# Red Hat JBoss Data Grid 6.5

## Security Guide

For securing Red Hat JBoss Data Grid 6.5.1

Last Updated: 2017-11-13

# Red Hat JBoss Data Grid 6.5 Security Guide

For securing Red Hat JBoss Data Grid 6.5.1

Gemma Sheldon
Red Hat Engineering Content Services
gsheldon@redhat.com

## Legal Notice

## Abstract

This guide presents information about security features and configurations in Red Hat JBoss Data Grid 6.5.1

# Table of Contents

# CHAPTER 1. INTRODUCTION TO SECURITY IN THE DATA GRID

## 1.1. SECURING DATA IN RED HAT JBOSS DATA GRID

In Red Hat JBoss Data Grid, data security can be implemented in the following ways:

**Role-based Access Control**

JBoss Data Grid features role-based access control for operations on designated secured caches. Roles can be assigned to users who access your application, with roles mapped to permissions for cache and cache-manager operations. Only authenticated users are able to perform the operations that are authorized for their role.

In Library mode, data is secured via role-based access control for CacheManagers and Caches, with authentication delegated to the container or application. In Remote Client-Server mode, JBoss Data Grid is secured by passing identity tokens from the Hot Rod client to the server, and role-based access control of Caches and CacheManagers.

**Node Authentication and Authorization**

Node-level security requires new nodes or merging partitions to authenticate before joining a cluster. Only authenticated nodes that are authorized to join the cluster are permitted to do so. This provides data protection by preventing authorized servers from storing your data.

**Encrypted Communications Within the Cluster**

JBoss Data Grid increases data security by supporting encrypted communications between the nodes in a cluster by using a user-specified cryptography algorithm, as supported by Java Cryptography Architecture (JCA).

JBoss Data Grid also provides audit logging for operations, and the ability to encrypt communication between the Hot Rod Client and Server using Transport Layer Security (TLS/SSL).

Report a bug

# CHAPTER 2. RED HAT JBOSS DATA GRID SECURITY: AUTHORIZATION AND AUTHENTICATION

Red Hat JBoss Data Grid is able to perform authorization on CacheManagers and Caches. JBoss Data Grid authorization is built on standard security features available in a JDK, such as JAAS and the SecurityManager.

If an application attempts to interact with a secured CacheManager and Cache, it must provide an identity which JBoss Data Grid's security layer can validate against a set of required roles and permissions. Once validated, the client is issued a token for subsequent operations. Where access is denied, an exception indicating a security violation is thrown.

When a cache has been configured for with authorization, retrieving it returns an instance of `SecureCache`. `SecureCache` is a simple wrapper around a cache, which checks whether the "current user" has the permissions required to perform an operation. The "current user" is a Subject associated with the `AccessControlContext`.

JBoss Data Grid maps Principals names to roles, which in turn, represent one or more permissions. The following diagram represents these relationships:

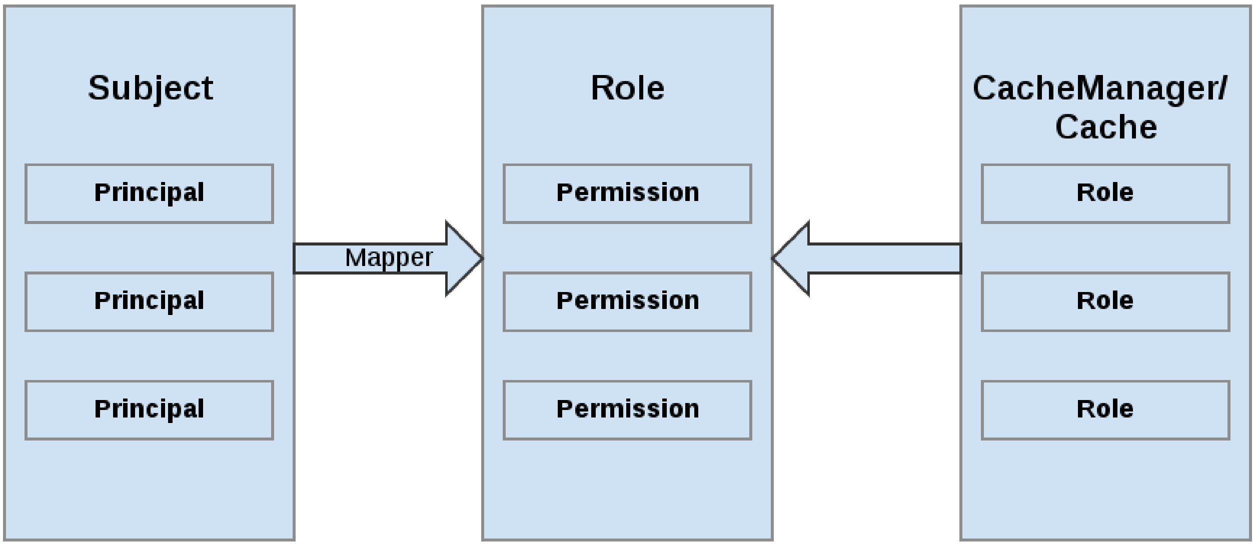

**Figure 2.1. Roles and Permissions Mapping**

Report a bug

## 2.1. PERMISSIONS

Access to a CacheManager or a Cache is controlled using a set of required permissions. Permissions control the type of action that is performed on the CacheManager or Cache, rather than the type of data being manipulated. Some of these permissions can apply to specifically name entities, such as a named cache. Different types of permissions are available depending on the entity.

**Table 2.1. CacheManager Permissions**

| Permission | Function | Description |
|---|---|---|

| Permission | Function | Description |
|---|---|---|
| CONFIGURATION | defineConfiguration | Whether a new cache configuration can be defined. |
| LISTEN | addListener | Whether listeners can be registered against a cache manager. |
| LIFECYCLE | stop, start | Whether the cache manager can be stopped or started respectively. |
| ALL | | A convenience permission which includes all of the above. |

**Table 2.2. Cache Permissions**

| Permission | Function | Description |
|---|---|---|
| READ | get, contains | Whether entries can be retrieved from the cache. |
| WRITE | put, putIfAbsent, replace, remove, evict | Whether data can be written/replaced/removed/evicted from the cache. |
| EXEC | distexec, mapreduce | Whether code execution can be run against the cache. |
| LISTEN | addListener | Whether listeners can be registered against a cache. |
| BULK_READ | keySet, values, entrySet,query | Whether bulk retrieve operations can be executed. |
| BULK_WRITE | clear, putAll | Whether bulk write operations can be executed. |
| LIFECYCLE | start, stop | Whether a cache can be started / stopped. |

| Permission | Function | Description |
|---|---|---|
| ADMIN | getVersion, addInterceptor*, removeInterceptor, getInterceptorChain, getEvictionManager, getComponentRegistry, getDistributionManager, getAuthorizationManager, evict, getRpcManager, getCacheConfiguration, getCacheManager, getInvocationContextContainer, setAvailability, getDataContainer, getStats, getXAResource | Whether access to the underlying components/internal structures is allowed. |
| ALL | | A convenience permission which includes all of the above. |
| ALL_READ | | Combines READ and BULK_READ. |
| ALL_WRITE | | Combines WRITE and BULK_WRITE. |

### NOTE

Some permissions may need to be combined with others in order to be useful. For example, EXEC with READ or with WRITE.

Report a bug

## 2.2. ROLE MAPPING

In order to convert the Principals in a Subject into a set of roles used for authorization, a `PrincipalRoleMapper` must be specified in the global configuration. Red Hat JBoss Data Grid ships with three mappers, and also allows you to provide a custom mapper.

