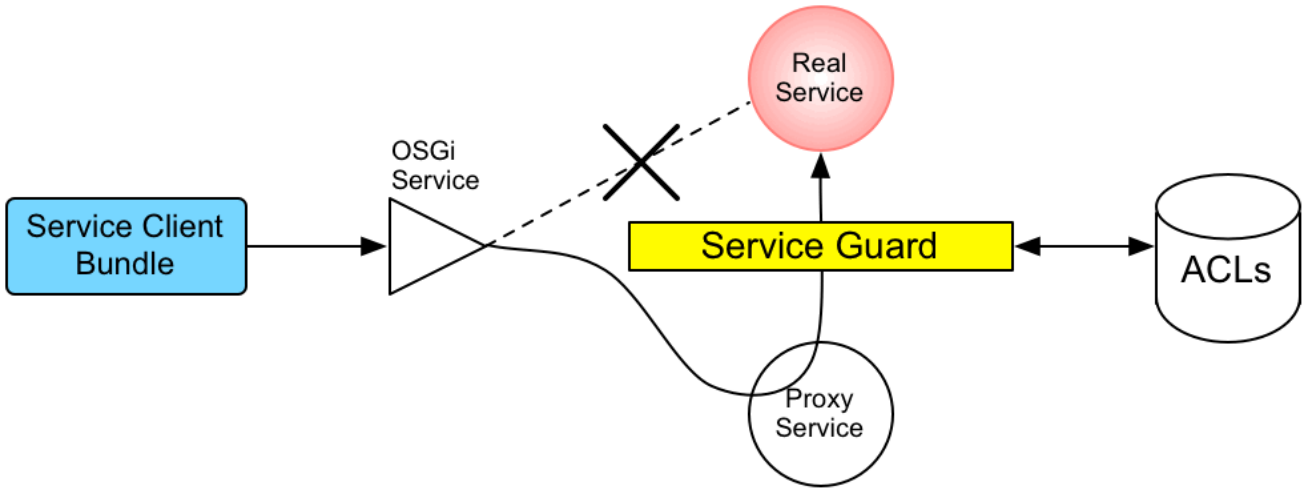
2.2.3. Customizing the Command Console ACLs

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

Architecture

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

Figure 2.2. Access Control Mechanism for OSGi Services



How it works

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

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

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

1. The invocation does **not** go directly to the requested OSGi service. Instead, the request is routed to a replacement proxy service, which has the same service properties as the original service (and some extra ones).

2. The service guard looks up the relevant ACL for the target OSGi service (where the ACLs are stored in the OSGi Config Admin service).

3. The ACL returns the list of roles that are allowed to make this particular method invocation on the service.

4. If no ACL is found for this command, the service guard defaults to the list of roles specified in the **karaf.secured.command.compulsory.roles** property in the **etc/system.properties** file.

5. The service guard checks the list of roles against the current security subject (the user that is making the method invocation), to see whether the current user has any of the required roles.

6. If no matching role is found, the method invocation is blocked and a **SecurityException** is raised.

7. Alternatively, if a matching role is found, the method invocation is delegated to the original OSGi service.

Configuring default security roles

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

Location of command console ACL files

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

Mapping command scopes to ACL file names

The command console ACL file names obey the following convention:

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

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

ACL file format

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

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

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

```
list = admin, manager, viewer
repo-list = admin, manager, viewer
info = admin, manager, viewer
version-list = admin, manager, viewer
repo-refresh = admin, manager
repo-add = admin, manager
repo-remove = admin, manager
install = admin
uninstall = admin
```

IMPORTANT

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

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

```
start[/.*[-][f].*/] = admin
start = admin, manager
stop[/.*[-][f].*/] = admin
stop = admin, manager
```

In this case, the **manager** role generally has permission to invoke the **bundle:start** and **bundle:stop** commands, but only the **admin** role has permission to invoke these commands with the force option, **-f**.

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

Dynamic configuration at run time

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

2.2.4. Defining ACLs for OSGi Services

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

ACL file format

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

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

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

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

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

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

How to define an ACL for a custom OSGi service

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

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

```
package org.example;

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

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

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

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

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

</blueprint>
```

3. To define an ACL for the the OSGi service, you must create an OSGi Config Admin PID with the prefix, **org.apache.karaf.service.acl**.
   For example, in the case of a Karaf container (where the OSGi Config Admin PIDs are stored as **.cfg** files under the **etc/auth/** directory), you can create the following ACL file for the **MyService** OSGi service:

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

NOTE

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

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

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

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

```
service.guard = (objectClass=org.example.MyService)
doit = admin, manager, viewer
```

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

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

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

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

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

How to invoke an OSGi service secured with RBAC

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

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

NOTE

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

How to discover the roles required by an OSGi service

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

2.3. HOW TO USE ENCRYPTED PROPERTY PLACEHOLDERS

When securing a Karaf container, do not use plain text passwords in configuration files. One way to avoid using plain text passwords is to use encrypted property placeholders whenever possible. See the following topics for details:

- Section 2.3.1, “About the master password for encrypting values”
- Section 2.3.2, “Using encrypted property placeholders”
- Section 2.3.3, “Invoking the **jasypt:digest** command”
- Section 2.3.4, “Invoking the **jasypt:decrypt** command”

2.3.1. About the master password for encrypting values

To use Jasypt to encrypt a value, a master password is required. It is up to you or an administrator to choose the master password. Jasypt provides several ways to set the master password.

One way is to specify the master password in plain text in a Blueprint configuration, for example:

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

  <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="myPassword" />
        </bean>
      </property>
    </enc:encryptor>
  </enc:property-placeholder>

</blueprint>
```

Instead of specifying the master password in plain text, you can do one of the following:

- Set an environment variable to your master password. In the Blueprint configuration file, specify this environment variable as the value of the **passwordEnvName** property. For example, if you set the **MASTER_PW** environment variable to your master password, then you would have this entry in your Blueprint configuration file:
  <property name="passwordEnvName" value="MASTER_PW">
- Set a Karaf system property to your master password. In the Blueprint configuration file, specify this system property as the value of the **passwordSys** property. For example, if you set the

karaf.password system property to your master password, then you would have this entry in your Blueprint configuration file:
<property name="passwordSys" value="karaf.password">

2.3.2. Using encrypted property placeholders

Use encrypted property placeholders in Blueprint configuration files when securing a Karaf container.

Prerequisites

- You know the master password for encrypting values.

Procedure

1. Plan to use the default encryption algorithm, which is **PBEWithMD5AndDES**, or choose the encryption algorithm to use as follows:

   a. Discover which algorithms are supported in your current Java environment by running the **jasypt:list-algorithms** command:

   ```
   karaf@root()> jasypt:list-algorithms
   ```

   There are no arguments or options. The output is a list of the identifiers for supported digest and Password Based Encryption (PBE) algorithms. The list includes algorithms provided by the Bouncy Castle library, which is part of Fuse 7.4. This list can be long. A short portion of it would look like this:

   ```
   karaf@root()> jasypt:list-algorithms
   DIGEST ALGORITHMS:
    - 1.0.10118.3.0.55
    - 1.2.804.2.1.1.1.1.2.2.1
      ...
    - 2.16.840.1.101.3.4.2.9
    - BLAKE2B-160
    - BLAKE2B-256
      ...
    - MD4
    - MD5
    - OID.1.0.10118.3.0.55
      ...
    - SHA3-512
    - SKEIN-1024-1024
    - SKEIN-1024-384
      ...
    - TIGER
    - WHIRLPOOL

   PBE ALGORITHMS:
    - PBEWITHHMACSHA1ANDAES_128
    - PBEWITHHMACSHA1ANDAES_256
      ...
    - PBEWITHSHA1ANDRC2_128
    - PBEWITHSHA1ANDRC2_40
   ```

```
  ...
- PBEWITHSHAANDIDEA-CBC
- PBEWITHSHAANDTWOFISH-CBC
```

b. Examine the list and find the identifier for the encryption algorithm that you want to use. You might want to consult with security experts at your site for help with choosing the algorithm.

2. To encrypt a sensitive configuration value, such as a password to be used in a configuration file, run the **jasypt:encrypt** command. The command has the following format:
jasypt:encrypt [*options*] [*input*]

   When you invoke this command without specifying any options, and you do not specify the value that you want to encrypt, the command prompts you for your master password and for the value to encrypt, and applies defaults for other options. For example:

```
karaf@root()> jasypt:encrypt
Master password: ********
Master password (repeat): ********
Data to encrypt: *****
Data to encrypt (repeat): *****
Algorithm used: PBEWithMD5AndDES
Encrypted data: oT8/LImAFQmOfXxuFGRDTAjD1I1+GxKL+TnHxFNwX4A=
```

   Invoke the **jasypt:encrypt** command for each value that you want to encrypt.

   To change the default behavior, specify one or more of the following options:

Option	Description	Example
-w or **--password-property**	Follow this option with an environment variable or a system property that is set to the value of your master password. Jasypt uses this value, in conjunction with an encryption algorithm, to derive the encryption key. If you do not specify the **-w** or the **-W** option, after you invoke the command, it prompts you to enter and confirm your master password.	**-w MASTER_PW**
-W or **--password**	Follow this option with the plain text value of your chosen master password. The plain text value of your master password appears in history. Jasypt uses this value, in conjunction with an encryption algorithm, to derive the encryption key. If you do not specify the **-w** or the **-W** option, after you invoke the command, it prompts you to enter and confirm your master password.	**-W "M@s!erP#"**

Option	Description	Example
-a or **--algorithm**	Follow this option with the identifier for the algorithm that you want the **jasypt:encrypt** command to use to derive the initial cryptographic key. The default is **PBEWithMD5AndDES**. All algorithms that are in the list that the **jasypt-list-algorithms** command outputs are supported. Auto-completion is available when specifying algorithm names on the command line.	For example: **-a PBEWITHMD5AND RC2**
-i or **--iterations**	Follow this option with an integer that indicates the number of times to iteratively create a hash of the initial key. Each iteration takes the previous hash result and hashes it again. The result is the final encryption key. The default is 1000.	For example: **-i 5000**
-h or **--hex**	Specify this option to obtain hexadecimal output. The default output is Base64.	For example: **-h**
--help	Displays information about command syntax and options.	**jasypt:encrypt --help**

3. Create a properties file that contains the encrypted values that you obtained by running the **jasypt:encrypt** command. Wrap each encrypted value in the **ENC()** function.
   For example, suppose you want to store some LDAP credentials in the **etc/ldap.properties** file. The file content would be something like this:

```
#ldap.properties
ldap.password=ENC(VMJ5S566MEDhQ5r6jiIqTB+fao3NN4pKnQ9xU0wiDCg=)
ldap.url=ldap://192.168.1.74:10389
```

4. Add the required namespaces for the encrypted property placeholders to your **blueprint.xml** file. These namespaces are for Aries extensions and Apache Karaf Jasypt. For example:

```
<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>
```

5. Configure the identifier for the Jasypt encryption algorithm that you used and the location of the properties file. The following example shows how to:

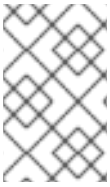
   - Configure the **ext:property-placeholder** element to read properties from the **etc/ldap.properties** file.
   - Configure the **enc:property-placeholder** element to:

- Identify the **PBEWithMD5AndDES** encryption algorithm.
- Read the master password from an environment variable, **JASYPT_ENCRYPTION_PASSWORD**, that you defined in the Karaf **bin/setenv** file.

```
<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>
    <ext:location>file:etc/ldap.properties</ext: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="passwordEnvName"
value="JASYPT_ENCRYPTION_PASSWORD" />
        </bean>
      </property>
    </enc:encryptor>
  </enc:property-placeholder>
…
</blueprint>
```

LDAP JAAS realm configuration that uses encrypted property placeholders

The following example adds to the **blueprint.xml** file in the previous example by showing an LDAP JAAS realm configuration that uses Jasypt encrypted property placeholders.



NOTE

When you use the process described in this topic to encrypt properties you cannot use the **@PropertyInject** annotation to decrypt the properties. Instead, use XML to inject properties into Java objects, as shown in this Blueprint example.

In this example, during container initialization, the **${ldap.password}** placeholder is replaced with the decrypted value of the **ldap.password** property from the **etc/ldap.properties** file.

```
<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="passwordEnvName" value="JASYPT_ENCRYPTION_PASSWORD" />
```

```
        </bean>
      </property>
    </enc:encryptor>
  </enc:property-placeholder>

  <jaas:config name="karaf" rank="200">
    <jaas:module className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule"
flags="required">
      initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
      debug=true
        connectionURL=${ldap.url}
        connectionUsername=cn=mqbroker,ou=Services,ou=system,dc=jbossfuse,dc=com
        connectionPassword=${ldap.password}
        connectionProtocol=
        authentication=simple
        userRoleName=cn
        userBase = ou=User,ou=ActiveMQ,ou=system,dc=jbossfuse,dc=com
        userSearchMatching=(uid={0})
        userSearchSubtree=true
        roleBase = ou=Group,ou=ActiveMQ,ou=system,dc=jbossfuse,dc=com
        roleName=cn
        roleSearchMatching= (member:=uid={1})
        roleSearchSubtree=true
    </jaas:module>
  </jaas:config>

</blueprint>
```

Examples of specifying environment variables or system properties

Rather than specifying your plain text master password when you encrypt a value, you can specify an environment variable or a system property that is set to your master password. For example, suppose that the **bin/setenv** file contains:

```
export MASTER_PASSWORD=passw0rd
```

You can encrypt a value with this command:

```
karaf@root()> jasypt:encrypt -w MASTER_PASSWORD "$en$!t!ve"
Algorithm used: PBEWithMD5AndDES
Encrypted data: /4DZCwqXD7cQ++TKQjt9QzmmcWv7TwmylCPkHumv2LQ=
```

If your **etc/system.properties** file contains:

```
master.password=passw0rd
```

You can encrypt a value with this command:

```
karaf@root()> jasypt:encrypt -w master.password "$en$!t!ve"
Algorithm used: PBEWithMD5AndDES
Encrypted data: 03+8UTJJtEXxHaJkVCmzhqLMUYtT8TBG2RMvOBQlfmQ=
```

2.3.3. Invoking the **jasypt:digest** command

A Jasypt digest is the result of applying cryptographic hash functions, such as MD5, to a value. Generating a digest is a type of one-way encryption. You cannot generate a digest and then reconstruct the original value from the digest. For especially sensitive values, you might want to generate a digest rather than encrypting a value. You can then specify the digest as a property placeholder.

The format for invoking the command to generate a digest is as follows:

jasypt:digest [*options*] [*input*]

If you do not specify any options, and you do not specify the input for which to create a digest, the command prompts you to specify the value that you want to encrypt and applies default values for options. For example:

```
karaf@root()> jasypt:digest
Input data to digest: ********
Input data to digest (repeat): ********
Algorithm used: MD5
Digest value:
8D4C0B3D5EE133BCFD7585A90F15C586741F814BC527EAE2A386B9AA6609B926AD9B3C418937
251373E08F18729AD2C93815A7F14D878AA0EF3268AA04729A614ECAE95029A112E9AD56FEDD
3FD7E28B73291C932B6F4C894737FBDE21AB382
```

The following example shows specification of the ***input*** argument on the command line:

```
karaf@root()> jasypt:digest ImportantPassword
```

This command applies default options and generates a digest that provides a one-way encryption of **ImportantPassword**. The command output looks something like this:

```
karaf@root()> jasypt:digest ImportantPassword
Algorithm used: MD5
Digest value:
0bL90nno/nHiTEdzx3dKa61LBDcWQQZMpjaONtY3b1fJBuDWbWTTtZ6tE5eOOPKh7orLTXS7XRt2bl
A2DrfnjWIIIETjge9n
```

Invoke the **jasypt:digest** command for each value for which you want one-way encryption.

To change the default behavior, specify one or more of the following options:

Option	Description	Example
-a or **--algorithm**	Follow this option with the identifier for the digest algorithm that you want the **jasypt:digest** command to use to generate the digest. The default is **MD5**. All digest algorithms that are in the list that the **jasypt-list-algorithms** command outputs are supported. Auto-completion is available when specifying algorithm names on the command line.	For example: **-a SHA-12**

Option	Description	Example
-i or **--iterations**	Follow this option with an integer that indicates the number of times to iteratively create a hash of the initial digest. Each iteration takes the previous hash result and hashes it again. The result is the final digest. The default is 1000.	For example: **-i 5000**
-s or **--salt-size**	Follow this option with an integer that indicates the number of bytes in the salt that **jasypt:digest** applies to create the digest. This is useful when you want to generate a digest for a sensitive value and you need to specify the digest in more than one location. For example, you can invoke **jasypt:digest** with the same input value but with different salt sizes. Each command generates a different digest even though the input was the same. The default is 8.	For example: **-s 12**
-h or **--hex**	Specify this option to obtain hexadecimal output. The default output is Base64.	For example: **-h**
--help	Displays information about command syntax and options.	**jasypt:digest --help**

After you obtain a digest, you can use it in the same way as described in Using encrypted property placeholders.

If you use non-default values, the calculation takes longer. For example:

```
karaf@root()> jasypt:digest --iterations 1000000 --salt-size 32 -a SHA-512 --hex passw0rd
Algorithm used: SHA-512
Digest value:
4007A85C4932A399D8376B4F2B3221E34F0AF349BB152BEAC80F03BEB2B368DA7900F0990C186
DB36D61741FA147B96DC9F73481991506FAA3662EA1693642CDAB89EB7E6B1DC21E1443D06D7
0A5842EB2851D37E262D5FC77A1D0909B3B2783
```

2.3.4. Invoking the jasypt:decrypt command

To verify the original value of an encryptyed placeholder, invoke the **jasypt:decrypt** command on the placeholder. You must have generated the placeholder by invoking the **jasypt:encrypt** command. You must know:

- The master password or the environment variable or system property that is set to the master password.
- The algorithm that the **jasypt:encrypt** command used to encrypt the value.
- The number of iterations that the **jasypt:encrypt** command used to encrypt the original value.

The format for invoking the **jasypt:decrypt** command is as follows:

jasypt:decrypt [*options*] [*input*]

If you do not specify any options, and you do not specify the encrypted input that you want to decrypt, then the command prompts you for your master password and for the value to decrypt, and applies default values for other options. For this example decryption to be successful, the **jasypt:encrypt** command must have used the defaults to encrypt the value. For example:

```
karaf@root()> jasypt:decrypt
Master password: ********
Data to decrypt: ******************************************
Algorithm used: PBEWithMD5AndDES
Decrypted data: $en$!t!ve
```

This command uses the master password that you specify and the default algorithm, which is **PBEWithMD5AndDES**, to create a decryption key. The command then uses this decryption key to decrypt the value that you enter at the prompt.

To change the default behavior, specify one or more of the following options:

Option	Description	Example
-w or **--password-property**	Follow this option with an environment variable or a system property that is set to the value of your master password. Jasypt uses this value, in conjunction with the decryption algorithm, to derive the initial decryption key. If you do not specify the **-w** or the **-W** option, after you invoke the command, it prompts you to enter and confirm your master password.	**-w MASTER_PW**
-W or **--password**	Follow this option with the plain text value of your chosen master password. The plain text value of your master password appears in history. Jasypt uses this value, in conjunction with the decryption algorithm, to derive the initial decryption key. If you do not specify the **-w** or the **-W** option, after you invoke the command, it prompts you to enter and confirm your master password.	**-W "M@s!erP#"**
-a or **--algorithm**	Follow this option with the identifier for the algorithm that you want the **jasypt:decrypt** command to use to derive the initial decryption key. The default is **PBEWithMD5AndDES**. The **jasypt:decrypt** command must use the same algortithm that the **jasypt:encrypt** command used to generate the specified placeholder input. All algorithms that are in the list that the **jasypt-list-algorithms** command outputs are supported. Auto-completion is available when specifying algorithm names on the command line.	For example: **-a PBEWITHMD5ANDRC2**

Option	Description	Example
-i or **--iterations**	Follow this option with an integer that indicates the number of times to iteratively create a hash of the initial key. Each iteration takes the previous hash result and hashes it again. The result is the final decryption key. The default is 1000. The **jasypt:decrypt** command must use the same number of iterations that the **jasypt:encrypt** command used to generate the specified placeholder input.	For example: **-i 5000**
-h or **--hex**	Specify this option to obtain hexadecimal output. The default output is Base64.	For example: **-h**
--help	Displays information about command syntax and options.	**jasypt:decrypt --help**

Examples of specifying environment variables or system properties

Rather than specifying your plain text master password when you decrypt a value, you can specify an environment variable or a system property that is set to your master password. For example, suppose that the **bin/setenv** file contains:

```
export MASTER_PASSWORD=passw0rd
```

You can decrypt a value with this command:

```
karaf@root()> jasypt:decrypt -a -w MASTER_PASSWORD
Data to decrypt: ********************************************
Algorithm used: PBEWithMD5AndDES
Decrypted data: $en$!t!ve
```

If your **etc/system.properties** file contains:

```
master.password=passw0rd
```

You can decrypt a value with this command:

```
karaf@root()> jasypt:decrypt -w master.password
Data to decrypt: ********************************************
Algorithm used: PBEWithMD5AndDES
Decrypted data: $en$!t!ve
```

2.4. ENABLING REMOTE JMX SSL

Overview

Red Hat JBoss Fuse provides a JMX port that allows remote monitoring and management of Karaf 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 Fuse 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 Karaf user with the **admin** role
  Edit the ***InstallDir*/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 Karaf installation:

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

Certificate Extensions: 1
[1]: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: AC 44 A5 F2 E6 2F B2 5A   5F 88 FE 69 60 B4 27 7D  .D.../.Z_..i`.'.
0010: B9 81 23 9C                                        ..#.
]
]

]
  Algorithm: [SHA256withRSA]
  Signature:
0000: 01 1D 95 C0 F2 03 B0 FD   CF 3A 1A 14 F5 2E 04 E5  .........:......
0010: DD 18 DD 0E 24 60 00 54   35 AE FE 36 7B 38 69 4C  ....$`.T5..6.8iL
0020: 1E 85 0A AF AE 24 1B 40   62 C9 F4 E5 A9 02 CD D3  .....$.@b.......
0030: 91 57 60 F6 EF D6 A4 84   56 BA 5D 21 11 F7 EA 09  .W`.....V.]!....
0040: 73 D5 6B 48 4A A9 09 93   8C 05 58 91 6C D0 53 81  s.kHJ.....X.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*/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-7.4.0.fuse-740028-redhat-00001/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 Karaf container, by copying it into the ***InstallDir*/deploy** directory (the hot deploy directory).

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

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

Edit the ***InstallDir*/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 Karaf container

You must restart the Karaf 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 Fuse installation:

```
cd <installDir>/jboss-fuse-7.4.0.fuse-740028-redhat-00001/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
```

2.5. USING AN ELYTRON CREDENTIAL STORE

Fuse includes the Elytron credential store feature that is part of JBoss EAP. A credential store can safely secure sensitive text strings by encrypting them in a storage file. Each container can have exactly one credential store.

In secure configurations, a typical problem is how to store passwords. For example, consider passwords for database access from various applications. For many authentication methods, passwords must be available in clear text before a server can send credentials to a database server. Storage of clear text passwords in text configuration files is generally not a good idea.

An Elytron credential store solves this problem. You securely store passwords and other sensitive values in a credential store, which is an encrypted file that complies with the PKCS#12 specification. A credential store does not store unencrypted values. The credential store uses PBE (Password Based Encryption) to encrypt both sensitive values, such as passwords, and the store itself.

The following topics provide details:

- Section 2.5.1, “Putting a credential store into use”
- Section 2.5.2, “Behavior when system properties hold credential store configuration”
- Section 2.5.3, “Description of credential store system properties and environment variables”

- Section 2.5.4, "**credential-store:create** command reference"
- Section 2.5.5, "**credential-store:store** command reference"
- Section 2.5.6, "**credential-store:list** command reference"
- Section 2.5.7, "**credential-store:remove** command reference"
- Section 2.5.8, "Example of Configuration Admin properties enabling credential store use"

2.5.1. Putting a credential store into use

In an Apache Karaf container that is running Fuse, to put a credential store into use, create and configure the credential store and then add values to it. Fuse continues to run and the credential store is available for use.

Prerequisites

- You want to use the following defaults when you create the credential store:
  - Create a PKCS#12 credential store.
  - Apply the **masked-SHA1-DES-EDE** algorithm to encrypt the credential store.
  - Iterate through the algorithm 200000 times.
  - Locate the credential store at **${karaf.etc}/credential.store.p12**.
- You want to save credential store configuration in **${karaf.etc}/system.properties**.

If you need to change any of these behaviors, see the information about invoking the **credential-store:create** command.

Procedure

1. Choose a credential store password.
   Later, when you add values to the credential store or when you want to decrypt values, a credential store command uses your credential store password to encrypt and decrypt the values.
2. Invoke the **credential-store:create** command, which prompts you to enter your chosen credential store password:

```
karaf@root()> credential-store:create --persist
Credential store password: *****
Credential store password (repeat): *****

Credential store configuration was persisted in ${karaf.etc}/system.properties and is effective.

Credential store was written to /data/servers/fuse-karaf-7.4.0.fuse-
740060/etc/credential.store.p12

By default, only system properties are encrypted.
Encryption of configuration admin properties can be enabled by setting
felix.cm.pm=elytron in etc/config.properties.
```

The command writes something like the following configuration in **etc/system.properties**:

```
credential.store.location = /data/servers/fuse-karaf-7.4.0.fuse-740060/etc/credential.store.p12
credential.store.protection.algorithm = masked-SHA1-DES-EDE
credential.store.protection.params =
MDkEKFJId25PaXIVQldKUWw5R2tLclhZQndpTGhhVXJsWG5lNVJMbTFCZEMCAwMNQAQI
0Whepb7H1BA=
credential.store.protection = m+1BcfRyCnI=
```

3. Add an encrypted value to the credential store by invoking the **credential-store:store** command as follows:
   credential-store:store *alias*

   Replace ***alias*** with a unique key value. Later, to retrieve the encrypted value that you are adding to the credential store, tools use this alias. For example, suppose you use the **db.password** system property in code, and your **etc/system.properties** file has an entry that sets the **db.password** property to the actual password for the database. The recommendation is to specify your system property, **db.password**, as the alias.

   After you invoke this command, it prompts you to enter and confirm the sensitive value that you want to add to the credential store. Continuing with the **db.password** alias example, at the prompt, you would enter the actual password for the database:

```
karaf@root()> credential-store:store db.password
Secret value to store: ******
Secret value to store (repeat): ******
Value stored in the credential store. To reference it use: CS:db.password
```

4. Update an entry in your **etc/system.properties** file or add a new entry. The entry that you update or add sets the alias that you specified in the **credential-store:store** command to the reference value that the command outputs. For example:

```
db.password = CS:db.password
```

   When Fuse is running with a configured credential store, it dynamically replaces each instance of, for example, the **db.password** system property, with the actual secret value that is in the credential store.

5. In the **credential-store:store** command, if the alias that you specified is a system property that is already in use, skip to the next step. If code is not already using the alias that you specified for the secret, then in each file that requires the secret, specify the alias, which you added as a system property in the previous step. For example, code would refer to **db.password**.

6. Repeat the previous three steps for each value that you want to add to the credential store.

Results

The credential store is ready for use. When Fuse starts or when the credential store bundle restarts, it processes system properties to find any that reference credential store entries. For each system property that does, Fuse obtains the associated value from the credential store and replaces the system property with the actual secret value. The actual secret value is then available to all components, bundles, and code that contain instances of that system property.

2.5.2. Behavior when system properties hold credential store configuration

Suppose that a credential store is in use and you are using system properties to hold its configuration parameters. When Fuse starts it processes all system properties. Fuse replaces system properties that are set to values that have the **CS:** prefix with the associated value that is in the credential store. Fuse proxies the **java.lang:type=Runtime** JMX MBean so that each call to the JMX **getSystemProperties()** method hides decrypted values.

For example, consider a credential store with one entry:

```
karaf@root()> credential-store:list --show-secrets
Alias        │ Reference       │ Secret value
──────────────┼─────────────────┼──────────────
db.password │ CS:db.password │ sec4et
```

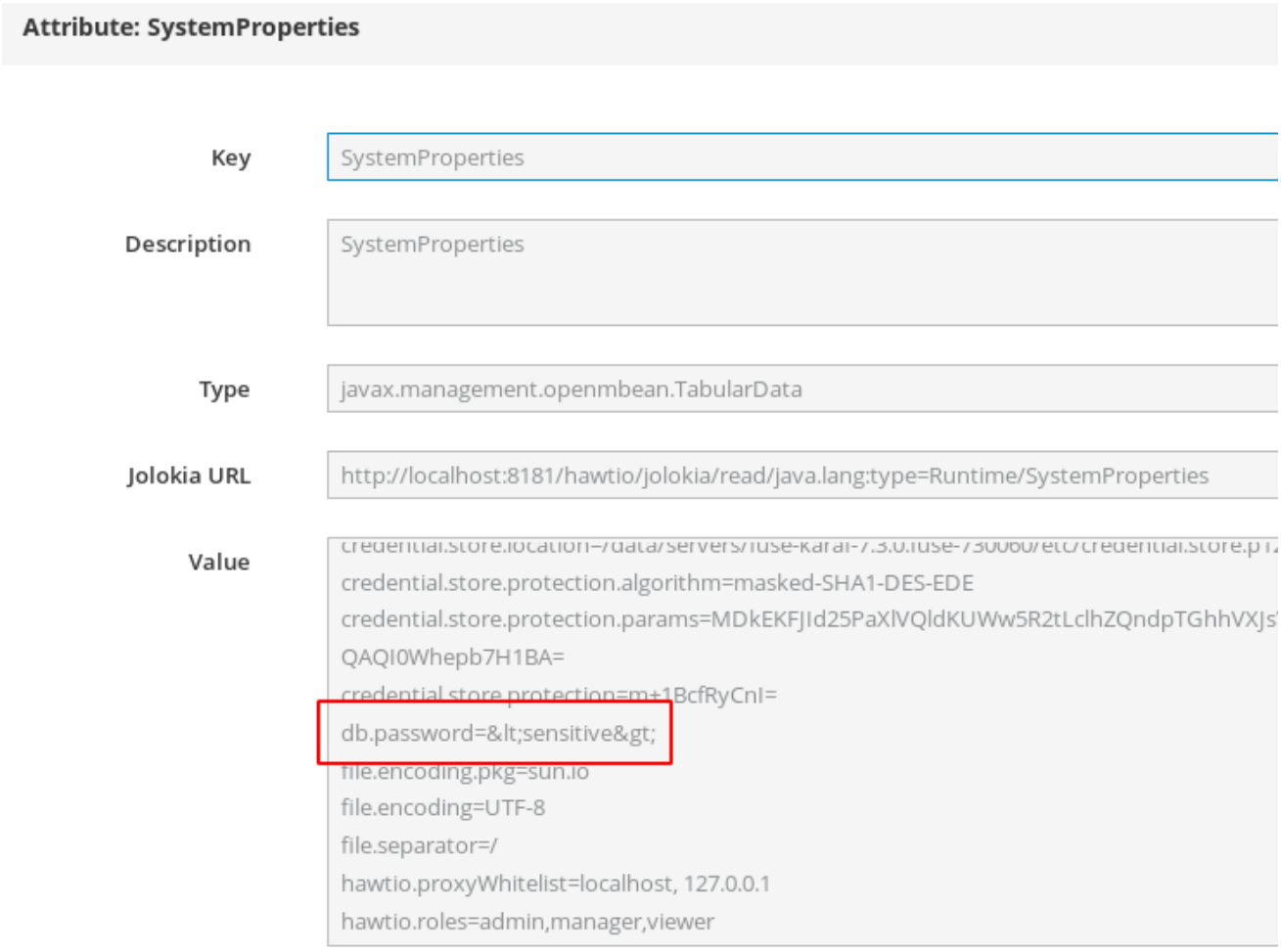
Assume that after you added this entry to the credential store, you edited the **etc/system.properties** file to add this entry:

db.password = CS:db.password

When Fuse starts or when you restart the **org.jboss.fuse.modules.fuse-credential-store-core** bundle, Fuse checks for any references to the **db.password** system property. For each reference, Fuse uses the **CS:db.password** alias to obtain the associated value from credential store. You can check this by invoking the following command:

```
karaf@root()> system:property db.password
sec4et
```

However, if you use JMX to check this, the value from the credential store is hidden:



2.5.3. Description of credential store system properties and environment variables

You can use system properties or environment variables to hold credential store configuration parameters. The options that you specify when you create a credential store determine:

- Whether you must set the properties or variables yourself.
- The exact value that a property or variable is, or must be, set to.

An understanding of the properties/variables helps you understand how a credential store works.

When you invoke the **credential-store:create** command and specify only the **--persist** option, the command sets system properties to credential store configuration parameters. You do not need to explicitly set credential store system properties.

To use credential store environment variables instead or to change the default behavior of the **credential-store:create** command, see **credential-store:create** command reference for details about options that you can specify when you create a credential store.

When you invoke the command that creates a credential store, any options that you specify determine the settings of the credential store properties or variables. If you must set properties or variables yourself, output from the **credential-store:create** command contains instructions for doing that. In other words, it is never up to you to decide what the setting of credential store system properties or environment variables should be. Execution of the **credential-store:create** command always determines the settings.

The following table describes the credential store properties and variables. For a particular parameter, if both the environment variable and the system property are set, the environment variable setting has precedence.

Name	Description
Environment variable: **CREDENTIAL_STORE_PROTECION_ALGORITHM** System property: **credential.store.protection.algorithm**	The Password Based Encryption (PBE) algorithm that credential store commands use to derive an encryption key.
Environment variable: **CREDENTIAL_STORE_LOCATION** System property: **credential.store.location**	Location of the credential store.
Environment variable: **CREDENTIAL_STORE_PROTECTION_PARAMS** System property: **credential.store.protection.params**	Parameters that a credential store uses to derive an encryption key. Parameters include iteration count, initial vector, and salt.

Name	Description
Environment variable: **CREDENTIAL_STORE_PROTECTION** System property: **credential.store.protection**	Password that a credential store command must decrypt to recover passwords or other secure data from a credential store. When you invoke the **credential-store:create** command, the command prompts you to specify a password. The encryption of that password is the setting of this environment variable or system property.

2.5.4. **credential-store:create** command reference

To create and configure a credential store, invoke the **credential-store:create** command, which has the following format:

credential-store:create [*options*]

When you do not specify any options, the command does the following:

- Prompts you for your chosen credential store password.
- Creates a PKCS#12 credential store
- Use the **masked-SHA1-DES-EDE** algorithm to encrypt the credential store
- Iterates through the algorithm 200000 times
- Locates the credential store at **${karaf.etc}/credential.store.p12**
- Does not store the credential store configuration

The following table describes the options, which you can specify to change the default behavior.

Option	Description
-w or **--password-property**	Follow this option with an environment variable or a system property that is set to the value of your master password. The credential store uses this value, in conjunction with an algorithm, to derive the encryption or decryption key. If you do not specify the **-w** or the **-W** option, after you invoke the command, it prompts you to enter and confirm your master password. For example: **-w MASTER_PW**

Option	Description
-W or **--password**	Follow this option with the plain text value of your chosen master password. The plain text value of your master password appears in history. The credential store uses this value, in conjunction with an algorithm, to derive the encryption or decryption key. If you do not specify the **-w** or the **-W** option, after you invoke the command, it prompts you to enter and confirm your master password. For example: **-W "M@s!erP#"**
-f or **--force**	Forces creation of the credential store. If a credential store exists at the intended location of the new credential store, specification of this option causes the command to overwrite the existing credential store. Any content in the existing credential store is lost. The default behavior is that the command does not create a credential store if there is already a credential store in the intended location.
-l or **--location**	Specifies the location for the new credential store. The recommendation is to use the default location, which is **${karaf.etc}/credential.store.p12**.
-ic or **--iteration-count**	Follow this option with an integer that indicates the number of times to iteratively apply the encryption algorithm being used. Each iteration takes the previous result and applies the algorithm again. The result is the final masked password. The default is 200000.
-a or **--algorithm**	Follow this option with the identifier for the algorithm that you want the **credential-store:create** command to use to generate the masked password. The recommendation is to use the default, which is **masked-SHA1-DES-EDE**.
-p or **--persist**	Stores the configuration of the new credential store in **${karaf.etc}/system.properties**. If you do not specify this option, the **credential-store:create** command sends the configuration information to the console with instructions for what to do next. See the example after this table. A reason to omit this option is because you want to see the credential store configuration parameter values. Or, you might omit this option because you plan to pass credential store configuration parameters to an application without using the **etc/system.properties** file.
--help	Displays information about command syntax and options.

Example of creating a credential store without specifying **--persist**

The following command creates a credential store but does not save the credential store configuration in **${karaf.etc}/system.properties**. The command uses the **masked-SHA1-DES-EDE** algorithm, which is the default.

```
karaf@root()> credential-store:create
Credential store password: *****
Credential store password (repeat): *****

Credential store was written to /data/servers/fuse-karaf-7.4.0.fuse-740060/etc/credential.store.p12

By default, only system properties are encrypted.
Encryption of configuration admin properties can be enabled by
setting felix.cm.pm=elytron in etc/config.properties.

Credential store configuration was not persisted and is not
effective. Please use one of the following configuration options and restart Fuse.
Option #1: Configure these system properties (e.g., in etc/system.properties):
 - credential.store.protection.algorithm=masked-SHA1-DES-EDE
 -
credential.store.protection.params=MDkEKGdOSkpRWXpndjhkVVZYbHF4elVpbUszNW0wc3NXczhNS
1A5cVlhZzcCAwMNQAQIDPzQ+BDGwX4=
 - credential.store.protection=0qudlx1XZFM=
 - credential.store.location=/data/servers/fuse-karaf-7.4.0.fuse-740060/etc/credential.store.p12
Option #2: Configure these environmental variables (e.g., in bin/setenv):
 - CREDENTIAL_STORE_PROTECTION_ALGORITHM=masked-SHA1-DES-EDE
 -
CREDENTIAL_STORE_PROTECTION_PARAMS=MDkEKGdOSkpRWXpndjhkVVZYbHF4elVpbUszN
W0wc3NXczhNS1A5cVlhZzcCAwMNQAQIDPzQ+BDGwX4=
 - CREDENTIAL_STORE_PROTECTION=0qudlx1XZFM=
 - CREDENTIAL_STORE_LOCATION=/data/servers/fuse-karaf-7.4.0.fuse-
740060/etc/credential.store.p12
```

2.5.5. credential-store:store command reference

To add an encrypted value to the credential store, invoke the **credential-store:store** command, which has the following format:

credential-store:store *alias* [*secret*]

Replace ***alias*** with a unique key value. To retrieve the encrypted value that you are adding to the credential store, tools use this alias.

Optionally, replace ***secret*** with the value that you want to encrypt and add to the credential store. Typically, this is a password, but it can be any value that you want to encrypt.

If you specify ***secret*** on the command line, its plain text value appears in history. If you do not specify ***secret*** on the command line, then the command prompts you for it and the value does not appear in history.

To view information about the command, enter:

credential-store:store --help.

The following command line is an example of adding an entry to the credential store:

```
karaf@root()> credential-store:store db.password sec4et
Value stored in the credential store. To reference it use: CS:db.password
```

The credential store now has an entry that can be referenced by specifying **CS:db.password**.

2.5.6. **credential-store:list** command reference

To obtain the alias for an entry in the credential store, invoke the **credential-store:list** command, which display a list of all entries in the credential store. For example:

```
karaf@root()> credential-store:list
Alias        │ Reference
──────────────────────┼──────────────────────────
db.password  │ CS:db.password
db2.password | CS:db2.password
```

To also list decryptions of the secret values that are encrypted in the credential store, invoke the command as follows:

```
karaf@root()> credential-store:list --show-secrets
Alias        │ Reference        │ Secret value
──────────────────────┼──────────────────────────┼──────────────────
db.password  │ CS:db.password   │ sec4et
db2.password | CS:db2.password | t0pSec4et
```

To view information about the command:

```
karaf@root()> credential-store:list --help
```

2.5.7. **credential-store:remove** command reference

To remove an entry from a credential store, invoke the **credential-store:remove** command, which has the following format:

credential-store:remove *alias*

Replace ***alias*** with the unique key value that you specified for the ***alias*** argument when you added the entry to the credential store. Do not specify the **CS:** prefix. You can invoke the **credential-store:list** command to obtain the alias.

The **credential-store:remove** command checks the credential store for an entry that has the alias that you specified, and if found, removes it. For example:

```
karaf@root()> credential-store:remove db.password

Alias        │ Reference        │ Secret value
──────────────────────┼──────────────────────────┼──────────────────
db2.password | CS:db2.password | t0pSec4et
```

To view information about the command:

karaf@root()> credential-store:remove --help

2.5.8. Example of Configuration Admin properties enabling credential store use

In a development environment, you can use Configuration Admin service properties to enable the use of a credential store. Configuration Admin properties are defined in **etc/*.cfg** files.

IMPORTANT

The use of Configuration Admin properties to enable the use of a credential store is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process. For more information about the support scope of Red Hat Technology Preview features, see https://access.redhat.com/support/offerings/techpreview/.

Preparation

- Invoke the **credential-store:create** command to create a credential store. See **credential-store:create command reference**.
- Enable the use of Configuration Admin properties by editing the **etc/config.properties** file to uncomment the line that contains **felix.cm.pm = elytron**:

```
# When uncommented, configuration properties handled by Configuration Admin service will be encrypted when storing
# in etc/ and in bundle data. Values of the properties will actually be aliases to credential store entries.
# Please consult the documentation for more details.
felix.cm.pm = elytron
```

What happens when Fuse starts

1. The **felix.configadmin** bundle:
   - Delays registering the **ConfigurationAdmin** service because the **felix.cm.pm** property is set.
   - Waits for the availability of the **org.apache.felix.cm.PersistenceManagerOSGi** service with the **name=cm** OSGi service registration property.
2. The Fuse credential store bundle:
   a. Loads the credential store by using the values set for the **credential.store.*** system properties or **CREDENTIAL_STORE_*** environment variables.
   b. Registers an OSGi service that implements the the **org.apache.felix.cm.PersistenceManagerOSGi** service.

   If anything fails, the credential store bundle registers the **PersistenceManager** service, which does nothing special. When something is broken or when the credential store is not available, Fuse should be able to read unencrypted configuration values. Encrypted values, specified with the **CS:** prefix are lost unless you remember the original values or you are able to recover the credential store and its configuration.
3. The **felix.configadmin** process uses the new persistence manager service to load and store the credential store configuration.

Example

Suppose the credential store has two entries:

```
karaf@root()> credential-store:list --show-secrets
Alias        │ Reference       │ Secret value
──────────────┼────────────────────────┼──────────────────
db.password │ CS:db.password │ sec4et
http.port   │ CS:http.port   │ 8182
```

In a Configuration Admin service configuration, you choose to use the alias for a sensitive value instead of the actual value. For example, you change a web configuration property as follows:

```
karaf@root()> config:property-list --pid org.ops4j.pax.web
  javax.servlet.context.tempdir = /data/servers/fuse-karaf-7.4.0.fuse-740060/data/pax-web-jsp
  org.ops4j.pax.web.config.file = /data/servers/fuse-karaf-7.4.0.fuse-740060/etc/undertow.xml
  org.ops4j.pax.web.session.cookie.httpOnly = true
  org.osgi.service.http.port = 8181

karaf@root()> config:property-set --pid org.ops4j.pax.web org.osgi.service.http.port CS:http.port

karaf@root()> config:property-list --pid org.ops4j.pax.web
  javax.servlet.context.tempdir = /data/servers/fuse-karaf-7.4.0.fuse-740060/data/pax-web-jsp
  org.ops4j.pax.web.config.file = /data/servers/fuse-karaf-7.4.0.fuse-740060/etc/undertow.xml
  org.ops4j.pax.web.session.cookie.httpOnly = true
  org.osgi.service.http.port = CS:http.port
```

In logs, the actual value, **8182** can appear, as you can see at the end of the following line. Whether a log shows the actual text value is determined by the component that consumes the encrypted value.

```
2019-03-12 15:36:25,648 INFO  {paxweb-config-2-thread-1} (ServerControllerImpl.java:458) :
Starting undertow http listener on 0.0.0.0:8182
```

In the previous commands, the second **config:property-list --pid org.ops4j.pax.web** command displays **CS:http.port** instead of **8182**, though the property has a numeric value. The **pax-web-undertow** process starts on this port. This is because OSGi hooks prevent the **felix.fileinstall** process, which shows the output of the **config:property-list --pid org.ops4j.pax.web** command, from seeing decrypted (dereferenced) values. This is also the reason why the **etc/org.ops4j.pax.web.cfg** file does not store decrypted (dereferenced) values, but instead stores, for example:

```
org.osgi.service.http.port = CS:http.port

org.ops4j.pax.web.config.file = ${karaf.etc}/undertow.xml
org.ops4j.pax.web.session.cookie.httpOnly = true

javax.servlet.context.tempdir = ${karaf.data}/pax-web-jsp
```

CHAPTER 3. SECURING THE UNDERTOW HTTP SERVER

Abstract

You can configure the built-in Undertow HTTP server to use SSL/TLS security by editing the contents of the **etc/undertow.xml** configuration file. In particular, you can add SSL/TLS security to the Fuse Console in this way.

3.1. UNDERTOW SERVER

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

- Fuse Console (by default, **http://localhost: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 Undertow server for all of your HTTP endpoints, you can conveniently add SSL/TLS security to these HTTP endpoints by following the steps described here.

3.2. CREATE X.509 CERTIFICATE AND PRIVATE KEY

Before you can enable SSL/TLS on the Undertow server, you must create an X.509 certificate and private key, where the certificate and private key must be in Java keystore format (JKS format). For details of how to create a signed certificate and private key, see Appendix A, *Managing Certificates*.

3.3. ENABLING SSL/TLS FOR UNDERTOW IN AN APACHE KARAF CONTAINER

For the following procedure, it is assumed that you have already created a signed X.509 certificate and private key pair in the keystore file, **alice.ks**, with keystore password, **StorePass**, and key password, **KeyPass**.

To enable SSL/TLS for Undertow in a Karaf container:

1. Make sure that the Pax Web server is configured to take its configuration from the **etc/undertow.xml** file. When you look at the contents of the **etc/org.ops4j.pax.web.cfg** file, you should see the following setting:

   ```
   org.ops4j.pax.web.config.file=${karaf.etc}/undertow.xml
   ```

2. Open the file, **etc/org.ops4j.pax.web.cfg**, in a text editor and add the following line:

   ```
   org.osgi.service.http.port.secure=8443
   ```

   Save and close the file, **etc/org.ops4j.pax.web.cfg**.

3. Open the file, **etc/undertow.xml**, in a text editor. The next steps assume you are working with the default **undertow.xml** file, unchanged since installation time.

4. Search for the XML elements, **http-listener** and **https-listener**. Comment out the **http-listener** element (by enclosing it between **<!--** and **-->**) and uncomment the **https-listener** element (spread over two lines). The edited fragment of XML should now look something like this:

```
<!-- HTTP(S) Listener references Socket Binding (and indirectly - Interfaces) -->
<!-- http-listener name="http" socket-binding="http" /> -->
<!-- verify-client: org.xnio.SslClientAuthMode.NOT_REQUESTED,
org.xnio.SslClientAuthMode.REQUESTED, org.xnio.SslClientAuthMode.REQUIRED -->
<https-listener name="https" socket-binding="https"
      security-realm="https" verify-client="NOT_REQUESTED" />
```

5. Search for the **w:keystore** element. By default, the **w:keystore** element is configured as follows:

```
<w:keystore path="${karaf.etc}/certs/server.keystore" provider="JKS" alias="server"
          keystore-password="secret" key-password="secret"
          generate-self-signed-certificate-host="localhost" />
```

   To install the **alice** certificate as the Undertow server's certificate, modify the **w:keystore** element attributes as follows:

   - Set **path** to the absolute location of the **alice.ks** file on the file system.
   - Set **provider** to **JKS**.
   - Set **alias** to the **alice** certificate alias in the keystore.
   - Set **keystore-password** to the value of the password that unlocks the key store.
   - Set **key-password** to the value of the password that encrypts the **alice** private key.
   - Delete the **generate-self-signed-certificate-host** attribute setting.

6. For example, after installing the **alice.ks** keystore, the modified **w:keystore** element would look something like this:

```
<w:keystore path="${karaf.etc}/certs/alice.ks" provider="JKS" alias="alice"
          keystore-password="StorePass" key-password="KeyPass" />
```

7. Search for the **<interface name="secure">** tag, which is used to specify the IP addresses the secure HTTPS port binds to. By default, this element is commented out, as follows:

```
<!--<interface name="secure">-->
   <!--<w:inet-address value="127.0.0.1" />-->
<!--</interface>-->
```

   Uncomment the element and customize the **value** attribute to specify the IP address which the HTTPS port binds to. For example, the wildcard value, **0.0.0.0**, configures HTTPS to bind to all available IP addresses:

```
<interface name="secure">
   <w:inet-address value="0.0.0.0" />
</interface>
```

8. Search for and uncomment the **<socket-binding name="https"** tag. When this tag is uncommented, it should look something like this:

```
<socket-binding name="https" interface="secure" port="${org.osgi.service.http.port.secure}"
/>
```

9. Save and close the file, **etc/undertow.xml**.

10. Restart the Fuse container, in order for the configuration changes to take effect.

3.4. CUSTOMIZING ALLOWED TLS PROTOCOLS AND CIPHER SUITES

You can customize the allowed TLS protocols and cipher suites by modifying the following attributes of the **w:engine** element in the **etc/undertow.xml** file:

enabled-cipher-suites

Specifies the list of allowed TLS/SSL cipher suites.

enabled-protocols

Specifies the list of allowed TLS/SSL protocols.

> **WARNING**
>
> Do not enable **SSL** protocol versions, as they are vulnerable to attack. Use only **TLS** protocol versions.

For full details of the available protocols and cipher suites, consult the appropriate JVM documentation and security provider documentation. For example, for Java 8, see Java Cryptography Architecture Oracle Providers Documentation for JDK 8.

3.5. CONNECT TO THE SECURE CONSOLE

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

```
https://localhost:8443/hawtio
```

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

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

CHAPTER 4. 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.

4.1. SECURE ACTIVEMQ CONNECTION FACTORY

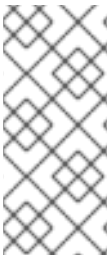
Overview

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

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

Programming the security properties

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



NOTE

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

Defining a secure connection factory

Example 4.1, “Defining a Secure Connection Factory Bean” shows how to create a secure connection factory bean in Blueprint, enabling both SSL/TLS security **and** JAAS authentication.

Example 4.1. Defining a Secure Connection Factory Bean

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

■

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

brokerURL

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

userName and **password**

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

trustStore

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

trustStorePassword

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

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

4.2. EXAMPLE CAMEL ACTIVEMQ COMPONENT CONFIGURATION

Overview

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

Prerequisites

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

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

Sample Camel ActiveMQ component

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

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ... >
  ...
  <!--
    Configure the activemqssl component:
  -->
  <bean id="jmsConnectionFactory"
      class="org.apache.activemq.ActiveMQSslConnectionFactory">
    <property name="brokerURL" value="ssl://localhost:61617" />
```

```
    <property name="userName" value="Username"/>
    <property name="password" value="Password"/>
    <property name="trustStore" value="/conf/client.ts"/>
    <property name="trustStorePassword" value="password"/>
  </bean>

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

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

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

</beans>
```

Sample Camel route

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

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

CHAPTER 5. SECURING THE CAMEL CXF COMPONENT

Abstract

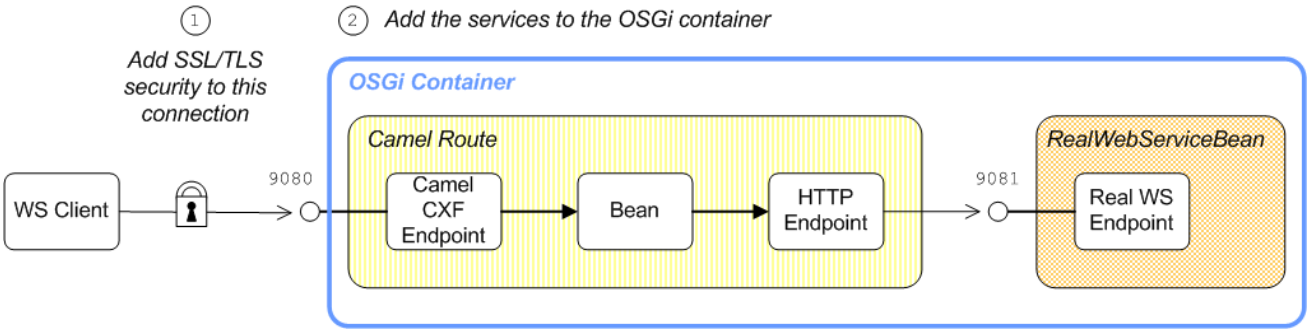
This chapter explains how to enable SSL/TLS security on a Camel CXF endpoint, using the Camel CXF proxy demonstration as the starting point. The Camel CXF component enables you to add Apache CXF endpoints to your Apache Camel routes. This makes it possible to simulate a Web service in Apache Camel or you could interpose a route between a WS client and a Web service to perform additional processing (which is the case considered here).

5.1. THE CAMEL CXF PROXY DEMONSTRATION

Overview

In order to explain how to secure a Camel CXF endpoint in OSGi, this tutorial builds on an example available from the standalone distribution of Apache Camel, the **Camel CXF proxy** demonstration. Figure 5.1, "Camel CXF Proxy Overview" gives an overview of how this demonstration works

Figure 5.1. Camel CXF Proxy Overview



The **report incident** Web service, which is implemented by the **RealWebServiceBean**, receives details of an incident (for example, a traffic accident) and returns a tracking code to the client. Instead of sending its requests directly to the real Web service, however, the WS client connects to a Camel CXF endpoint, which is interposed between the WS client and the real Web service. The Apache Camel route performs some processing on the WSDL message (using the **enrichBean**) before forwarding it to the real Web service.

> **WARNING**
>
> If you 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.

Modifications

In order to demonstrate how to enable SSL/TLS on a Camel CXF endpoint in the context of OSGi, this chapter contains instructions on how to modify the basic demonstration as follows:

1. SSL/TLS security is enabled on the connection between the WS client and the Camel CXF endpoint.
2. The Apache Camel route and the **RealWebServiceBean** bean are both deployed into the OSGi container.

Obtaining the demonstration code

The Camel CXF proxy demonstration is available only from the standalone distribution of Apache Camel, which is included in the ***InstallDir*/extras** directory. Using a standard archive utility, expand the Camel archive file and extract the contents to a convenient location on your filesystem.

Assuming that you have installed Apache Camel in *CamelInstallDir*, you can find the Camel CXF proxy demonstration in the following directory:

```
CamelInstallDir/examples/camel-example-cxf-proxy
```

Obtaining the sample certificates

This demonstration needs X.509 certificates. In a real deployment, you should generate these certificates yourself using a private certificate authority. For this demonstration, however, we use some sample certificates from the Apache CXF **wsdl_first_http** example. This demonstration is available from the standalone distribution of Apache CXF, which is included in the ***InstallDir*/extras** directory. Using a standard archive utility, expand the CXF archive file and extract the contents to a convenient location on your filesystem.

Assuming that you have installed Apache CXF in *CXFInstallDir*, you can find the **wsdl_first_http** demonstration in the following directory:

```
CXFInstallDir/samples/wsdl_first_http
```

Physical part of the WSDL contract

The physical part of the WSDL contract refers to the **wsdl:service** and **wsdl:port** elements. These elements specify the transport details that are needed to connect to a specific Web services endpoint. For the purposes of this demonstration, this is the most interesting part of the contract and it is shown in Example 5.1, "The ReportIncidentEndpointService WSDL Service".

Example 5.1. The ReportIncidentEndpointService WSDL Service

```
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   ...
 xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
 targetNamespace="http://reportincident.example.camel.apache.org">
   ...
   <!-- Service definition -->
   <wsdl:service name="ReportIncidentEndpointService">
      <wsdl:port name="ReportIncidentEndpoint" binding="tns:ReportIncidentBinding">
         <soap:address location="http://localhost:9080/camel-example-cxf-
proxy/webservices/incident"/>
      </wsdl:port>
   </wsdl:service>

</wsdl:definitions>
```

NOTE

The address URL appearing in the WSDL contract (the value of the **soap:address** element's **location** attribute) is not important here, because the application code overrides the default value of the address URL.

WSDL addressing details

A WS client needs three pieces of information to connect to a WSDL service: the *WSDL service name*, the *WSDL port name*, and the *address URL* of the Web service. The following addressing details are used to connect to the proxy Web service and to the real Web service in this example:

WSDL service name

The full QName of the WSDL service is as follows:

```
{http://reportincident.example.camel.apache.org}ReportIncidentEndpointService
```

WSDL port name

The full QName of the WSDL port is as follows:

```
{http://reportincident.example.camel.apache.org}ReportIncidentEndpoint
```

Address URL

The address URL of the **proxy Web service** endpoint (which uses the HTTPS protocol) is as follows:

```
https://localhost:9080/camel-example-cxf-proxy/webservices/incident
```

NOTE

The preceding address is specified when the **reportIncident** bean is created using a **cxf:cxfEndpoint** element in the bundle's Spring configuration file, **src/main/resources/META-INF/spring/camel-config.xml**.

The address URL of the **real Web service** endpoint (using the HTTP protocol) is as follows:

```
http://localhost:9081/real-webservice
```

NOTE

The preceding address is specified when the **realWebService** bean is created in the bundle's Spring configuration file, **src/main/resources/META-INF/spring/camel-config.xml**.

5.2. SECURING THE WEB SERVICES PROXY

Overview

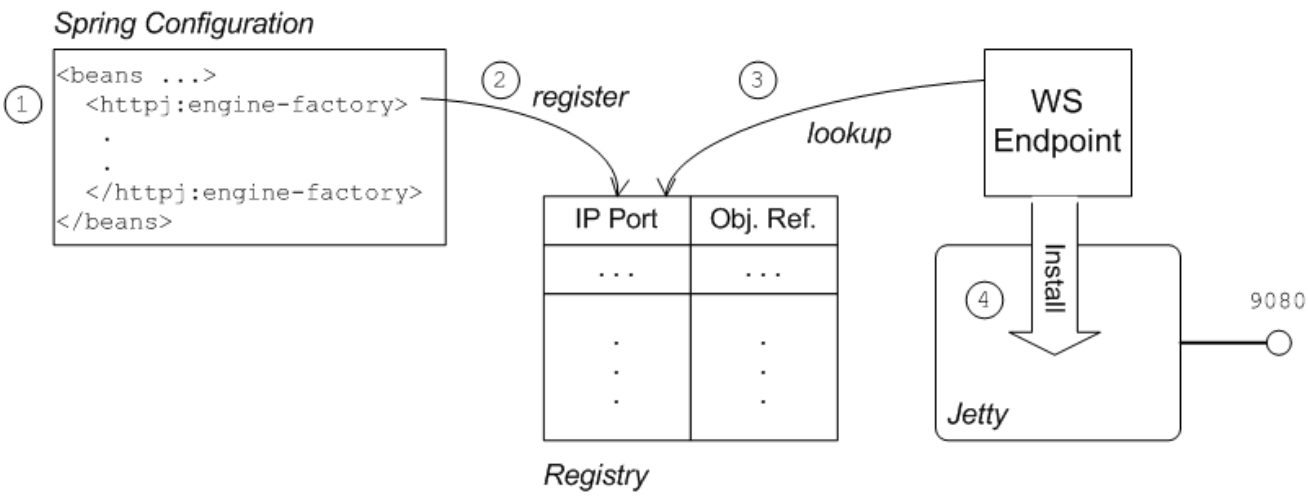
This section explains how to enable SSL/TLS security on the Camel CXF endpoint, which acts as a proxy for the real Web service. Assuming that you already have the X.509 certificates available, all that is required is to add a block of configuration data to the Spring configuration file (where the configuration data is contained in a **httpj:engine-factory** element). There is just one slightly subtle aspect to this, however: you need to understand how the Camel CXF endpoint gets associated with the SSL/TLS configuration details.

Implicit configuration

A WS endpoint can be configured by creating the endpoint in Spring and then configuring SSL/TLS properties on its Jetty container. The configuration can be somewhat confusing, however, for the following reason: the Jetty container (which is configured by a **httpj:engine-factory** element in Spring) **does not explicitly reference the WS endpoints it contains** and the WS endpoints **do not explicitly reference the Jetty container** either. The connection between the Jetty container and its contained endpoints is established implicitly, in that they are both configured to use the same TCP port, as illustrated by WS Endpoint Implicitly Configured by httpj:engine-factory .

WS Endpoint Implicitly Configured by httpj:engine-factory

Element



The connection between the Web service endpoint and the **httpj:engine-factory** element is established as follows:

1. The Spring container loads and parses the file containing the **httpj:engine-factory** element.
2. When the **httpj:engine-factory** bean is created, a corresponding entry is created in the registry, storing a reference to the bean. The **httpj:engine-factory** bean is also used to initialize a Jetty container that listens on the specified TCP port.
3. When the WS endpoint is created, it scans the registry to see if it can find a **httpj:engine-factory** bean with the same TCP port as the TCP port in the endpoint's address URL.
4. If one of the beans matches the endpoint's TCP port, the WS endpoint installs itself into the corresponding Jetty container. If the Jetty container has SSL/TLS enabled, the WS endpoint shares those security settings.

Steps to add SSL/TLS security to the Jetty container

To add SSL/TLS security to the Jetty container, thereby securing the WS proxy endpoint, perform the following steps:

1. the section called "Add certificates to the bundle resources" .
2. the section called "Modify POM to switch off resource filtering" .
3. the section called "Instantiate the CXF Bus" .
4. the section called "Add the httpj:engine-factory element to Spring" .
5. the section called "Define the cxfcore:, sec: and httpj: prefixes" .
6. the section called "Modify proxy address URL to use HTTPS" .

Add certificates to the bundle resources

The certificates used in this demonstration are taken from a sample in the Apache CXF 3.2.7.fuse-740026-redhat-00001 product. If you install the standalone version of Apache CXF (available in the ***InstallDir*/extras/** directory), you will find the sample certificates in the ***CXFInstallDir*/samples/wsdl_first_https/src/main/config** directory.

Copy the **clientKeystore.jks** and **serviceKeystore.jks** keystores from the ***CXFInstallDir*/samples/wsdl_first_https/src/main/config** directory to the ***CamelInstallDir*/examples/camel-example-cxf-proxy/src/main/resources/certs** directory (you must first create the **certs** sub-directory).

Modify POM to switch off resource filtering

Including the certificates directly in the bundle as resource is the most convenient way to deploy them. But when you deploy certificates as resources in a Maven project, you must remember to disable Maven resource filtering, which corrupts binary files.

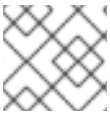
To disable filtering of **.jks** files in Maven, open the project POM file, ***CamelInstallDir*/examples/camel-example-cxf-proxy/pom.xml**, with a text editor and add the following **resources** element as a child of the **build** element:

```
<?xml version="1.0" encoding="UTF-8"?>
...
<project ...>
  ...
  <build>
    <plugins>
      ...
    </plugins>

    <resources> <resource> <directory>src/main/resources</directory> <filtering>true</filtering>
<excludes> <exclude>/.jks</exclude> </excludes> </resource> <resource>
<directory>src/main/resources</directory> <filtering>false</filtering> <includes>
<include>/.jks</include> </includes> </resource> </resources>
  </build>

</project>
```

Instantiate the CXF Bus

You should instantiate the CXF bus explicitly in the Spring XML (this ensures that it will be available to the Jetty container, which is instantiated by the **httpj:engine-factory** element in the next step). Edit the **camel-config.xml** file in the **src/main/resources/META-INF/spring** directory, adding the **cxfcore:bus** element as a child of the **beans** element, as follows:

```
<beans ... >
  ...
  <cxfcore:bus/>
  ...
</beans>
```



NOTE

The **cxfcore:** namespace prefix will be defined in a later step.

Add the httpj:engine-factory element to Spring

configuration

To configure the Jetty container that listens on TCP port 9080 to use SSL/TLS security, edit the **camel-config.xml** file in the **src/main/resources/META-INF/spring** directory, adding the **httpj:engine-factory** element as shown in Example 5.2, "httpj:engine-factory Element with SSL/TLS Enabled".

In this example, the **required** attribute of the **sec:clientAuthentication** element is set to **false**, which means that a connecting client is **not** required to present an X.509 certificate to the server during the SSL/TLS handshake (although it may do so, if it has such a certificate).