**Table 2.3. Mappers**

| Mapper Name | Java | XML | Description |
|---|---|---|---|
| IdentityRoleMapper | org.infinispan.security.impl.IdentityRoleMapper | <identity-role-mapper /> | Uses the Principal name as the role name. |

| Mapper Name | Java | XML | Description |
|---|---|---|---|
| CommonNameRoleMapper | org.infinispan.security.impl.CommonRoleMapper | <common-name-role-mapper /> | If the Principal name is a Distinguished Name (DN), this mapper extracts the Common Name (CN) and uses it as a role name. For example the DN **cn=managers,ou=people,dc=example,dc=com** will be mapped to the role **managers**. |
| ClusterRoleMapper | org.infinispan.security.impl.ClusterRoleMapper | <cluster-role-mapper /> | Uses the **ClusterRegistry** to store principal to role mappings. This allows the use of the CLI's **GRANT** and **DENY** commands to add/remove roles to a Principal. |
| Custom Role Mapper | | <custom-role-mapper class="a.b.c" /> | Supply the fully-qualified class name of an implementation of **org.infinispan.security.impl.PrincipalRoleMapper** |

Report a bug

## 2.3. CONFIGURING AUTHENTICATION AND ROLE MAPPING USING JBOSS EAP LOGIN MODULES

When using Red Hat JBoss EAP log in module for querying roles from LDAP, you must implement your own mapping of Principals to Roles, as JBoss EAP uses its own custom classes. The following example demonstrates how to map a principal obtained from JBoss EAP login module to a role. It maps user principal name to a role, performing a similar action to the **IdentityRoleMapper**:

**Example 2.1. Mapping a Principal from JBoss EAP's Login Module**

```
public class SimplePrincipalGroupRoleMapper implements
PrincipalRoleMapper {
   @Override
   public Set<String> principalToRoles(Principal principal) {
      if (principal instanceof SimpleGroup) {
         Enumeration<Principal> members = ((SimpleGroup)
principal).members();
         if (members.hasMoreElements()) {
            Set<String> roles = new HashSet<String>();
            while (members.hasMoreElements()) {
               Principal innerPrincipal = members.nextElement();
               if (innerPrincipal instanceof SimplePrincipal) {
                  SimplePrincipal sp = (SimplePrincipal)
innerPrincipal;
                  roles.add(sp.getName());
               }
            }
            return roles;
         }
      }
      return null;
   }
}
```

**Example 2.2. Example of JBoss EAP LDAP login module configuration**

```
 <security-domain name="ispn-secure" cache-type="default">
                  <authentication>
                     <login-module
code="org.jboss.security.auth.spi.LdapLoginModule" flag="required">
                        <module-option
name="java.naming.factory.initial"
value="com.sun.jndi.ldap.LdapCtxFactory"/>
                        <module-option name="java.naming.provider.url"
value="ldap://localhost:389"/>
                        <module-option
name="java.naming.security.authentication" value="simple"/>
                        <module-option name="principalDNPrefix"
value="uid="/>
                        <module-option name="principalDNSuffix"
value=",ou=People,dc=infinispan,dc=org"/>
                        <module-option name="rolesCtxDN"
value="ou=Roles,dc=infinispan,dc=org"/>
                        <module-option name="uidAttributeID"
value="member"/>
                        <module-option name="matchOnUserDN"
value="true"/>
                        <module-option name="roleAttributeID"
value="cn"/>
                        <module-option name="roleAttributeIsDN"
value="false"/>
                        <module-option name="searchScope"
value="ONELEVEL_SCOPE"/>
                     </login-module>
                  </authentication>
                </security-domain>
```

**Example 2.3. Example of JBoss EAP Login Module Configuration**

```
<security-domain name="krb-admin" cache-type="default">
```

```
                    <authentication>
                        <login-module code="Kerberos" flag="required">
                            <module-option name="useKeyTab"
value="true"/>
                            <module-option name="principal"
value="admin@INFINISPAN.ORG"/>
                            <module-option name="keyTab"
value="${basedir}/keytab/admin.keytab"/>
                        </login-module>
                    </authentication>
                </security-domain>
```

When using GSSAPI authentication, this would typically involve using LDAP for role mapping, with JBoss EAP server authenticating itself to the LDAP server via GSSAPI. For more information on how to configure this, see the *JBoss EAP Administration and Configuration Guide*

**IMPORTANT**

For information about how to configure JBoss EAP login modules, see the *JBoss EAP Administration and Configuration Guide* and see the Red Hat Directory Server *Administration Guide* how to configure LDAP server, and specify users and their role mapping.

Report a bug

## 2.4. CONFIGURING RED HAT JBOSS DATA GRID FOR AUTHORIZATION

Authorization is configured at two levels: the cache container (CacheManager), and at the single cache.

### CacheManager

The following is an example configuration for authorization at the CacheManager level:

**Example 2.4. CacheManager Authorization (Declarative Configuration)**

```
<cache-container name="local" default-cache="default">
        <security>
           <authorization>
             <identity-role-mapper />
             <role name="admin" permissions="ALL"/>
             <role name="reader" permissions="READ"/>
             <role name="writer" permissions="WRITE"/>
             <role name="supervisor" permissions="ALL_READ ALL_WRITE"/>
           </authorization>
        </security>
</cache-container>
```

Each cache container determines:

- whether to use authorization.

- a class which will map principals to a set of roles.
- a set of named roles and the permissions they represent.

You can choose to use only a subset of the roles defined at the container level.

### Roles

Roles may be applied on a cache-per-cache basis, using the roles defined at the cache-container level, as follows:

**Example 2.5. Defining Roles**

```
<local-cache name="secured">
  <security>
    <authorization roles="admin reader writer supervisor"/>
  </security>
</local-cache>
```

**IMPORTANT**

Any cache that is intended to require authentication must have a listing of roles defined; otherwise authentication is not enforced as the no-anonymous policy is defined by the cache's authorization.

### Programmatic CacheManager Authorization (Library Mode)

The following example shows how to set up the same authorization parameters for Library mode using programmatic configuration:

**Example 2.6. CacheManager Authorization Programmatic Configuration**

```
GlobalConfigurationBuilder global = new GlobalConfigurationBuilder();
  global
     .security()
        .authorization()
           .principalRoleMapper(new IdentityRoleMapper())
           .role("admin")
              .permission(CachePermission.ALL)
           .role("supervisor")
              .permission(CachePermission.EXEC)
              .permission(CachePermission.READ)
              .permission(CachePermission.WRITE)
           .role("reader")
              .permission(CachePermission.READ);
  ConfigurationBuilder config = new ConfigurationBuilder();
  config
     .security()
        .enable()
        .authorization()
           .role("admin")
           .role("supervisor")
           .role("reader");
```

Report a bug

## 2.5. DATA SECURITY FOR LIBRARY MODE

### 2.5.1. Subject and Principal Classes

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

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

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

**`getPrincipals()`** returns all principals contained in the subject. **`getPrincipals(Class c)`** returns only those principals that are instances of class **`c`** or one of its subclasses. An empty set is returned if the subject has no matching principals.

**NOTE**

The **`java.security.acl.Group`** interface is a sub-interface of **`java.security.Principal`**, so an instance in the principals set may represent a logical grouping of other principals or groups of principals.

Report a bug

### 2.5.2. Obtaining a Subject

In order to use a secured cache in Library mode, you must obtain a **`javax.security.auth.Subject`**. The Subject represents information for a single cache entity, such as a person or a service.

Red Hat JBoss Data Grid allows a JAAS Subject to be obtained either by using your container's features, or by using a third-party library.

In JBoss containers, this can be done using the following:

```
Subject subject = SecurityContextAssociation.getSubject();
```

The Subject must be populated with a set of Principals, which represent the user and groups it belongs to in your security domain, for example, an LDAP or Active Directory.

The Java EE API allows retrieval of a container-set Principal through the following methods:

- Servlets: **`ServletRequest.getUserPrincipal()`**

- EJBs: `EJBContext.getCallerPrincipal()`
- MessageDrivenBeans: `MessageDrivenContext.getCallerPrincipal()`

The `mapper` is then used to identify the principals associated with the Subject and convert them into roles that correspond to those you have defined at the container level.

A Principal is only one of the components of a Subject, which is retrieved from the `java.security.AccessControlContext`. Either the container sets the Subject on the `AccessControlContext`, or the user must map the Principal to an appropriate Subject before wrapping the call to the JBoss Data Grid API using a `Security.doAs()` method.

Once a Subject has been obtained, the cache can be interacted with in the context of a PrivilegedAction.

**Example 2.7. Obtaining a Subject**

```
import org.infinispan.security.Security;

Security.doAs(subject, new PrivilegedExceptionAction<Void>() {
public Void run() throws Exception {
    cache.put("key", "value");
}
});
```

The `Security.doAs()` method is in place of the typical Subject.doAs() method. Unless the `AccessControlContext` must be modified for reasons specific to your application's security model, using `Security.doAs()` provides a performance advantage.

To obtain the current Subject, use `Security.getSubject();`, which will retrieve the Subject from either the JBoss Data Grid context, or from the `AccessControlContext`.

Report a bug

### 2.5.3. Subject Authentication

Subject Authentication requires a JAAS login. The login process consists of the following points:

1. An application instantiates a `LoginContext` and passes in the name of the login configuration and a `CallbackHandler` to populate the `Callback` objects, as required by the configuration `LoginModules`.
2. The `LoginContext` consults a `Configuration` to load all the `LoginModules` included in the named login configuration. If no such named configuration exists the `other` configuration is used as a default.
3. The application invokes the `LoginContext.login` method.
4. The login method invokes all the loaded `LoginModules`. As each `LoginModule` attempts to authenticate the subject, it invokes the handle method on the associated `CallbackHandler` to obtain the information required for the authentication process. The required information is passed to the handle method in the form of an array of `Callback` objects. Upon success, the `LoginModules` associate relevant principals and credentials with the subject.

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

6. If authentication succeeds, the application retrieves the authenticated subject using the **LoginContext.getSubject** method.

7. After the scope of the subject authentication is complete, all principals and related information associated with the subject by the **login** method can be removed by invoking the **LoginContext.logout** method.

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

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

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

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

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

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

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

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

- The **`LoginContext`** creates each configured **`LoginModule`** using its public no-arg constructor.

- Each **`LoginModule`** is initialized with a call to its initialize method. The **`Subject`** argument is guaranteed to be non-null. The signature of the initialize method is: **`public void initialize(Subject subject, CallbackHandler callbackHandler, Map sharedState, Map options)`**

- The **`login`** method is called to start the authentication process. For example, a method implementation might prompt the user for a user name and password and then verify the information against data stored in a naming service such as NIS or LDAP. Alternative implementations might interface to smart cards and biometric devices, or simply extract user information from the underlying operating system. The validation of user identity by each **`LoginModule`** is considered phase 1 of JAAS authentication. The signature of the **`login`** method is **`boolean login() throws LoginException`** . A **`LoginException`** indicates failure. A return value of true indicates that the method succeeded, whereas a return value of false indicates that the login module should be ignored.

- If the **`LoginContext`**'s overall authentication succeeds, **`commit`** is invoked on each **`LoginModule`**. If phase 1 succeeds for a **`LoginModule`**, then the commit method continues with phase 2 and associates the relevant principals, public credentials, and/or private credentials with the subject. If phase 1 fails for a **`LoginModule`**, then **`commit`** removes any previously stored authentication state, such as user names or passwords. The signature of the **`commit`** method is: **`boolean commit() throws LoginException`** . Failure to complete the commit phase is indicated by throwing a **`LoginException`**. A return of true indicates that the method succeeded, whereas a return of false indicates that the login module should be ignored.

- If the **`LoginContext`**'s overall authentication fails, then the **`abort`** method is invoked on each **`LoginModule`**. The **`abort`** method removes or destroys any authentication state created by the login or initialize methods. The signature of the **`abort`** method is **`boolean abort() throws LoginException`** . Failure to complete the **`abort`** phase is indicated by throwing a **`LoginException`**. A return of true indicates that the method succeeded, whereas a return of false indicates that the login module should be ignored.

- To remove the authentication state after a successful login, the application invokes **`logout`** on the **`LoginContext`**. This in turn results in a **`logout`** method invocation on each

**LoginModule**. The **logout** method removes the principals and credentials originally associated with the subject during the **commit** operation. Credentials should be destroyed upon removal. The signature of the **logout** method is: **boolean logout() throws LoginException**. Failure to complete the logout process is indicated by throwing a **LoginException**. A return of true indicates that the method succeeded, whereas a return of false indicates that the login module should be ignored.

When a **LoginModule** must communicate with the user to obtain authentication information, it uses a **CallbackHandler** object. Applications implement the CallbackHandler interface and pass it to the **LoginContext**, which send the authentication information directly to the underlying login modules.

Login modules use the **CallbackHandler** both to gather input from users, such as a password or smart card PIN, and to supply information to users, such as status information. By allowing the application to specify the **CallbackHandler**, underlying **LoginModule**s remain independent from the different ways applications interact with users. For example, a **CallbackHandler**'s implementation for a GUI application might display a window to solicit user input. On the other hand, a **CallbackHandler** implementation for a non-GUI environment, such as an application server, might simply obtain credential information by using an application server API. The CallbackHandler interface has one method to implement:

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

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

Report a bug

### 2.5.4. Authorization Using a SecurityManager

In Red Hat JBoss Data Grid's Remote Client-Server mode, authorization is able to work without a SecurityManager for basic cache operations. In Library mode, a SecurityManager may also be used to perform some of the more complex tasks, such as distexec, map/reduce, and query.

In order to enforce access restrictions, enable the SecurityManager in your JVM using one of the following methods:

**Command Line**

```
java -Djava.security.manager ...
```

**Programmaticaly**

```
System.setSecurityManager(new SecurityManager());
```

Using the JDK's default implementation is not required, however an appropriate policy file must be supplied. The JBoss Data Grid distribution includes an example policy file, which demonstrates the permissions required by some of JBoss Data Grid's JAR files. These permissions must be integrated with those required by your application.

Report a bug

## 2.5.5. Security Manager in Java

### 2.5.5.1. About the Java Security Manager

**Java Security Manager**

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

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

Report a bug

### 2.5.5.2. About Java Security Manager Policies

**Security Policy**

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

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

**Basic Information**

A security policy's entry consists of the following configuration elements, which are connected to the **`policytool`**:

**CodeBase**

The URL location (excluding the host and domain information) where the code originates from. This parameter is optional.

**SignedBy**

The alias used in the keystore to reference the signer whose private key was used to sign the code. This can be a single value or a comma-separated list of values. This parameter is optional. If omitted, presence or lack of a signature has no impact on the Java Security Manager.