Example 5.2. httpj:engine-factory Element with SSL/TLS Enabled

```
<beans ... >
  ...
  <httpj:engine-factory bus="cxf">
    <httpj:engine port="${proxy.port}">
      <httpj:tlsServerParameters secureSocketProtocol="TLSv1">
        <sec:keyManagers keyPassword="skpass">
          <sec:keyStore resource="certs/serviceKeystore.jks" password="sspass" type="JKS"/>
        </sec:keyManagers>
        <sec:trustManagers>
          <sec:keyStore resource="certs/serviceKeystore.jks" password="sspass" type="JKS"/>
        </sec:trustManagers>
        <sec:cipherSuitesFilter>
          <sec:include>.*_WITH_3DES_.*</sec:include>
          <sec:include>.*_WITH_DES_.*</sec:include>
          <sec:exclude>.*_WITH_NULL_.*</sec:exclude>
          <sec:exclude>.*_DH_anon_.*</sec:exclude>
        </sec:cipherSuitesFilter>
        <sec:clientAuthentication want="true" required="false"/>
      </httpj:tlsServerParameters>
    </httpj:engine>
  </httpj:engine-factory>

</beans>
```

IMPORTANT

You must set secureSocketProtocol to **TLSv1** on the server side, in order to protect against the Poodle vulnerability (CVE-2014-3566)

Define the cxfcore:, sec: and httpj: prefixes

Define the **cxfcore:**, **sec:** and **httpj:** namespace prefixes, which appear in the definitions of the **cxfcore:bus** element and the **httpj:engine-factory** element, by adding the following highlighted lines to the **beans** element in the **camel-config.xml** file:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:camel="http://camel.apache.org/schema/spring"
    xmlns:cxf="http://camel.apache.org/schema/cxf"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:cxfcore="http://cxf.apache.org/core"
    xmlns:sec="http://cxf.apache.org/configuration/security"
    xmlns:httpj="http://cxf.apache.org/transports/http-jetty/configuration"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
    http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd
    http://camel.apache.org/schema/cxf http://camel.apache.org/schema/cxf/camel-cxf.xsd
    http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
    http://cxf.apache.org/core http://cxf.apache.org/schemas/core.xsd
    http://cxf.apache.org/configuration/security
http://cxf.apache.org/schemas/configuration/security.xsd
    http://cxf.apache.org/transports/http-jetty/configuration
http://cxf.apache.org/schemas/configuration/http-jetty.xsd
    ">
```

NOTE

It is essential to specify the locations of the **http://cxf.apache.org/configuration/security** schema and the **http://cxf.apache.org/transports/http-jetty/configuration** schema in the **xsi:schemaLocation** attribute. These will not automatically be provided by the OSGi container.

Modify proxy address URL to use HTTPS

The proxy endpoint at the start of the Apache Camel route is configured by the **cxf:cxfEndpoint** element in the **camel-config.xml** file. By default, this proxy endpoint is configured to use the HTTP protocol. You must modify the address URL to use the secure HTTPS protocol instead, however. In the **camel-config.xml** file, edit the address attribute of the **cxf:cxfEndpoint** element, replacing the **http:** prefix by the **https:** prefix, as shown in the following fragment:

```
<beans ...>
  ...
  <cxf:cxfEndpoint id="reportIncident"
                address="https://localhost:${proxy.port}/camel-example-cxf-proxy/webservices/incident"
                endpointName="s:ReportIncidentEndpoint"
```

```
                serviceName="s:ReportIncidentEndpointService"
                wsdlURL="etc/report_incident.wsdl"
                xmlns:s="http://reportincident.example.camel.apache.org"/>
  ...
</beans>
```

Notice also that the address URL is configured to use the TCP port, **${proxy.port}** (which has the value **9080** by default). This TCP port value is the same as the value set for the Jetty container (configured by the **http:engine-factory** element), thus ensuring that this endpoint is deployed into the Jetty container. The attributes of the **cxf:cxfEndpoint** specify the WSDL addressing details as described in the section called "WSDL addressing details":

serviceName

Specifies the WSDL service name.

endpointName

Specifies the WSDL port name.

address

Specifies the address URL of the proxy Web service.

5.3. DEPLOYING THE APACHE CAMEL ROUTE

Overview

The Maven POM file in the basic Camel CXF proxy demonstration is already configured to generate an OSGi bundle. Hence, after building the demonstration using Maven, the demonstration bundle (which contains the Apache Camel route and the **RealWebServicesBean** bean) is ready for deployment into the OSGi container.

Prerequisites

Before deploying the Apache Camel route into the OSGi container, you must configure the proxy Web service to use SSL/TLS security, as described in the previous section, Section 5.2, "Securing the Web Services Proxy".

Steps to deploy the Camel route

To deploy the Web services proxy demonstration into the OSGi container, perform the following steps:

1. the section called "Build the demonstration".
2. the section called "Start the OSGi container".
3. the section called "Install the required features".
4. the section called "Deploy the bundle".
5. the section called "Check the console output".

Build the demonstration

Use Maven to build and install the demonstration as an OSGi bundle. Open a command prompt, switch the current directory to ***CamelInstallDir*/examples/camel-example-cxf-proxy**, and enter the following command:

```
mvn install -Dmaven.test.skip=true
```

Start the OSGi container

If you have not already done so, start up the Karaf console (and container instance) by entering the following command in a new command prompt:

```
./fuse
```

Install the required features

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

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

You also need the **camel-http** feature, which defines the bundles required for the Camel/HTTP component. To install the **camel-http** feature, enter the following console command:

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

Deploy the bundle

Deploy the **camel-example-cxf-proxy** bundle, by entering the following console command:

```
JBossFuse:karaf@root> install -s mvn:org.apache.camel/camel-example-cxf-proxy/2.21.0.fuse-740039-redhat-00001
```

NOTE

In this case, it is preferable to deploy the bundle directly using **install**, rather than using hot deploy, so that you can see the bundle output on the console screen.

If you have any difficulty using the **mvn** URL handler, see olink:ESBOSGiGuide/UrlHandlers-Maven for details of how to set it up.

Check the console output

After the bundle is successfully deployed, you should see output like the following in the console window:

```
JBossFuse:karaf@root> Starting real web service...
Started real web service at: http://localhost:9081/real-webservice
```

5.4. SECURING THE WEB SERVICES CLIENT

Overview

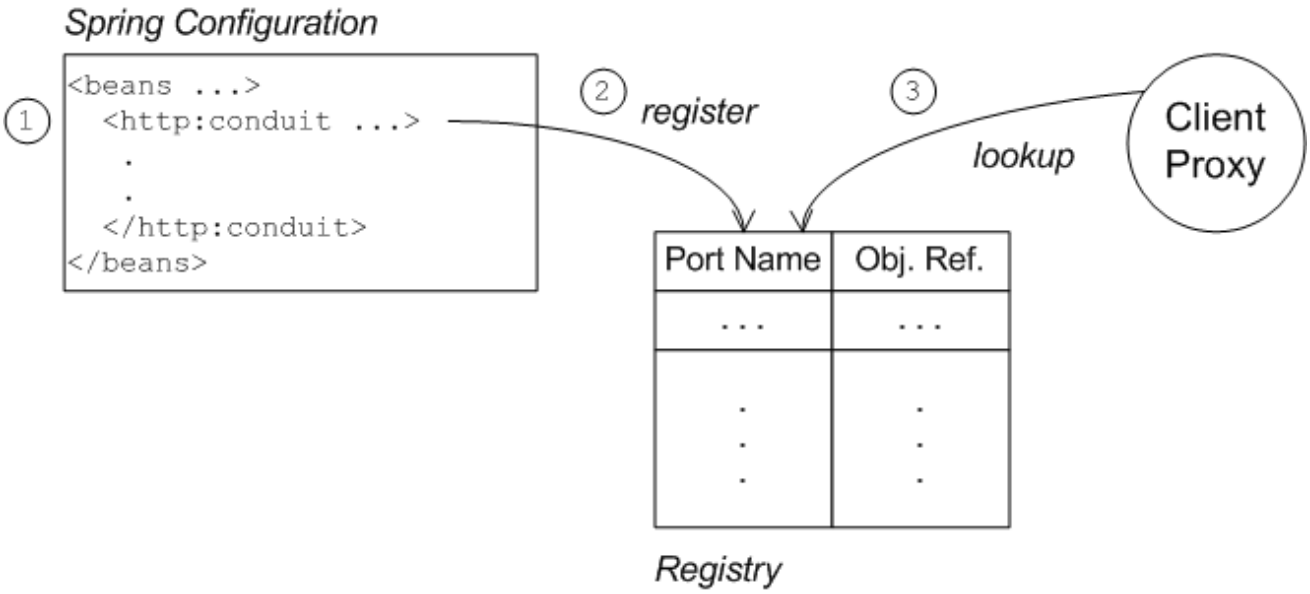
In the basic Camel CXF proxy demonstration, the Web services client is actually implemented as a JUnit test under the **src/test** directory. This means that the client can easily be run using the Maven command, **mvn test**. To enable SSL/TLS security on the client, the Java implementation of the test client is completely replaced and a Spring file, containing the SSL/TLS configuration, is added to the **src/test/resources/META-INF/spring** directory. Before describing the steps you need to perform to set up the client, this section explains some details of the client's Java code and Spring configuration.

Implicit configuration

Apart from changing the URL scheme on the endpoint address to **https:**, most of the configuration to enable SSL/TLS security on a client proxy is contained in a **http:conduit** element in Spring configuration. The way in which this configuration is applied to the client proxy, however, is potentially confusing, for the following reason: the **http:conduit** element **does not explicitly reference the client proxy** and the client proxy **does not explicitly reference thehttp:conduit element**. The connection between the **http:conduit** element and the client proxy is established implicitly, in that they both reference the same WSDL port, as illustrated by Client Proxy Implicitly Configured by http:conduit.

Client Proxy Implicitly Configured by http:conduit

Element



The connection between the client proxy and the **http:conduit** element is established as follows:

1. The client loads and parses the Spring configuration file containing the **http:conduit** element.
2. When the **http:conduit** bean is created, a corresponding entry is created in the registry, which stores a reference to the bean under the specified WSDL port name (where the name is stored in QName format).
3. When the JAX-WS client proxy is created, it scans the registry to see if it can find a **http:conduit** bean associated with the proxy's WSDL port name. If it finds such a bean, it automatically injects the configuration details into the proxy.

Certificates needed on the client side

The client is configured with the following **clientKeystore.jks** keystore file from the **src/main/resources/certs** directory. This keystore contains two entries, as follows:

Trusted cert entry

A trusted certificate entry containing the CA certificate that issued and signed both the server certificate and the client certificate.

Private key entry

A private key entry containing the client's own X.509 certificate and private key. In fact, this certificate is not strictly necessary to run the current example, because the server does not require the client to send a certificate during the TLS handshake (see Example 5.2, "httpj:engine-factory Element with SSL/TLS Enabled").

Loading Spring definitions into the client

The example client is not deployed directly into a Spring container, but it requires some Spring definitions in order to define a secure HTTP conduit. So how can you create the Spring definitions without a Spring container? It turns out that it is easy to read Spring definitions into a Java-based client using the **org.apache.cxf.bus.spring.SpringBusFactory** class.

The following code shows how to read Spring definitions from the file, **META-INF/spring/cxf-client.xml**, and create an Apache CXF Bus object that incorporates those definitions:

```
// Java
import org.apache.cxf.bus.spring.SpringBusFactory;
...
protected void startCxfBus() throws Exception {
    bf = new SpringBusFactory();
    Bus bus = bf.createBus("META-INF/spring/cxf-client.xml");
    bf.setDefaultBus(bus);
}
```

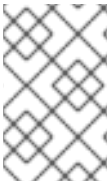
Creating the client proxy

In principle, there are several different ways of creating a WSDL proxy: you could use the JAX-WS API to create a proxy based on the contents of a WSDL file; you could use the JAX-WS API to create a proxy **without** a WSDL file; or you could use the Apache CXF-specific class, **JaxWsProxyFactoryBean**, to create a proxy.

For this SSL/TLS client, the most convenient approach is to use the JAX-WS API to create a proxy without using a WSDL file, as shown in the following Java sample:

```
// Java
import javax.xml.ws.Service;
import org.apache.camel.example.reportincident.ReportIncidentEndpoint;
...
// create the webservice client and send the request
Service s = Service.create(SERVICE_NAME);
s.addPort(
    PORT_NAME,
    "http://schemas.xmlsoap.org/soap/",
    ADDRESS_URL
  );
ReportIncidentEndpoint client =
  s.getPort(PORT_NAME, ReportIncidentEndpoint.class);
```

-



NOTE

In this example, you **cannot** use the **JaxWsProxyFactoryBean** approach to create a proxy, because a proxy created in this way fails to find the HTTP conduit settings specified in the Spring configuration file.

The **SERVICE_NAME** and **PORT_NAME** constants are the QNames of the WSDL service and the WSDL port respectively, as defined in Example 5.1, "The ReportIncidentEndpointService WSDL Service". The **ADDRESS_URL** string has the same value as the proxy Web service address and is defined as follows:

```
private static final String ADDRESS_URL =
  "https://localhost:9080/camel-example-cxf-proxy/webservices/incident";
```

In particular, note that the address **must** be defined with the URL scheme, **https**, which selects HTTP over SSL/TLS.

Steps to add SSL/TLS security to the client

To define a JAX-WS client with SSL/TLS security enabled, perform the following steps:

1. the section called "Create the Java client as a test case".
2. the section called "Add the http:conduit element to Spring configuration".
3. the section called "Run the client".

Create the Java client as a test case

Example 5.3, "ReportIncidentRoutesTest Java client" shows the complete code for a Java client that is implemented as a JUnit test case. This client replaces the existing test, **ReportIncidentRoutesTest.java**, in the **src/test/java/org/apache/camel/example/reportincident** sub-directory of the **examples/camel-example-cxf-proxy** demonstration.

To add the client to the ***CamelInstallDir*/examples/camel-example-cxf-proxy** demonstration, go to the **src/test/java/org/apache/camel/example/reportincident** sub-directory, move the existing **ReportIncidentRoutesTest.java** file to a backup location, then create a new **ReportIncidentRoutesTest.java** file and paste the code from Example 5.3, "ReportIncidentRoutesTest Java client" into this file.

Example 5.3. ReportIncidentRoutesTest Java client

```
// Java
package org.apache.camel.example.reportincident;

import org.apache.camel.spring.Main;
import org.apache.cxf.jaxws.JaxWsProxyFactoryBean;
import org.junit.Test;

import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
```

```
import org.apache.cxf.Bus;
import org.apache.cxf.bus.spring.SpringBusFactory;
import org.apache.camel.example.reportincident.ReportIncidentEndpoint;
import org.apache.camel.example.reportincident.ReportIncidentEndpointService;

import static org.junit.Assert.assertEquals;

/**
 * Unit test of our routes
 */
public class ReportIncidentRoutesTest {

    private static final QName SERVICE_NAME
        = new QName("http://reportincident.example.camel.apache.org",
"ReportIncidentEndpointService");

    private static final QName PORT_NAME =
        new QName("http://reportincident.example.camel.apache.org", "ReportIncidentEndpoint");

    private static final String WSDL_URL = "file:src/main/resources/etc/report_incident.wsdl";

    // should be the same address as we have in our route
    private static final String ADDRESS_URL = "https://localhost:9080/camel-example-cxf-
proxy/webservices/incident";

    protected SpringBusFactory bf;

    protected void startCxfBus() throws Exception {
        bf = new SpringBusFactory();
        Bus bus = bf.createBus("META-INF/spring/cxf-client.xml");
        bf.setDefaultBus(bus);
    }

    @Test
    public void testRendportIncident() throws Exception {
        startCxfBus();
        runTest();
    }

    protected void runTest() throws Exception {

        // create input parameter
        InputReportIncident input = new InputReportIncident();
        input.setIncidentId("123");
        input.setIncidentDate("2008-08-18");
        input.setGivenName("Claus");
        input.setFamilyName("Ibsen");
        input.setSummary("Bla");
        input.setDetails("Bla bla");
        input.setEmail("davsclaus@apache.org");
        input.setPhone("0045 2962 7576");

        // create the webservice client and send the request
        Service s = Service.create(SERVICE_NAME);
        s.addPort(PORT_NAME, "http://schemas.xmlsoap.org/soap/", ADDRESS_URL);
        ReportIncidentEndpoint client = s.getPort(PORT_NAME, ReportIncidentEndpoint.class);
```

```
        OutputReportIncident out = client.reportIncident(input);

        // assert we got a OK back
        assertEquals("OK;456", out.getCode());
    }
}
```

Add the http:conduit element to Spring configuration

Example 5.4, "http:conduit Element with SSL/TLS Enabled" shows the Spring configuration that defines a **http:conduit** element for the **ReportIncidentEndpoint** WSDL port. The **http:conduit** element is configured to enable SSL/TLS security for any client proxies that use the specified WSDL port.

To add the Spring configuration to the client test case, create the **src/test/resources/META-INF/spring** sub-directory, use your favorite text editor to create the file, **cxf-client.xml**, and then paste the contents of Example 5.4, "http:conduit Element with SSL/TLS Enabled" into the file.

Example 5.4. http:conduit Element with SSL/TLS Enabled

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:cxf="http://camel.apache.org/schema/cxf"
      xmlns:sec="http://cxf.apache.org/configuration/security"
      xmlns:http="http://cxf.apache.org/transports/http/configuration"
      xsi:schemaLocation="
      http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
      http://camel.apache.org/schema/cxf http://camel.apache.org/schema/cxf/camel-cxf.xsd
      http://cxf.apache.org/configuration/security
http://cxf.apache.org/schemas/configuration/security.xsd
      http://cxf.apache.org/transports/http/configuration
http://cxf.apache.org/schemas/configuration/http-conf.xsd
      ">

  <http:conduit name="
{http://reportincident.example.camel.apache.org}ReportIncidentEndpoint.http-conduit">
    <http:tlsClientParameters disableCNCheck="true" secureSocketProtocol="TLSv1">
      <sec:keyManagers keyPassword="ckpass">
          <sec:keyStore password="cspass" type="JKS"
          resource="certs/clientKeystore.jks" />
      </sec:keyManagers>
      <sec:trustManagers>
          <sec:keyStore password="cspass" type="JKS"
          resource="certs/clientKeystore.jks" />
      </sec:trustManagers>
      <sec:cipherSuitesFilter>
        <sec:include>.*_WITH_3DES_.*</sec:include>
        <sec:include>.*_WITH_DES_.*</sec:include>
        <sec:exclude>.*WITH_NULL.</sec:exclude>*
        <sec:exclude>.*DH_anon.</sec:exclude>*
      </sec:cipherSuitesFilter>
    </http:tlsClientParameters>
```

```
    </http:conduit>

  </beans>
```

Please note the following points about the preceding configuration:

- The **http:** and **sec:** namespace prefixes are needed to define the **http:conduit** element. In the **xsi:schemaLocation** element, it is also essential to specify the locations of the corresponding **http://cxf.apache.org/configuration/security** and **http://cxf.apache.org/transports/http/configuration** namespaces.
- The **disableCNCheck** attribute of the **http:tlsClientParameters** element is set to **true**. This means that the client does **not** check whether the Common Name in the server's X.509 certificate matches the server hostname. For more details, see Appendix A, *Managing Certificates*.