**Principals**

A list of ***`principal_type/principal_name`*** pairs, which must be present within the executing thread's principal set. The Principals entry is optional. If it is omitted, it signifies that the principals of the executing thread will have no impact on the Java Security Manager.

**Permissions**

A permission is the access which is granted to the code. Many permissions are provided as part of the Java Enterprise Edition 6 (Java EE 6) specification. This document only covers additional permissions which are provided by JBoss EAP 6.

**IMPORTANT**

Refer to your container documentation on how to configure the security policy, as it may differ depending on the implementation.

Report a bug

### 2.5.5.3. Write a Java Security Manager Policy

**Introduction**

An application called `policytool` is included with most JDK and JRE distributions, for the purpose of creating and editing Java Security Manager security policies. Detailed information about `policytool` is linked from http://docs.oracle.com/javase/6/docs/technotes/tools/.

**Procedure 2.1. Setup a new Java Security Manager Policy**

1. **Start `policytool`.**
   Start the `policytool` tool in one of the following ways.

   - **Red Hat Enterprise Linux**
     From your GUI or a command prompt, run `/usr/bin/policytool`.

   - **Microsoft Windows Server**
     Run `policytool.exe` from your Start menu or from the `bin\` of your Java installation. The location can vary.

2. **Create a policy.**
   To create a policy, select **Add Policy Entry**. Add the parameters you need, then click **Done**.

3. **Edit an existing policy**
   Select the policy from the list of existing policies, and select the **Edit Policy Entry** button. Edit the parameters as needed.

4. **Delete an existing policy.**
   Select the policy from the list of existing policies, and select the **Remove Policy Entry** button.

Report a bug

### 2.5.5.4. Run Red Hat JBoss Data Grid Server Within the Java Security Manager

To specify a Java Security Manager policy, you need to edit the Java options passed to the domain or server instance during the bootstrap process. For this reason, you cannot pass the parameters as options to the `domain.sh` or `standalone.sh` scripts. The following procedure guides you through the steps of configuring your instance to run within a Java Security Manager policy.

**Prerequisites**

- Before you following this procedure, you need to write a security policy, using the **`policytool`** command which is included with your Java Development Kit (JDK). This procedure assumes that your policy is located at ***`JDG_HOME`*****`/bin/server.policy`**. As an alternative, write the security policy using any text editor and manually save it as ***`JDG_HOME`*****`/bin/server.policy`**

- The domain or standalone server must be completely stopped before you edit any configuration files.

Perform the following procedure for each physical host or instance in your domain, if you have domain members spread across multiple systems.

**Procedure 2.2. Configure the Security Manager for JBoss Data Grid Server**

1. **Open the configuration file.**
   Open the configuration file for editing. This file is located in one of two places, depending on whether you use a managed domain or standalone server. This is not the executable file used to start the server or domain.

   - **Managed Domain**

     - For Linux: ***`JDG_HOME`*****`/bin/domain.conf`**

     - For Windows: ***`JDG_HOME`*****`\bin\domain.conf.bat`**

   - **Standalone Server**

     - For Linux: ***`JDG_HOME`*****`/bin/standalone.conf`**

     - For Windows: ***`JDG_HOME`*****`\bin\standalone.conf.bat`**

2. **Add the Java options to the file.**
   To ensure the Java options are used, add them to the code block that begins with:

   ```
   if [ "x$JAVA_OPTS" = "x" ]; then
   ```

   You can modify the **`-Djava.security.policy`** value to specify the exact location of your security policy. It should go onto one line only, with no line break. Using == when setting the **`-Djava.security.policy`** property specifies that the security manager will use *only* the specified policy file. Using = specifies that the security manager will use the specified policy *combined with* the policy set in the **`policy.url`** section of ***`JAVA_HOME`*****`/lib/security/java.security`**.

   **IMPORTANT**

   JBoss Enterprise Application Platform releases from 6.2.2 onwards require that the system property **`jboss.modules.policy-permissions`** is set to *true*.

   **Example 2.8. domain.conf**

   ```
   JAVA_OPTS="$JAVA_OPTS -Djava.security.manager -
   Djava.security.policy==$PWD/server.policy -
   ```

```
Djboss.home.dir=/path/to/JDG_HOME -Djboss.modules.policy-
permissions=true"
```

**Example 2.9. domain.conf.bat**

```
set "JAVA_OPTS=%JAVA_OPTS% -Djava.security.manager -
Djava.security.policy==\path\to\server.policy -
Djboss.home.dir=\path\to\JDG_HOME -Djboss.modules.policy-
permissions=true"
```

**Example 2.10. standalone.conf**

```
JAVA_OPTS="$JAVA_OPTS -Djava.security.manager -
Djava.security.policy==$PWD/server.policy -
Djboss.home.dir=$JBOSS_HOME -Djboss.modules.policy-
permissions=true"
```

**Example 2.11. standalone.conf.bat**

```
set "JAVA_OPTS=%JAVA_OPTS% -Djava.security.manager -
Djava.security.policy==\path\to\server.policy -
Djboss.home.dir=%JBOSS_HOME% -Djboss.modules.policy-
permissions=true"
```

3. **Start the domain or server.**
   Start the domain or server as normal.

Report a bug

## 2.6. DATA SECURITY FOR REMOTE CLIENT SERVER MODE

### 2.6.1. About Security Realms

A *security realm* is a series of mappings between users and passwords, and users and roles. Security realms are a mechanism for adding authentication and authorization to your EJB and Web applications. Red Hat JBoss Data Grid Server provides two security realms by default:

- **`ManagementRealm`** stores authentication information for the Management API, which provides the functionality for the Management CLI and web-based Management Console. It provides an authentication system for managing JBoss Data Grid Server itself. You could also use the **`ManagementRealm`** if your application needed to authenticate with the same business rules you use for the Management API.

- **`ApplicationRealm`** stores user, password, and role information for Web Applications and EJBs.

Each realm is stored in two files on the filesystem:

- ***REALM*-users.properties** stores usernames and hashed passwords.
- ***REALM*-roles.properties** stores user-to-role mappings.
- **mgmt-groups.properties** stores user-to-role mapping file for **ManagementRealm**.

The properties files are stored in the **domain/configuration/** and **standalone/configuration/** directories. The files are written simultaneously by the **add-user.sh** or **add-user.bat** command. When you run the command, the first decision you make is which realm to add your new user to.

Report a bug

### 2.6.2. Add a New Security Realm

1. **Run the Management CLI.**
   Start the **cli.sh** or **cli.bat** command and connect to the server.

2. **Create the new security realm itself.**
   Run the following command to create a new security realm named **MyDomainRealm** on a domain controller or a standalone server.

   ```
   /host=master/core-service=management/security-
   realm=MyDomainRealm:add()
   ```

3. **Create the references to the properties file which will store information about the new role.**
   Run the following command to create a pointer a file named **myfile.properties**, which will contain the properties pertaining to the new role.

   **NOTE**

   The newly-created properties file is not managed by the included **add-user.sh** and **add-user.bat** scripts. It must be managed externally.

   ```
   /host=master/core-service=management/security-
   realm=MyDomainRealm/authentication=properties:add(path=myfile.proper
   ties)
   ```

**Result**

Your new security realm is created. When you add users and roles to this new realm, the information will be stored in a separate file from the default security realms. You can manage this new file using your own applications or procedures.

Report a bug

### 2.6.3. Add a User to a Security Realm

1. **Run the add-user.sh or add-user.bat command.**
   Open a terminal and change directories to the ***JDG_HOME*/bin/** directory. If you run Red Hat Enterprise Linux or another UNIX-like operating system, run **add-user.sh**. If you run Microsoft Windows Server, run **add-user.bat**.

2. **Choose whether to add a Management User or Application User.**
   For this procedure, type **b** to add an Application User.

3. **Choose the realm the user will be added to.**
   By default, the only available realm is `ApplicationRealm`. If you have added a custom realm, you can type its name instead.

4. **Type the username, password, and roles, when prompted.**
   Type the desired username, password, and optional roles when prompted. Verify your choice by typing **yes**, or type **no** to cancel the changes. The changes are written to each of the properties files for the security realm.

Report a bug

### 2.6.4. Configuring Security Realms Declaratively

In Remote Client-Server mode, a Hot Rod endpoint must specify a security realm.

The security realm declares an ***authentication*** and an ***authorization*** section.

**Example 2.12. Configuring Security Realms Declaratively**

```
<security-realms>
            <security-realm name="ManagementRealm">
                <authentication>
                    <local default-user="$local" skip-group-
loading="true"/>
                    <properties path="mgmt-users.properties" relative-
to="jboss.server.config.dir"/>
                </authentication>
                <authorization map-groups-to-roles="false">
                    <properties path="mgmt-groups.properties" relative-
to="jboss.server.config.dir"/>
                </authorization>
            </security-realm>
            <security-realm name="ApplicationRealm">
                <authentication>
                    <local default-user="$local" allowed-users="*"
skip-group-loading="true"/>
                    <properties path="application-users.properties"
relative-to="jboss.server.config.dir"/>
                </authentication>
                <authorization>
                    <properties path="application-roles.properties"
relative-to="jboss.server.config.dir"/>
                </authorization>
            </security-realm>
        </security-realms>
```

The ***server-identities*** parameter can also be used to specify certificates.

Report a bug

### 2.6.5. Loading Roles from LDAP for Authorization (Remote Client-Server Mode)

An LDAP directory contains entries for user accounts and groups, cross referenced by attributes. Depending on the LDAP server configuration, a user entity may map the groups the user belongs to through **memberOf** attributes; a group entity may map which users belong to it through **uniqueMember** attributes; or both mappings may be maintained by the LDAP server.

Users generally authenticate against the server using a simple user name. When searching for group membership information, depending on the directory server in use, searches could be performed using this simple name or using the distinguished name of the user's entry in the directory.

The authentication step of a user connecting to the server always happens first. Once the user is successfully authenticated the server loads the user's groups. The authentication step and the authorization step each require a connection to the LDAP server. The realm optimizes this process by reusing the authentication connection for the group loading step. As will be shown within the configuration steps below it is possible to define rules within the authorization section to convert a user's simple user name to their distinguished name. The result of a "user name to distinguished name mapping" search during authentication is cached and reused during the authorization query when the **force** attribute is set to "false". When **force** is true, the search is performed again during authorization (while loading groups). This is typically done when different servers perform authentication and authorization.

```
<authorization>
    <ldap connection="...">
     <!-- OPTIONAL -->
       <username-to-dn force="true">
           <!-- Only one of the following. -->
           <username-is-dn />
           <username-filter base-dn="..." recursive="..." user-dn-
attribute="..." attribute="..." />
           <advanced-filter base-dn="..." recursive="..." user-dn-
attribute="..." filter="..." />
        </username-to-dn>

       <group-search group-name="..." iterative="..." group-dn-
attribute="..." group-name-attribute="..." >
           <!-- One of the following -->
           <group-to-principal base-dn="..." recursive="..." search-
by="...">
               <membership-filter principal-attribute="..." />
           </group-to-principal>
           <principal-to-group group-attribute="..." />
       </group-search>
    </ldap>
</authorization>
```

**IMPORTANT**

These examples specify some attributes with their default values. This is done for demonstration. Attributes that specify their default values are removed from the configuration when it is persisted by the server. The exception is the **force** attribute. It is required, even when set to the default value of **false**.

**username-to-dn**

The **`username-to-dn`** element specifies how to map the user name to the distinguished name of their entry in the LDAP directory. This element is only required when *both* of the following are true:

- The authentication and authorization steps are against different LDAP servers.
- The group search uses the distinguished name.

**1:1 username-to-dn**

This specifies that the user name entered by the remote user is the user's distinguished name.

```
<username-to-dn force="false">
   <username-is-dn />
</username-to-dn>
```

This defines a 1:1 mapping and there is no additional configuration.

**username-filter**

The next option is very similar to the simple option described above for the authentication step. A specified attribute is searched for a match against the supplied user name.

```
<username-to-dn force="true">
    <username-filter base-dn="dc=people,dc=harold,dc=example,dc=com"
recursive="false" attribute="sn" user-dn-attribute="dn" />
</username-to-dn>
```

The attributes that can be set here are:

- **`base-dn`**: The distinguished name of the context to begin the search.
- **`recursive`**: Whether the search will extend to sub contexts. Defaults to **`false`**.
- **`attribute`**: The attribute of the users entry to try and match against the supplied user name. Defaults to **`uid`**.
- **`user-dn-attribute`**: The attribute to read to obtain the users distinguished name. Defaults to **dn**.

**advanced-filter**

The final option is to specify an advanced filter, as in the authentication section this is an opportunity to use a custom filter to locate the users distinguished name.

```
<username-to-dn force="true">
    <advanced-filter base-dn="dc=people,dc=harold,dc=example,dc=com"
recursive="false" filter="sAMAccountName={0}" user-dn-attribute="dn" />
</username-to-dn>
```

For the attributes that match those in the *username-filter* example, the meaning and default values are the same. There is one new attribute:

- **`filter`**: Custom filter used to search for a user's entry where the user name will be substituted in the **`{0}`** place holder.

**IMPORTANT**

The XML must remain valid after the filter is defined so if any special characters are used such as & ensure the proper form is used. For example `&amp;` for the & character.

### The Group Search

There are two different styles that can be used when searching for group membership information. The first style is where the user's entry contains an attribute that references the groups the user is a member of. The second style is where the group contains an attribute referencing the users entry.

When there is a choice of which style to use Red Hat recommends that the configuration for a user's entry referencing the group is used. This is because with this method group information can be loaded by reading attributes of known distinguished names without having to perform any searches. The other approach requires extensive searches to identify the groups that reference the user.

Before describing the configuration here are some LDIF examples to illustrate this.

**Example 2.13. Principal to Group - LDIF example.**

This example illustrates where we have a user **`TestUserOne`** who is a member of **`GroupOne`**, **`GroupOne`** is in turn a member of **`GroupFive`**. The group membership is shown by the use of a **`memberOf`** attribute which is set to the distinguished name of the group of which the user (or group) is a member.