  IMPORTANT

  Disabling the CN check is **not** recommended in a production deployment.

- In the **sec:keystore** elements, the certificate locations are specified using the **resource** attribute, which finds the certificates on the classpath. When Maven runs the test, it automatically makes the contents of **src/main/resources** available on the classpath, so that the certificates can be read from the **src/main/resources/certs** directory.

  NOTE

  You also have the option of specifying a certificate location using the **file** attribute, which looks in the filesystem. But the **resource** attribute is more suitable for use with applications packaged in bundles.

- The **sec:cipherSuitesFilter** element is configured to exclude cipher suites matching ***.*WITH_NULL.**** and ***.*DH_anon.****. These cipher suites are effectively incomplete and are **not** intended for normal use.

  IMPORTANT

  It is recommended that you always **exclude** the ciphers matching ***.*WITH_NULL.**** and ***.*DH_anon.****.

- The **secureSocketProtocol** attribute should be set to TLSv1, to match the server protocol and to ensure that the SSLv3 protocol is not used (POODLE security vulnerability (CVE-2014-3566)).

Run the client

Because the client is defined as a test case, you can run the client using the standard Maven test goal. To run the client, open a new command window, change directory to ***CamelInstallDir*/examples/camel-example-cxf-proxy**, and enter the following Maven command:

```
mvn test
```

If the test runs successfully, you should see the following output in the OSGi console window:

```
Incident was 123, changed to 456

Invoked real web service: id=456 by Claus Ibsen
```

CHAPTER 6. SECURING THE FUSE CONSOLE

See the Managing Fuse guide for information on how to implement the following security features to secure Fuse containers for standalone deployments:

- Set HTTPS as the required protocol
- Use public keys to secure responses
- Enable SSL/TLS security
- Control user access

By default, the Fuse Console is not accessible remotely. For information on how to access the Fuse Console remotely, see the unlocking the Fuse Console section in the Managing Fuse guide.

CHAPTER 7. INTEGRATION WITH RED HAT SINGLE SIGN-ON

Red Hat Single Sign-On (RH-SSO) option works works with JAAS to provide enterprise security for certain Web client applications and services running inside Fuse and Fuse administration services (SSH, JMX, and Fuse Management Console).

Adapters are provided for the following types of container in Red Hat Fuse:

- Section 7.1, "Adapter for Spring Boot container"
- Section 7.2, "Adapter for Apache Karaf container"
- Section 7.3, "Adapter for JBoss EAP container"

7.1. ADAPTER FOR SPRING BOOT CONTAINER

The adapter for the Spring Boot container supports the following embedded Web containers:

- Undertow
- Jetty
- Tomcat

For details on installing and using the Red Hat Single Sign-On adapter for the Spring Boot container, see Spring Boot Adapter in the Red Hat Single Sign-On *Securing Applications and Services Guide*.

7.2. ADAPTER FOR APACHE KARAF CONTAINER

The adapter for the Apache Karaf container can secure the following components:

- Classic WAR applications deployed on Fuse with Pax Web War Extender.
- Servlets deployed on Fuse as OSGI services with Pax Web Whiteboard Extender and additionally servlets registered through **org.osgi.service.http.HttpService#registerServlet()`** which is a standard OSGi Enterprise HTTP Service.
- Apache Camel Undertow endpoints running with the Camel Undertow component.
- Apache CXF endpoints running on their own separate Undertow engine.
- Apache CXF endpoints running on the default engine provided by the CXF servlet.
- SSH and JMX admin access.
- Hawtio administration console.

For details on installing and using the Red Hat Single Sign-On adapter for the Apache Karaf container, see JBoss Fuse 7 Adapter in the Red Hat Single Sign-On *Securing Applications and Services Guide*.

7.3. ADAPTER FOR JBOSS EAP CONTAINER

The adapter for the JBoss Enterprise Application Platform (EAP) container provides security for WARs, enabling you to define role-based security constraints on your URLs.

For details on installing and using the Red Hat Single Sign-On adapter for the JBoss EAP container, see JBoss EAP Adapter in the Red Hat Single Sign-On *Securing Applications and Services Guide*.

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 389 Directory Server
- Add user entries to the LDAP server
- Add groups to manage security roles
- Configure Fuse to use LDAP authentication
- Configure Fuse to use roles for authorization
- Configure SSL/TLS connections to the LDAP server

8.2. SET-UP A DIRECTORY SERVER AND CONSOLE

This stage of the tutorial explains how to install the X.500 directory server and the management console from the Fedora 389 Directory Server project. If you already have access to a 389 Directory Server instance, you can skip the instructions for installing the 389 Directory Server and install the 389 Management Console instead.

Prerequisites

If you are installing on a Red Hat Enterprise Linux platform, you must first install the Extra Packages for Enterprise Linux (EPEL). See the installation notes under RHEL/Cent OS/ EPEL (RHEL 6, RHEL 7, Cent OS 6, Cent OSý7) on the **fedoraproject.org** site.

Install 389 Directory Server

If you do not have access to an existing **389 Directory Server** instance, you can install **389 Directory Server** on your local machine, as follows:

1. On Red Hat Enterprise Linux and Fedora platforms, use the standard **dnf** package management utility to install **389 Directory Server**. Enter the following command at a command prompt (you must have administrator privileges on your machine):

```
sudo dnf install 389-ds
```

NOTE

The required **389-ds** and **389-console** RPM packages are available for Fedora, RHEL6+EPEL, and CentOS7+EPEL platforms. At the time of writing, the **389-console** package is not yet available for RHEL 7.

2. After installing the 389 directory server packages, enter the following command to configure the directory server:

```
sudo setup-ds-admin.pl
```

   The script is interactive and prompts you to provide the basic configuration settings for the 389 directory server. When the script is complete, it automatically launches the 389 directory server in the background.

3. For more details about how to install **389 Directory Server**, see the Download page.

Install 389 Management Console

If you already have access to a **389 Directory Server** instance, you only need to install the 389 Management Console, which enables you to log in and manage the server remotely. You can install the 389 Management Console, as follows:

- **On Red Hat Enterprise Linux and Fedora platforms**–use the standard **dnf** package management utility to install the 389 Management Console. Enter the following command at a command prompt (you must have administrator privileges on your machine):

```
sudo dnf install 389-console
```

- **On Windows platforms**–see the Windows Console download instructions from **fedoraproject.org**.

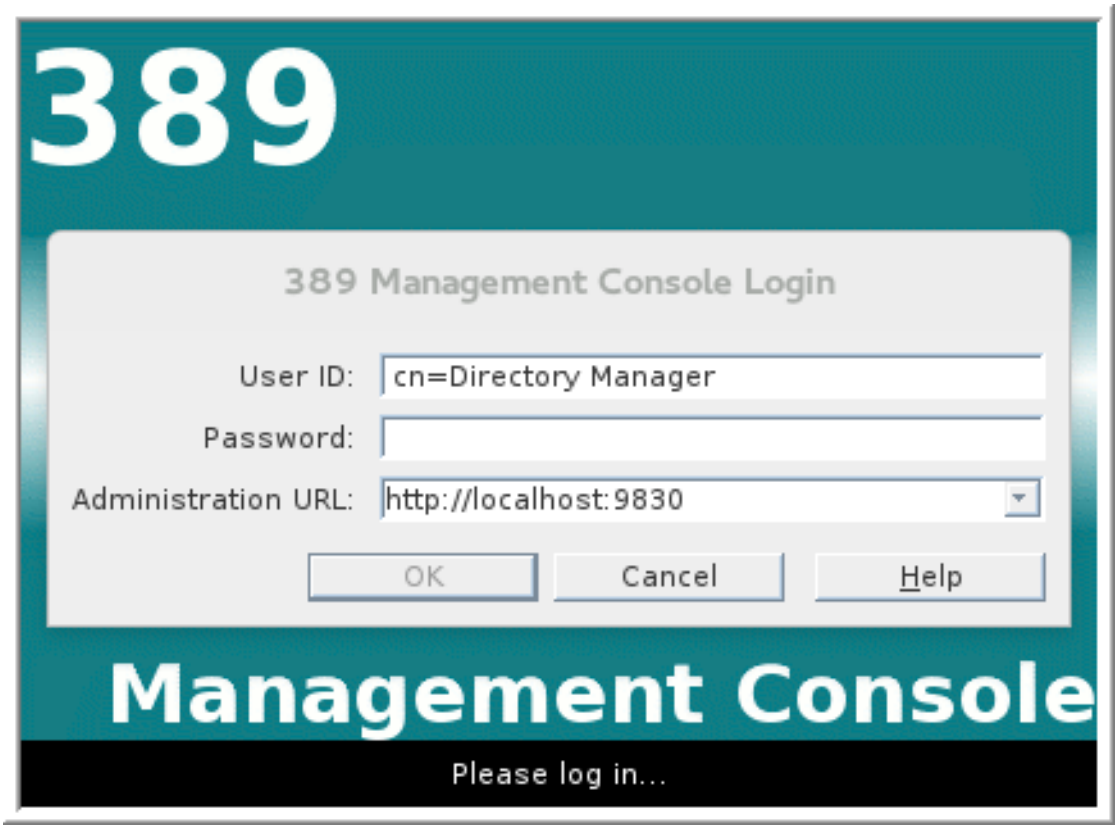
Connect the console to the server

To connect the 389 Directory Server Console to the LDAP server:

1. Enter the following command to start up the 389 Management Console:

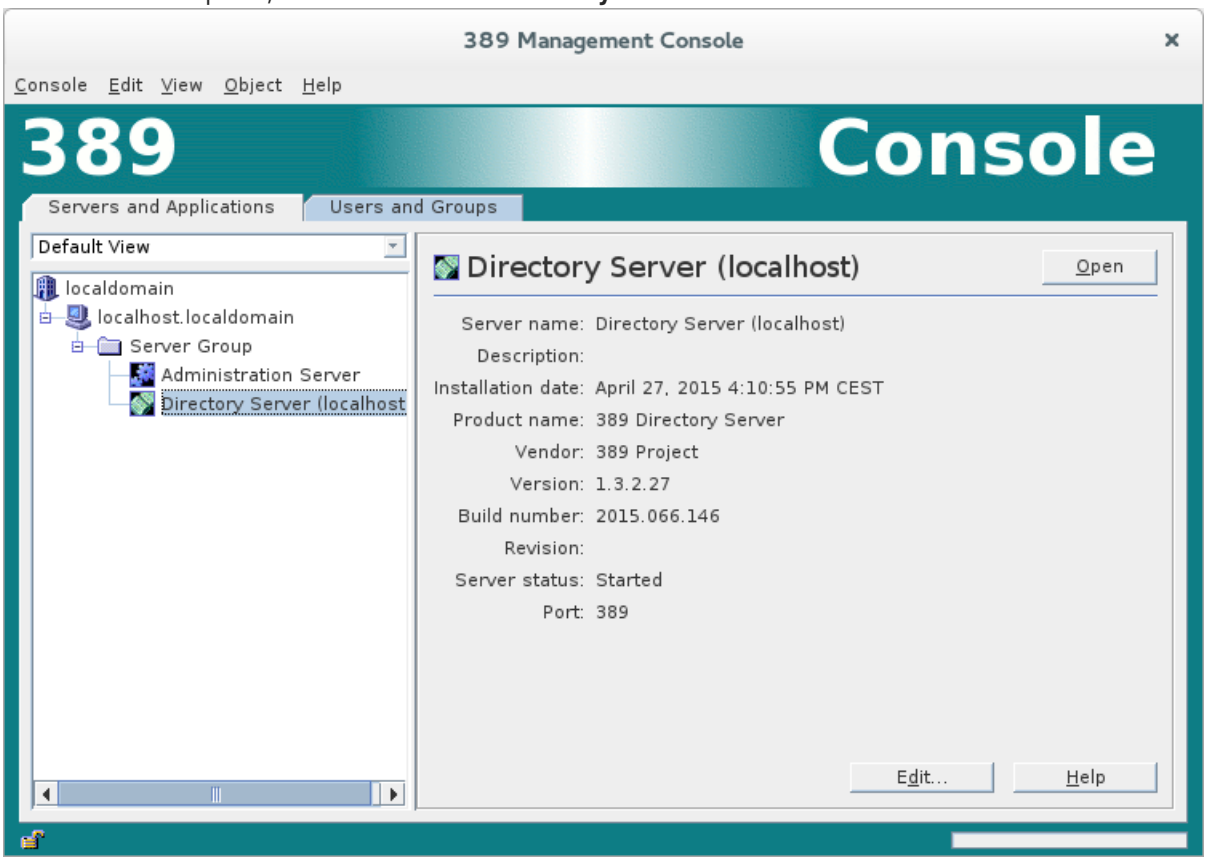
```
389-console
```

2. A login dialog appears. Fill in the LDAP login credentials in the **User ID** and **Password** fields, and customize the hostname in the **Administration URL** field to connect to your 389 management server instance (port **9830** is the default port for the 389 management server instance).



3. The 389 Management Console window appears. Select the **Servers and Applications** tab.

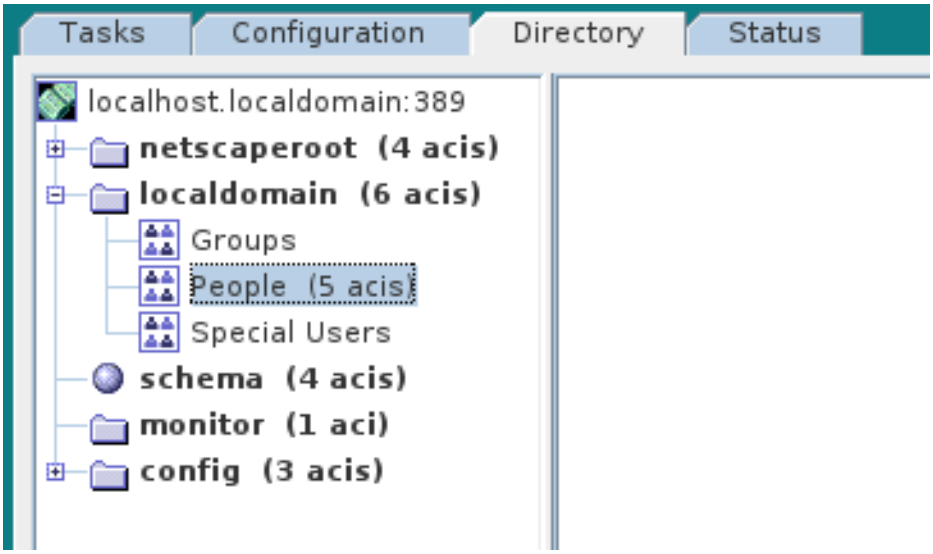
4. In the left-hand pane, drill down to the **Directory Server** icon.



5. Select the **Directory Server** icon in the left-hand pane and click **Open**, to open the **389 Directory Server Console**.

6. In the **389 Directory Server Console**, click the **Directory** tab, to view the Directory Information Tree (DIT).

7. Expand the root node, ***YourDomain*** (usually named after a hostname, and shown as **localdomain** in the following screenshot), to view the DIT.



8.3. ADD USER ENTRIES TO THE DIRECTORY SERVER

The basic prerequisite for using LDAP authentication with the OSGi container is to have an X.500 directory server running and configured with a collection of user entries. For many use cases, you will also want to configure a number of groups to manage user roles.