It is not shown here but a user could potentially have multiple **`memberOf`** attributes set, one for each group of which the user is directly a member.

```
dn: uid=TestUserOne,ou=users,dc=principal-to-group,dc=example,dc=org
objectClass: extensibleObject
objectClass: top
objectClass: groupMember
objectClass: inetOrgPerson
objectClass: uidObject
objectClass: person
objectClass: organizationalPerson
cn: Test User One
sn: Test User One
uid: TestUserOne
distinguishedName: uid=TestUserOne,ou=users,dc=principal-to-
group,dc=example,dc=org
memberOf: uid=GroupOne,ou=groups,dc=principal-to-group,dc=example,dc=org
memberOf: uid=Slashy/Group,ou=groups,dc=principal-to-
group,dc=example,dc=org
userPassword::
e1NTSEF9WFpURzhLVjc4WVZBQUJNbEI3Ym96UVAva0RTNlFNWUpLOTdTMUE9PQ==

dn: uid=GroupOne,ou=groups,dc=principal-to-group,dc=example,dc=org
objectClass: extensibleObject
objectClass: top
objectClass: groupMember
objectClass: group
objectClass: uidObject
```

```
uid: GroupOne
distinguishedName: uid=GroupOne,ou=groups,dc=principal-to-
group,dc=example,dc=org
memberOf: uid=GroupFive,ou=subgroups,ou=groups,dc=principal-to-
group,dc=example,dc=org

dn: uid=GroupFive,ou=subgroups,ou=groups,dc=principal-to-
group,dc=example,dc=org
objectClass: extensibleObject
objectClass: top
objectClass: groupMember
objectClass: group
objectClass: uidObject
uid: GroupFive
distinguishedName: uid=GroupFive,ou=subgroups,ou=groups,dc=principal-to-
group,dc=example,dc=org
```

**Example 2.14. Group to Principal - LDIF Example**

This example shows the same user **`TestUserOne`** who is a member of **`GroupOne`** which is in turn a member of **`GroupFive`** - however in this case it is an attribute **`uniqueMember`** from the group to the user being used for the cross reference.

Again the attribute used for the group membership cross reference can be repeated, if you look at *GroupFive* there is also a reference to another user *TestUserFive* which is not shown here.

```
dn: uid=TestUserOne,ou=users,dc=group-to-principal,dc=example,dc=org
objectClass: top
objectClass: inetOrgPerson
objectClass: uidObject
objectClass: person
objectClass: organizationalPerson
cn: Test User One
sn: Test User One
uid: TestUserOne
userPassword::
e1NTSEF9SjR0OTRDR1ltaHc1VVZQOEJvbXhUYjl1dkFVd1lQTmRLSEdzaWc9PQ==

dn: uid=GroupOne,ou=groups,dc=group-to-principal,dc=example,dc=org
objectClass: top
objectClass: groupOfUniqueNames
objectClass: uidObject
cn: Group One
uid: GroupOne
uniqueMember: uid=TestUserOne,ou=users,dc=group-to-
principal,dc=example,dc=org

dn: uid=GroupFive,ou=subgroups,ou=groups,dc=group-to-
principal,dc=example,dc=org
objectClass: top
objectClass: groupOfUniqueNames
objectClass: uidObject
cn: Group Five
uid: GroupFive
```

```
uniqueMember: uid=TestUserFive,ou=users,dc=group-to-
principal,dc=example,dc=org
uniqueMember: uid=GroupOne,ou=groups,dc=group-to-
principal,dc=example,dc=org
```

### General Group Searching

Before looking at the examples for the two approaches shown above we first need to define the attributes common to both of these.

```
<group-search group-name="..." iterative="..." group-dn-attribute="..."
group-name-attribute="..." >
    ...
</group-search>
```

- **`group-name`**: This attribute is used to specify the form that should be used for the group name returned as the list of groups of which the user is a member. This can either be the simple form of the group name or the group's distinguished name. If the distinguished name is required this attribute can be set to **`DISTINGUISHED_NAME`**. Defaults to **`SIMPLE`**.

- **`iterative`**: This attribute is used to indicate if, after identifying the groups a user is a member of, we should also iteratively search based on the groups to identify which groups the groups are a member of. If iterative searching is enabled we keep going until either we reach a group that is not a member if any other groups or a cycle is detected. Defaults to **`false`**.

Cyclic group membership is not a problem. A record of each search is kept to prevent groups that have already been searched from being searched again.

**IMPORTANT**

For iterative searching to work the group entries need to look the same as user entries. The same approach used to identify the groups a user is a member of is then used to identify the groups of which the group is a member. This would not be possible if for group to group membership the name of the attribute used for the cross reference changes or if the direction of the reference changes.

- **`group-dn-attribute`**: On an entry for a group which attribute is its distinguished name. Defaults to **dn**.

- **`group-name-attribute`**: On an entry for a group which attribute is its simple name. Defaults to **`uid`**.

**Example 2.15. Principal to Group Example Configuration**

Based on the example LDIF from above here is an example configuration iteratively loading a user's groups where the attribute used to cross reference is the **`memberOf`** attribute on the user.

```
<authorization>
    <ldap connection="LocalLdap">
        <username-to-dn>
            <username-filter base-dn="ou=users,dc=principal-to-
group,dc=example,dc=org" recursive="false" attribute="uid" user-dn-
```

```
attribute="dn" />
        </username-to-dn>
        <group-search group-name="SIMPLE" iterative="true" group-dn-
attribute="dn" group-name-attribute="uid">
            <principal-to-group group-attribute="memberOf" />
        </group-search>
    </ldap>
</authorization>
```

The most important aspect of this configuration is that the **`principal-to-group`** element has been added with a single attribute.

- **`group-attribute`**: The name of the attribute on the user entry that matches the distinguished name of the group the user is a member of. Defaults to **`memberOf`**.

**Example 2.16. Group to Principal Example Configuration**

This example shows an iterative search for the group to principal LDIF example shown above.

```
<authorization>
      <ldap connection="LocalLdap">
          <username-to-dn>
              <username-filter base-dn="ou=users,dc=group-to-
principal,dc=example,dc=org" recursive="false" attribute="uid" user-dn-
attribute="dn" />
          </username-to-dn>
          <group-search group-name="SIMPLE" iterative="true" group-dn-
attribute="dn" group-name-attribute="uid">
              <group-to-principal base-dn="ou=groups,dc=group-to-
principal,dc=example,dc=org" recursive="true" search-
by="DISTINGUISHED_NAME">
                  <membership-filter principal-attribute="uniqueMember"
/>
              </group-to-principal>
          </group-search>
      </ldap>
  </authorization>
```

Here an element **`group-to-principal`** is added. This element is used to define how searches for groups that reference the user entry will be performed. The following attributes are set:

- **`base-dn`**: The distinguished name of the context to use to begin the search.

- **`recursive`**: Whether sub-contexts also be searched. Defaults to **`false`**.

- **`search-by`**: The form of the role name used in searches. Valid values are **`SIMPLE`** and **`DISTINGUISHED_NAME`**. Defaults to **`DISTINGUISHED_NAME`**.

Within the *group-to-principal* element there is a *membership-filter* element to define the cross reference.

- **principal-attribute**: The name of the attribute on the group entry that references the user entry. Defaults to **member**.

Report a bug

### 2.6.6. Hot Rod Interface Security

#### 2.6.6.1. Publish Hot Rod Endpoints as a Public Interface

Red Hat JBoss Data Grid's Hot Rod server operates as a management interface as a default. To extend its operations to a public interface, alter the value of the ***interface*** parameter in the **socket-binding** element from **management** to **public** as follows:

```
<socket-binding name="hotrod" interface="public" port="11222" />
```

Report a bug

#### 2.6.6.2. Encryption of communication between Hot Rod Server and Hot Rod client

Hot Rod can be encrypted using TLS/SSL, and has the option to require certificate-based client authentication.

Use the following procedure to secure the Hot Rod connector using SSL.

**Procedure 2.3. Secure Hot Rod Using SSL/TLS**

1. **Generate a Keystore**
   Create a Java Keystore using the keytool application distributed with the JDK and add your certificate to it. The certificate can be either self signed, or obtained from a trusted CA depending on your security policy.

2. **Place the Keystore in the Configuration Directory**
   Put the keystore in the **~/JDG_HOME/standalone/configuration** directory with the **standalone-hotrod-ssl.xml** file from the **~/JDG_HOME/docs/examples/configs** directory.

3. **Declare an SSL Server Identity**
   Declare an SSL server identity within a security realm in the management section of the configuration file. The SSL server identity must specify the path to a keystore and its secret key.

```
<server-identities>
  <ssl protocol="...">
    <keystore path="..." relative-to="..." keystore-
password="${VAULT::VAULT_BLOCK::ATTRIBUTE_NAME::ENCRYPTED_VALUE}" />
  </ssl>
  <secret value="..." />
</server-identities>
```

   See Section 2.6.7.4, “Configure Hot Rod Authentication (X.509)” for details about these parameters.

4. **Add the Security Element**
   Add the security element to the Hot Rod connector as follows:

```
<hotrod-connector socket-binding="hotrod" cache-container="local">
    <encryption ssl="true" security-realm="ApplicationRealm"
require-ssl-client-auth="false" />
</hotrod-connector>
```

a. **Server Authentication of Certificate**
   If you require the server to perform authentication of the client certificate, create a truststore that contains the valid client certificates and set the ***`require-ssl-client-auth`*** attribute to **`true`**.

5. **Start the Server**
   Start the server using the following:

```
bin/standalone.sh -c standalone-hotrod-ssl.xml
```

   This will start a server with a Hot Rod endpoint on port 11222. This endpoint will only accept SSL connections.

Securing Hot Rod using SSL can also be configured programmatically.

**Example 2.17. Secure Hot Rod Using SSL/TLS**

```
package org.infinispan.client.hotrod.configuration;

import java.util.Arrays;

import javax.net.ssl.KeyManager;
import javax.net.ssl.SSLContext;
import javax.net.ssl.TrustManager;

public class SslConfiguration {
 private final boolean enabled;
 private final String keyStoreFileName;
 private final char[]
VAULT::VAULT_BLOCK::ATTRIBUTE_NAME::keyStorePassword;
 private final SSLContext sslContext;
 private final String trustStoreFileName;
 private final char[]
VAULT::VAULT_BLOCK::ATTRIBUTE_NAME::trustStorePassword;

 SslConfiguration(boolean enabled, String keyStoreFileName, char[]
keyStorePassword, SSLContext sslContext, String trustStoreFileName,
char[] trustStorePassword) {
  this.enabled = enabled;
  this.keyStoreFileName = keyStoreFileName;
  this.keyStorePassword =
VAULT::VAULT_BLOCK::ATTRIBUTE_NAME::keyStorePassword;
  this.sslContext = sslContext;
  this.trustStoreFileName = trustStoreFileName;
  this.trustStorePassword =
VAULT::VAULT_BLOCK::ATTRIBUTE_NAME::trustStorePassword;
 }

 public boolean enabled() {
  return enabled;
 }

 public String keyStoreFileName() {
  return keyStoreFileName;
 }

 public char[] keyStorePassword() {
  return keyStorePassword;
 }

 public SSLContext sslContext() {
  return sslContext;
 }

 public String trustStoreFileName() {
  return trustStoreFileName;
 }

 public char[] trustStorePassword() {
  return trustStorePassword;
 }

 @Override
 public String toString() {
  return "SslConfiguration [enabled=" + enabled + ", keyStoreFileName="
   + keyStoreFileName + ", sslContext=" + sslContext + ",
trustStoreFileName=" + trustStoreFileName + "]";
 }
}
```

**IMPORTANT**

To prevent plain text passwords from appearing in configurations or source codes, plain text passwords should be changed to Vault passwords. For more information about how to set up Vault passwords, see the *Red Hat Enterprise Application Platform Security Guide*.

Report a bug

### 2.6.7. User Authentication over Hot Rod Using SASL

User authentication over Hot Rod can be implemented using the following Simple Authentication and Security Layer (SASL) mechanisms:

- **PLAIN** is the least secure mechanism because credentials are transported in plain text format. However, it is also the simplest mechanism to implement. This mechanism can be used in conjunction with encryption (SSL) for additional security.

- **DIGEST-MD5** is a mechanism than hashes the credentials before transporting them. As a result, it is more secure than the **PLAIN** mechanism.

- **GSSAPI** is a mechanism that uses Kerberos tickets. As a result, it requires a correctly configured Kerberos Domain Controller (for example, Microsoft Active Directory).

- **EXTERNAL** is a mechanism that obtains the required credentials from the underlying transport (for example, from a `X.509` client certificate) and therefore requires client certificate encryption to work correctly.

Report a bug

### 2.6.7.1. Configure Hot Rod Authentication (GSSAPI/Kerberos)

Use the following steps to set up Hot Rod Authentication using the SASL GSSAPI/Kerberos mechanism:

**Procedure 2.4. Configure SASL GSSAPI/Kerberos Authentication**

1. **Server-side Configuration**
   The following steps must be configured on the server-side:

   a. Define a Kerberos security login module using the security domain subsystem:

```
<system-properties>
    <property name="java.security.krb5.conf"
value="/tmp/infinispan/krb5.conf"/>
    <property name="java.security.krb5.debug" value="true"/>
    <property name="jboss.security.disable.secdomain.option"
value="true"/>
</system-properties>

<security-domain name="infinispan-server" cache-type="default">
    <authentication>
        <login-module code="Kerberos" flag="required">
            <module-option name="debug" value="true"/>
            <module-option name="storeKey" value="true"/>
            <module-option name="refreshKrb5Config"
value="true"/>
            <module-option name="useKeyTab" value="true"/>
            <module-option name="doNotPrompt" value="true"/>
            <module-option name="keyTab"
value="/tmp/infinispan/infinispan.keytab"/>
            <module-option name="principal"
value="HOTROD/localhost@INFINISPAN.ORG"/>
        </login-module>
    </authentication>
</security-domain>
```

   b. Ensure that the cache-container has authorization roles defined, and these roles are applied in the cache's authorization block as seen in Section 2.4, “Configuring Red Hat JBoss Data Grid for Authorization”.

   c. Configure a Hot Rod connector as follows:

```
<hotrod-connector socket-binding="hotrod"
    cache-container="default">
 <authentication security-realm="ApplicationRealm">
  <sasl server-name="node0"
        mechanisms="{mechanism_name}"
        qop="{qop_name}"
        strength="{value}">
   <policy>
    <no-anonymous value="true" />
   </policy>
   <property
name="com.sun.security.sasl.digest.utf8">true</property>
  </sasl>
  </authentication>
</hotrod-connector>
```

- The ***server-name*** attribute specifies the name that the server declares to incoming clients. The client configuration must also contain the same server name value.

- The ***server-context-name*** attribute specifies the name of the login context used to retrieve a server subject for certain SASL mechanisms (for example, GSSAPI).

- The ***mechanisms*** attribute specifies the authentication mechanism in use. See Section 2.6.7, “User Authentication over Hot Rod Using SASL” for a list of supported mechanisms.

- The ***qop*** attribute specifies the SASL quality of protection value for the configuration. Supported values for this attribute are **auth** (authentication), **auth-int** (authentication and integrity, meaning that messages are verified against checksums to detect tampering), and **auth-conf** (authentication, integrity, and confidentiality, meaning that messages are also encrypted). Multiple values can be specified, for example, **auth-int auth-conf**. The ordering implies preference, so the first value which matches both the client and server's preference is chosen.