Alternative to adding user entries

If you already have user entries and groups defined in your LDAP server, you might prefer to map the existing LDAP groups to JAAS roles using the **roles.mapping** property in the **LDAPLoginModule** configuration, instead of creating new entries. For details, see Section 2.1.7, "JAAS LDAP Login Module".

Goals

In this portion of the tutorial you will

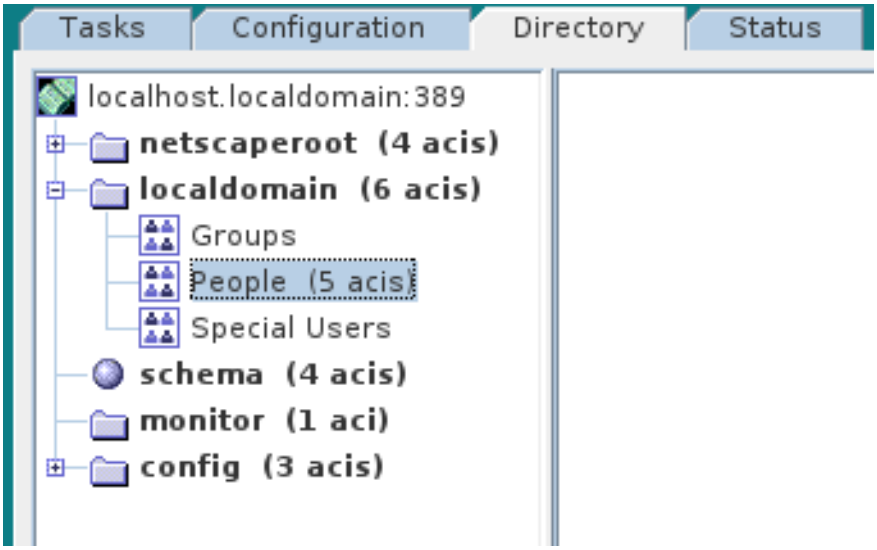
- Add three user entries to the LDAP server
- Add four groups to the LDAP server

Adding user entries

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

1. Ensure that the LDAP server and console are running. See Section 8.2, "Set-up a Directory Server and Console".

2. In the **Directory Server Console**, click on the **Directory** tab, and drill down to the **People** node, under the ***YourDomain*** node (where ***YourDomain*** is shown as **localdomain** in the following screenshots).



3. Right-click the **People** node, and select menu:[> New > > **User** >] from the context menu, to open the **Create New User** dialog.

4. Select the **User** tab in the left-hand pane of the **Create New User** dialog.

5. Fill in the fields of the **User** tab, as follows:

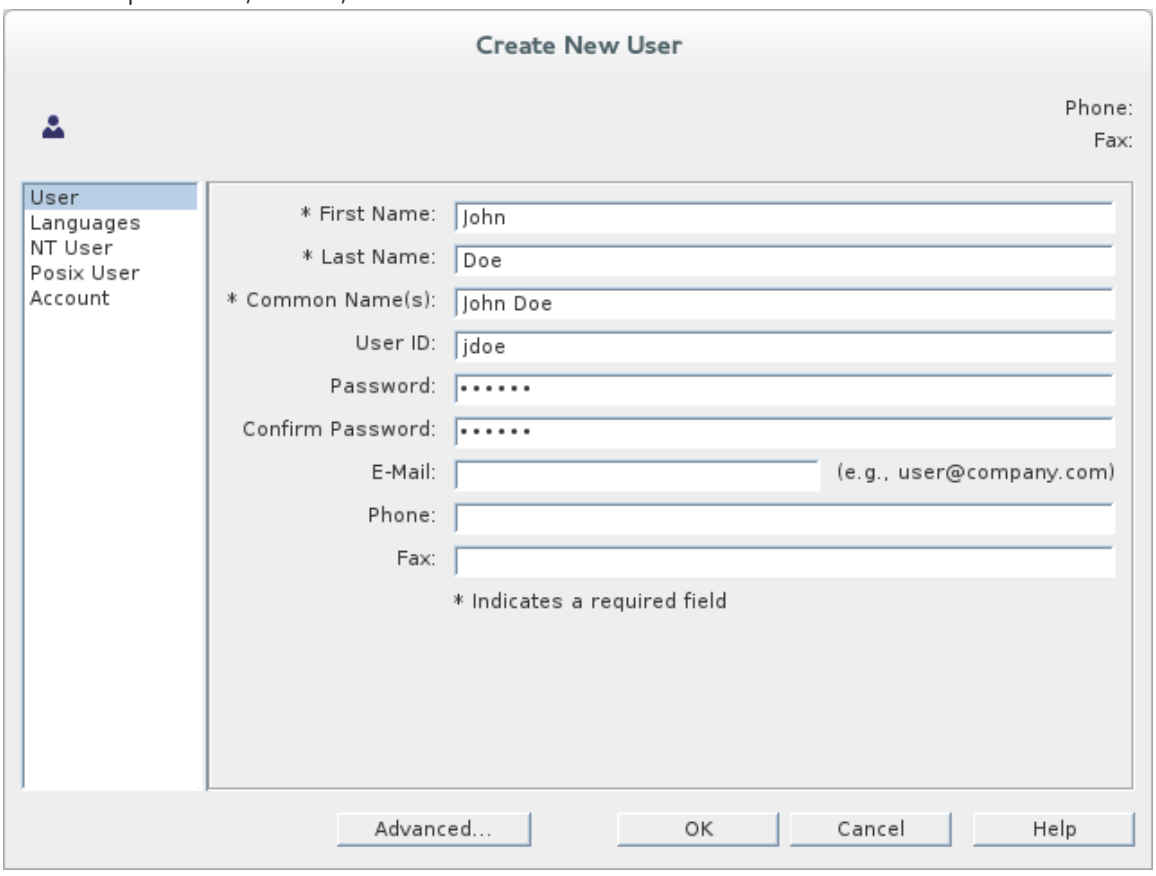
   a. Set the **First Name** field to **John**.

   b. Set the **Last Name** field to **Doe**.

   c. Set the **User ID** field to **jdoe**.

   d. Enter the password, **secret**, in the **Password** field.

   e. Enter the password, **secret**, in the **Confirm Password** field.



6. Click **OK**.

7. Add a user **Jane Doe** by following Step 3 to Step 6.
   In Step 5.e, use **janedoe** for the new user's **User ID** and use the password, **secret**, for the password fields.

8. Add a user **Camel Rider** by following Step 3 to Step 6.
   In Step 5.e, use **crider** for the new user's **User ID** and use the password, **secret**, for the password fields.

Adding groups for the roles

To add the groups that define the roles:

1. In the **Directory** tab of the **Directory Server Console**, drill down to the **Groups** node, under the ***YourDomain*** node.

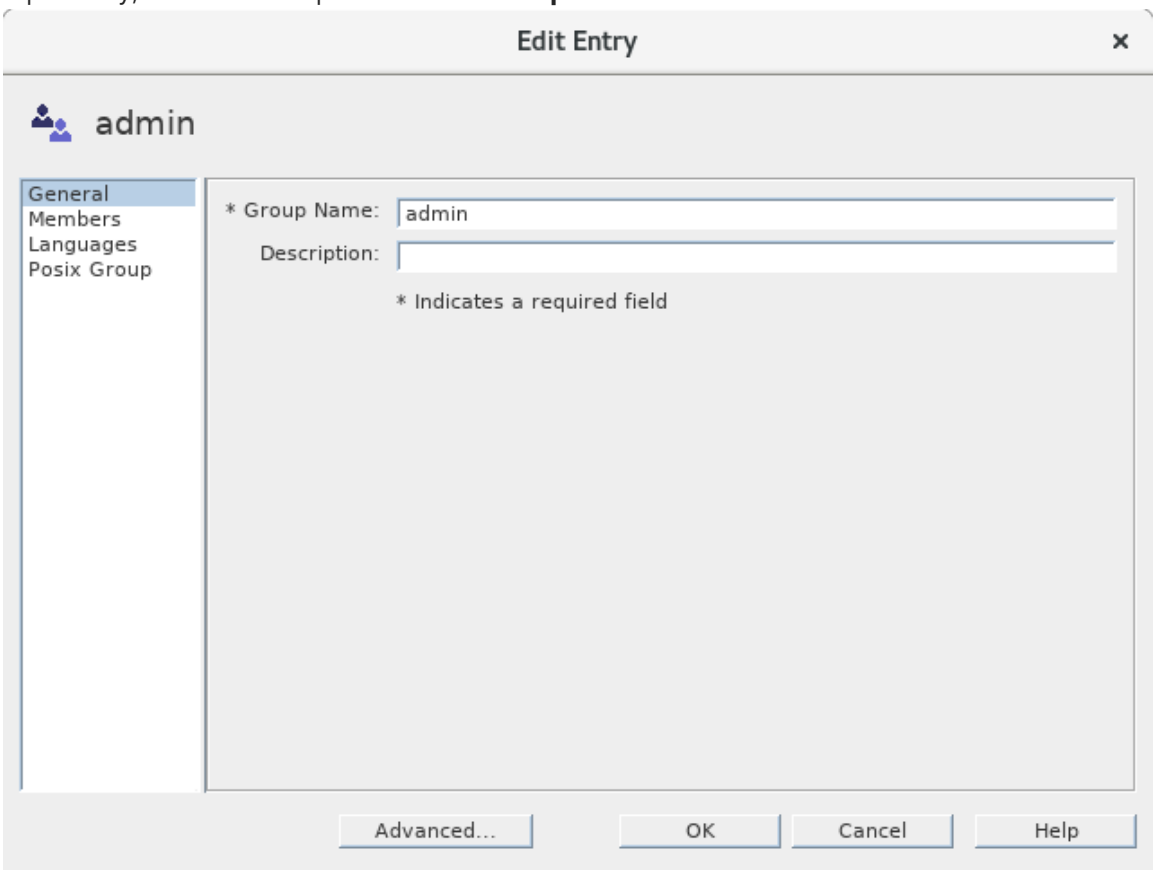
2. Right-click the **Groups** node, and select menu:[> New > > **Group** >] from the context menu, to open the **Create New Group** dialog.

3. Select the **General** tab in the left-hand pane of the **Create New Group** dialog.

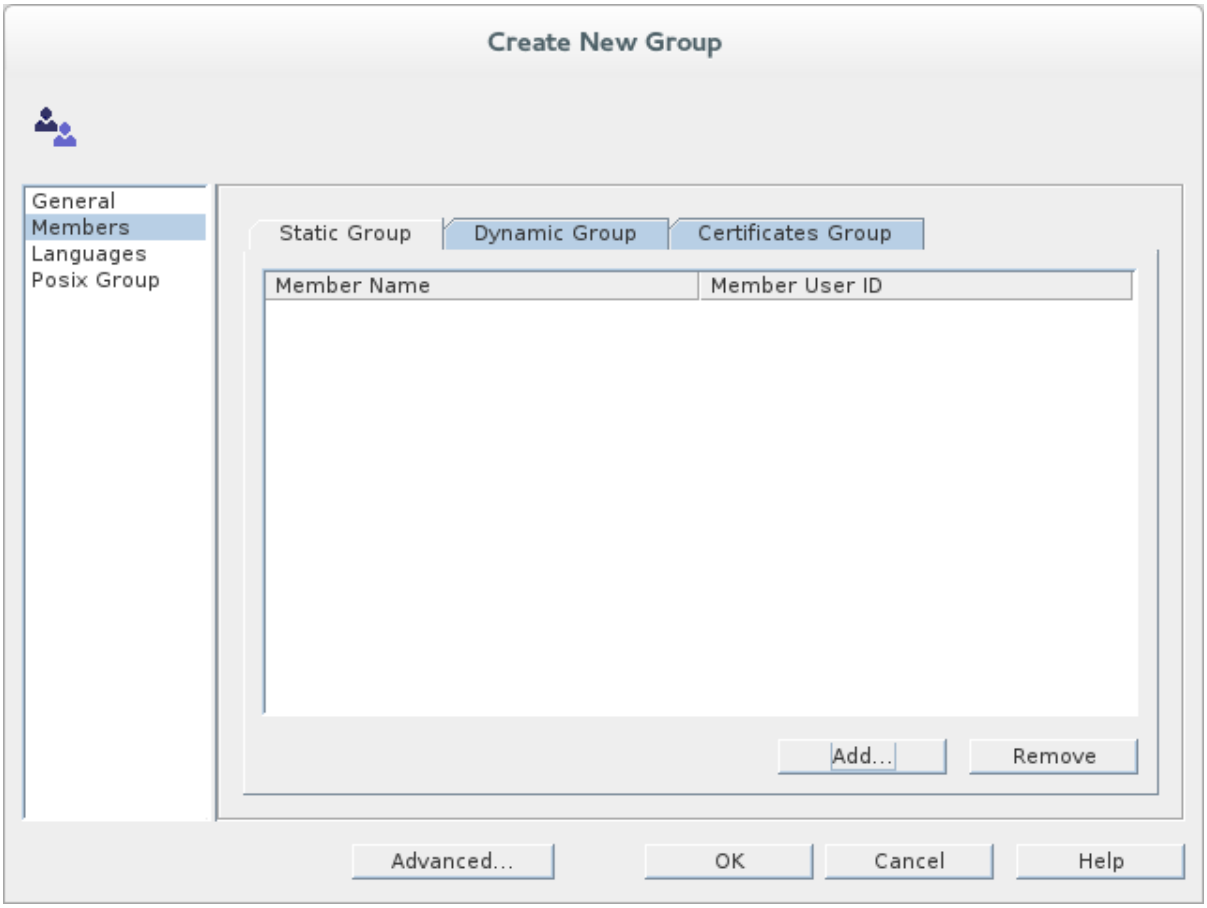
4. Fill in the fields of the **General** tab, as follows:

   a. Set the **Group Name** field to **admin**.

   b. Optionally, enter a description in the **Description** field.

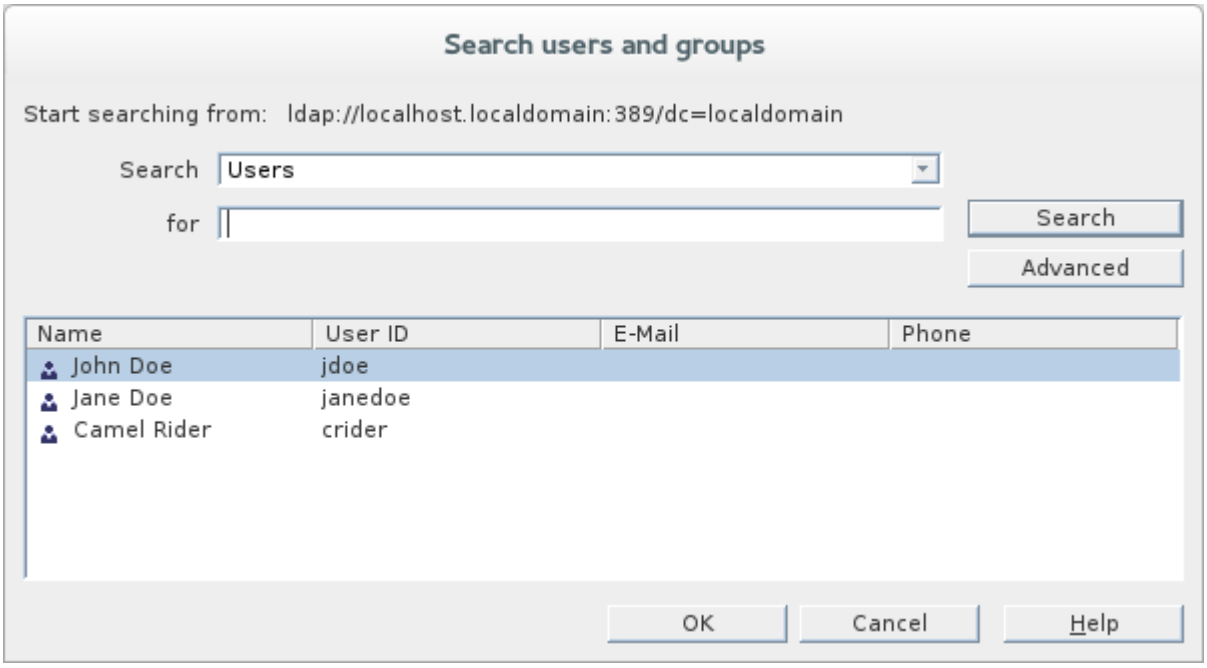


5. Select the **Members** tab in the left-hand pane of the **Create New Group** dialog.



6. Click **Add** to open the **Search users and groups** dialog.

7. In the **Search** field, select **Users** from the drop-down menu, and click the **Search** button.



8. From the list of users that is now displayed, select **John Doe**.

9. Click **OK**, to close the **Search users and groups** dialog.

10. Click **OK**, to close the **Create New Group** dialog.

11. Add a **manager** role by following Step 2 to Step 10.
    In Step 4, enter **manager** in the **Group Name** field.

In Step 8, select **Jane Doe**.

12. Add a **viewer** role by following Step 2 to Step 10.
    In Step 4, enter **viewer** in the **Group Name** field.

    In Step 8, select **Camel Rider**.

13. Add an **ssh** role by following Step 2 to Step 10.
    In Step 4, enter **ssh** in the **Group Name** field.

    In Step 8, select *all* of the users, **John Doe**, **Jane Doe**, and **Camel Rider**.

8.4. ENABLE LDAP AUTHENTICATION IN THE OSGI CONTAINER

This section explains how to configure an LDAP realm in the OSGi container. The new realm overrides the default **karaf** realm, so that the container authenticates credentials based on user entries stored in the X.500 directory server.

References

More detailed documentation is available on LDAP authentication, as follows:

- **LDAPLoginModule options**—are described in detail in Section 2.1.7, "JAAS LDAP Login Module".
- **Configurations for other directory servers**—this tutorial covers only 389-DS. For details of how to configure other directory servers, such as Microsoft Active Directory, see the section called "Filter settings for different directory servers".

Procedure for standalone OSGi container

To enable LDAP authentication in a standalone OSGi container:

1. Ensure that the X.500 directory server is running.
2. Start the Karaf container by entering the following command in a terminal window:

```
./bin/fuse
```

3. Create a 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="2.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0">

  <jaas:config name="karaf" rank="200">
    <jaas:module className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule"
                 flags="required">
      initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
      connection.url=ldap://localhost:389
```

```
    connection.username=cn=Directory Manager
    connection.password=DIRECTORY_MANAGER_PASSWORD
    connection.protocol=
    user.base.dn=ou=People,dc=localdomain
    user.filter=(&amp;(objectClass=inetOrgPerson)(uid=%u))
    user.search.subtree=true
    role.base.dn=ou=Groups,dc=localdomain
    role.name.attribute=cn
    role.filter=(uniquemember=%fqdn)
    role.search.subtree=true
    authentication=simple
  </jaas:module>
 </jaas:config>
</blueprint>
```

You must customize the following settings in the **ldap-module.xml** file:

connection.url

Set this URL to the actual location of your directory server instance. Normally, this URL has the format, **ldap://*Hostname*:*Port***. For example, the default port for the 389 Directory Server is IP port **389**.

connection.username

Specifies the username that is used to authenticate the connection to the directory server. For 389 Directory Server, the default is usually **cn=Directory Manager**.

connection.password

Specifies the password part of the credentials for connecting to the directory server.

authentication

You can specify either of the following alternatives for the authentication protocol:

- **simple** implies that user credentials are supplied and you are obliged to set the **connection.username** and **connection.password** options in this case.

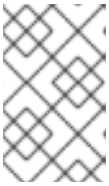
- **none** implies that authentication is **not** performed. You must not set the **connection.username** and **connection.password** options in this case.
  This login module creates a JAAS realm called **karaf**, which is the same name as the default JAAS realm used by Fuse. By redefining this realm with a **rank** attribute value greater than **0**, it overrides the standard **karaf** realm which has the rank **0**.

  For more details about how to configure Fuse to use LDAP, see Section 2.1.7, "JAAS LDAP Login Module".

IMPORTANT

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

5. To deploy the new LDAP module, copy the **ldap-module.xml** into the Karaf container's **deploy**/ directory (hot deploy).
   The LDAP module is automatically activated.



NOTE

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

Test the LDAP authentication

Test the new LDAP realm by connecting to the running container using the Karaf **client** utility, as follows:

1. Open a new command prompt.
2. Change directory to the Karaf ***InstallDir***/**bin** directory.
3. 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. At the command console, type **jaas:** followed by the [Tab] key (to activate content completion):

   ```
   jdoe@root()> jaas:
   Display all 31 possibilities? (31 lines)?
   jaas:cancel
   jaas:group-add
   ...
   jaas:whoami
   ```

   You should see that **jdoe** has access to all of the **jaas** commands (consistent with the **admin** role).
4. Log off the remote console by entering the **logout** command.
5. 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. At the command console, type **jaas:** followed by the [Tab] key (to activate content completion):

   ```
   janedoe@root()> jaas:
   Display all 25 possibilities? (25 lines)?
   jaas:cancel
   jaas:group-add
   ...
   jaas:users
   ```

   You should see that **janedoe** has access to almost all of the **jaas** commands (consistent with the **manager** role).
6. Log off the remote console by entering the **logout** command.

7. Enter the following command to log on to the running container instance using the identity **crider**:

```
./client -u crider -p secret
```

   You should successfully log into the container's remote console. At the command console, type **jaas:** followed by the [Tab] key (to activate content completion):

```
crider@root()> jaas:
jaas:manage
jaas:realm-list
jaas:realm-manage
jaas:realms
jaas:user-list
jaas:users
```

   You should see that **crider** has access to only five of the **jaas** commands (consistent with the **viewer** role).

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

Troubleshooting

If you run into any difficulties while testing the LDAP connection, increase the logging level to **DEBUG** to get a detailed trace of what is happening on the connection to the LDAP server.

Perform the following steps:

1. From the Karaf console, enter the following command to increase the logging level to **DEBUG**:

```
log:set DEBUG
```

2. Observe the Karaf log in real time:

```
log:tail
```

   To escape from the log listing, type Ctrl-C.

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 Fuse applications.

A.1. WHAT IS AN X.509 CERTIFICATE?

Role of certificates

An X.509 certificate binds a name to a public key value. The role of the certificate is to associate a public key with the identity contained in the X.509 certificate.

Integrity of the public key

Authentication of a secure application depends on the integrity of the public key value in the application's certificate. If an impostor replaces the public key with its own public key, it can impersonate the true application and gain access to secure data.

To prevent this type of attack, all certificates must be signed by a *certification authority* (CA). A CA is a trusted node that confirms the integrity of the public key value in a certificate.

Digital signatures

A CA signs a certificate by adding its *digital signature* to the certificate. A digital signature is a message encoded with the CA's private key. The CA's public key is made available to applications by distributing a certificate for the CA. Applications verify that certificates are validly signed by decoding the CA's digital signature with the CA's public key.

WARNING

The supplied demonstration certificates are self-signed certificates. These certificates are insecure because anyone can access their private key. To secure your system, you must create new certificates signed by a trusted CA.

Contents of an X.509 certificate

An X.509 certificate contains information about the certificate subject and the certificate issuer (the CA that issued the certificate). A certificate is encoded in Abstract Syntax Notation One (ASN.1), a standard syntax for describing messages that can be sent or received on a network.

The role of a certificate is to associate an identity with a public key value. In more detail, a certificate includes:

- A *subject distinguished name (DN)* that identifies the certificate owner.
- The *public key* associated with the subject.

- X.509 version information.
- A *serial number* that uniquely identifies the certificate.
- An *issuer DN* that identifies the CA that issued the certificate.
- The digital signature of the issuer.
- Information about the algorithm used to sign the certificate.
- Some optional X.509 v.3 extensions; for example, an extension exists that distinguishes between CA certificates and end-entity certificates.

Distinguished names

A DN is a general purpose X.500 identifier that is often used in the context of security.

See Appendix B, *ASN.1 and Distinguished Names* for more details about DNs.

A.2. CERTIFICATION AUTHORITIES

A.2.1. Introduction to Certificate Authorities

A CA consists of a set of tools for generating and managing certificates and a database that contains all of the generated certificates. When setting up a system, it is important to choose a suitable CA that is sufficiently secure for your requirements.

There are two types of CA you can use:

- commercial CAs are companies that sign certificates for many systems.
- private CAs are trusted nodes that you set up and use to sign certificates for your system only.

A.2.2. Commercial Certification Authorities

Signing certificates

There are several commercial CAs available. The mechanism for signing a certificate using a commercial CA depends on which CA you choose.

Advantages of commercial CAs

An advantage of commercial CAs is that they are often trusted by a large number of people. If your applications are designed to be available to systems external to your organization, use a commercial CA to sign your certificates. If your applications are for use within an internal network, a private CA might be appropriate.

Criteria for choosing a CA

Before choosing a commercial CA, consider the following criteria:

- What are the certificate-signing policies of the commercial CAs?
- Are your applications designed to be available on an internal network only?

- What are the potential costs of setting up a private CA compared to the costs of subscribing to a commercial CA?

A.2.3. Private Certification Authorities

Choosing a CA software package

If you want to take responsibility for signing certificates for your system, set up a private CA. To set up a private CA, you require access to a software package that provides utilities for creating and signing certificates. Several packages of this type are available.

OpenSSL software package

One software package that allows you to set up a private CA is OpenSSL, http://www.openssl.org. The OpenSSL package includes basic command line utilities for generating and signing certificates. Complete documentation for the OpenSSL command line utilities is available at http://www.openssl.org/docs.

Setting up a private CA using OpenSSL

To set up a private CA, see the instructions in Section A.5, "Creating Your Own Certificates" .

Choosing a host for a private certification authority

Choosing a host is an important step in setting up a private CA. The level of security associated with the CA host determines the level of trust associated with certificates signed by the CA.

If you are setting up a CA for use in the development and testing of Red Hat Fuse 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.

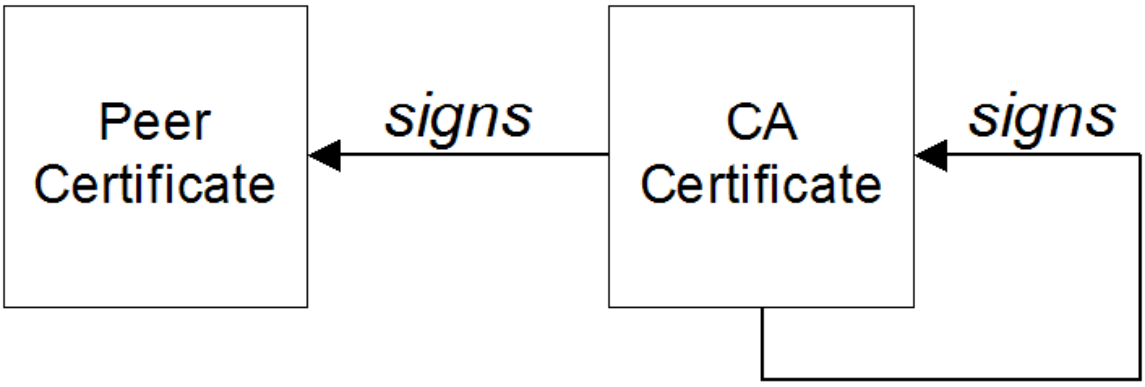
A.3. CERTIFICATE CHAINING

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

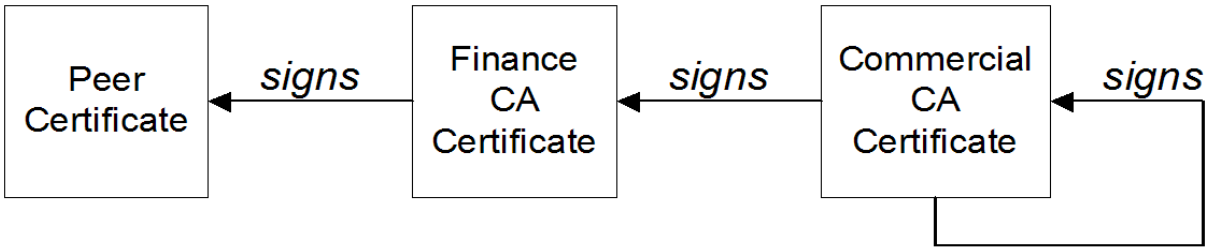
The purpose of a certificate chain is to establish a chain of trust from a peer certificate to a trusted CA certificate. The CA vouches for the identity in the peer certificate by signing it. If the CA is one that you trust (indicated by the presence of a copy of the CA certificate in your root certificate 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=RedHat,
OU=System,CN=www.redhat.com
```

Where the CN has been set to the host name, **www.redhat.com**.

For details of how to set the subject DN in a new certificate, see Section A.5, "Creating Your Own Certificates".

Using subjectAltName (multi-homed hosts)

Using the subject DN's Common Name for the certificate identity has the disadvantage that only **one**

host name can be specified at a time. If you deploy a certificate on a multi-homed host, however, you might find it is practical to allow the certificate to be used with **any** of the multi-homed host names. In this case, it is necessary to define a certificate with multiple, alternative identities, and this is only possible using the **subjectAltName** certificate extension.

For example, if you have a multi-homed host that supports connections to either of the following host names:

```
www.redhat.com
www.jboss.org
```

Then you can define a **subjectAltName** that explicitly lists both of these DNS host names. If you generate your certificates using the **openssl** utility, edit the relevant line of your **openssl.cnf** configuration file to specify the value of the **subjectAltName** extension, as follows:

```
subjectAltName=DNS:www.redhat.com,DNS:www.jboss.org
```

Where the HTTPS protocol matches the server host name against either of the DNS host names listed in the **subjectAltName** (the **subjectAltName** takes precedence over the Common Name).

The HTTPS protocol also supports the wildcard character, *, in host names. For example, you can define the **subjectAltName** as follows:

```
subjectAltName=DNS:*.jboss.org
```

This certificate identity matches any three-component host name in the domain **jboss.org**.

WARNING

You must **never** use the wildcard character in the domain name (and you must take care never to do this accidentally by forgetting to type the dot, **.**, delimiter in front of the domain name). For example, if you specified ***jboss.org**, your certificate could be used on *any* domain that ends in the letters **jboss**.

A.5. CREATING YOUR OWN CERTIFICATES

Abstract

This chapter describes the techniques and procedures to set up your own private Certificate Authority (CA) and to use this CA to generate and sign your own certificates.

WARNING

Creating and managing your own certificates requires an expert knowledge of security. While the procedures described in this chapter can be convenient for generating your own certificates for demonstration and testing environments, it is **not recommended** to use these certificates in a production environment.

A.5.1. Install the OpenSSL Utilities

Installing OpenSSL on RHEL and Fedora platforms

On Red Hat Enterprise Linux (RHEL) 5 and 6 and Fedora platforms, are made available as an RPM package. To install OpenSSL, enter the following command (executed with administrator privileges):

```
yum install openssl
```

Source code distribution

The source distribution of OpenSSL is available from http://www.openssl.org/docs. The OpenSSL project provides source code distributions **only**. You cannot download a binary install of the OpenSSL utilities from the OpenSSL Web site.

A.5.2. Set Up a Private Certificate Authority

Overview

If you choose to use a private CA you need to generate your own certificates for your applications to use. The OpenSSL project provides free command-line utilities for setting up a private CA, creating signed certificates, and adding the CA to your Java keystore.

WARNING

Setting up a private CA for a production environment requires a high level of expertise and extra care must be taken to protect the certificate store from external threats.

Steps to set up a private Certificate Authority

To set up your own private Certificate Authority:

1. Create the directory structure for the CA, as follows:

```
X509CA/demoCA
X509CA/demoCA/private
```

X509CA/demoCA/certs
X509CA/demoCA/newcerts
X509CA/demoCA/crl

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

Example A.1. OpenSSL Configuration

```
#
# 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 -keyalg RSA -dname "CN=Alice, OU=Engineering, O=Red Hat,
   ST=Dublin, C=IE" -validity 365 -alias alice -keypass KeyPass -keystore alice.ks -storepass
   StorePass
   ```

   Because the specified keystore, **alice.ks**, did not exist prior to issuing the command implicitly creates a new keystore and sets its password to ***StorePass***.

   The **-dname** and **-validity** flags define the contents of the newly created X.509 certificate.

   > **NOTE**
   >
   > When specifying the certificate's Distinguished Name (through the **-dname** parameter), you must be sure to observe any policy constraints specified in the **openssl.cfg** file. If those policy constraints are not heeded, you will not be able to sign the certificate using the CA (in the next steps).

   > **NOTE**
   >
   > It is essential to generate the key pair with the **-keyalg RSA** option (or a key algorithm of similar strength). The default key algorithm uses a combination of DSA encryption and SHA-1 signature. But the SHA-1 algorithm is no longer regarded as sufficiently secure and modern Web browsers will reject certificates signed using SHA-1. When you select the RSA key algorithm, the **keytool** utility uses an SHA-2 algorithm instead.

2. Create a certificate signing request using the **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 the section called "Steps to set up a private Certificate Authority").

   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
OU	organizationalUnitName	1...64	2.5.4.11

String Representation	X.500 Attribute Type	Size of Data	Equivalent OID
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
```