- The ***strength*** attribute specifies the SASL cipher strength. Valid values are **low**, **medium**, and **high**.

- The **no-anonymous** element within the **policy** element specifies whether mechanisms that accept anonymous login are permitted. Set this value to **false** to permit and **true** to deny.

2. **Client-side Configuration**
   The following steps must be configured on the client-side:

   a. Define a login module in a login configuration file (**gss.conf**) on the client side:

   ```
   GssExample {
       com.sun.security.auth.module.Krb5LoginModule required
   client=TRUE;
   };
   ```

   b. Set up the following system properties:

   ```
   java.security.auth.login.config=gss.conf
   java.security.krb5.conf=/etc/krb5.conf
   ```

**NOTE**

The **krb5.conf** file is dependent on the environment and must point to the Kerberos Key Distribution Center.

c. Configure the Hot Rod Client:

```
public class MyCallbackHandler implements CallbackHandler {
   final private String username;
   final private char[] password;
   final private String realm;

   public MyCallbackHandler() { }

   public MyCallbackHandler (String username, String realm,
char[] password) {
      this.username = username;
      this.password = password;
      this.realm = realm;
   }

   @Override
   public void handle(Callback[] callbacks) throws IOException,
UnsupportedCallbackException {
      for (Callback callback : callbacks) {
         if (callback instanceof NameCallback) {
            NameCallback nameCallback = (NameCallback) callback;
            nameCallback.setName(username);
         } else if (callback instanceof PasswordCallback) {
            PasswordCallback passwordCallback =
(PasswordCallback) callback;
            passwordCallback.setPassword(password);
         } else if (callback instanceof AuthorizeCallback) {
            AuthorizeCallback authorizeCallback =
(AuthorizeCallback) callback;

authorizeCallback.setAuthorized(authorizeCallback.getAuthenticati
onID().equals(
                  authorizeCallback.getAuthorizationID()));
         } else if (callback instanceof RealmCallback) {
            RealmCallback realmCallback = (RealmCallback)
callback;
            realmCallback.setText(realm);
         } else {
            throw new UnsupportedCallbackException(callback);
         }
      }
   }}
LoginContext lc = new LoginContext("GssExample", new
MyCallbackHandler("krb_user",
"krb_password".toCharArra()));lc.login();Subject clientSubject =
lc.getSubject();

ConfigurationBuilder clientBuilder = new
ConfigurationBuilder();clientBuilder
    .addServer()
        .host("127.0.0.1")
        .port(11222)
    .socketTimeout(1200000)
    .security()
        .authentication()
```

```
            .enable()
            .serverName("infinispan-server")
            .saslMechanism("GSSAPI")
            .clientSubject(clientSubject)
            .callbackHandler(new
MyCallbackHandler());remoteCacheManager = new
RemoteCacheManager(clientBuilder.build());RemoteCache<String,
String> cache = remoteCacheManager.getCache("secured");
```

Report a bug

### 2.6.7.2. Configure Hot Rod Authentication (MD5)

Use the following steps to set up Hot Rod Authentication using the SASL using the MD5 mechanism:

**Procedure 2.5. Configure Hot Rod Authentication (MD5)**

1. Set up the Hot Rod Connector configuration by adding the **`sasl`** element to the **`authentication`** element (for details on the **`authentication`** element, see Section 2.6.4, “Configuring Security Realms Declaratively”) as follows:

   ```
   <hotrod-connector socket-binding="hotrod"
                               cache-container="default">
       <authentication security-realm="ApplicationRealm">
           <sasl server-name="myhotrodserver"
                    mechanisms="DIGEST-MD5"
                    qop="auth" />
       </authentication>
   </hotrod-connector>
   ```

   - The ***`server-name`*** attribute specifies the name that the server declares to incoming clients. The client configuration must also contain the same server name value.
   - The ***`mechanisms`*** attribute specifies the authentication mechanism in use. See Section 2.6.7, “User Authentication over Hot Rod Using SASL” for a list of supported mechanisms.
   - The ***`qop`*** attribute specifies the SASL quality of production value for the configuration. Supported values for this attribute are **`auth`**, **`auth-int`**, and **`auth-conf`**.

2. Connect the client to the configured Hot Rod connector as follows:

   ```
   public class MyCallbackHandler implements CallbackHandler {
      final private String username;
      final private char[] password;
      final private String realm;

      public MyCallbackHandler (String username, String realm, char[]
   password) {
         this.username = username;
         this.password = password;
         this.realm = realm;
      }

      @Override
   ```

```
   public void handle(Callback[] callbacks) throws IOException,
UnsupportedCallbackException {
      for (Callback callback : callbacks) {
         if (callback instanceof NameCallback) {
            NameCallback nameCallback = (NameCallback) callback;
            nameCallback.setName(username);
         } else if (callback instanceof PasswordCallback) {
            PasswordCallback passwordCallback = (PasswordCallback)
callback;
            passwordCallback.setPassword(password);
         } else if (callback instanceof AuthorizeCallback) {
            AuthorizeCallback authorizeCallback =
(AuthorizeCallback) callback;

authorizeCallback.setAuthorized(authorizeCallback.getAuthenticationI
D().equals(
                  authorizeCallback.getAuthorizationID()));
         } else if (callback instanceof RealmCallback) {
            RealmCallback realmCallback = (RealmCallback) callback;
            realmCallback.setText(realm);
         } else {
            throw new UnsupportedCallbackException(callback);
         }
      }
   }}
ConfigurationBuilder clientBuilder = new
ConfigurationBuilder();clientBuilder
    .addServer()
        .host("127.0.0.1")
        .port(11222)
    .socketTimeout(1200000)
    .security()
        .authentication()
            .enable()
            .serverName("myhotrodserver")
            .saslMechanism("DIGEST-MD5")
            .callbackHandler(new MyCallbackHandler("myuser",
"ApplicationRealm", "qwer1234!".toCharArray()));remoteCacheManager =
new RemoteCacheManager(clientBuilder.build());RemoteCache<String,
String> cache = remoteCacheManager.getCache("secured");
```

Report a bug

### 2.6.7.3. Configure Hot Rod Using LDAP/Active Directory

Use the following to configure authentication over Hot Rod using LDAP or Microsoft Active Directory:

```
<security-realms>
 <security-realm name="ApplicationRealm">
  <authentication>
   <ldap connection="ldap_connection"
         recursive="true"
         base-dn="cn=users,dc=infinispan,dc=org">
    <username-filter attribute="cn" />
   </ldap>
  </authentication>
 </security-realm>
</security-realms>
<outbound-connections>
 <ldap name="ldap_connection"
       url="ldap://my_ldap_server"
       search-dn="CN=test,CN=Users,DC=infinispan,DC=org"
       search-credential="Test_password"/>
</outbound-connections>
```

The following are some details about the elements and parameters used in this configuration:

- The **`security-realm`** element's ***`name`*** parameter specifies the security realm to reference to use when establishing the connection.

- The **`authentication`** element contains the authentication details.

- The **`ldap`** element specifies how LDAP searches are used to authenticate a user. First, a connection to LDAP is established and a search is conducted using the supplied user name to identify the distinguished name of the user. A subsequent connection to the server is established using the password supplied by the user. If the second connection succeeds, the authentication is a success.

  - The ***`connection`*** parameter specifies the name of the connection to use to connect to LDAP.

  - The (optional) ***`recursive`*** parameter specifies whether the filter is executed recursively. The default value for this parameter is **`false`**.

  - The ***`base-dn`*** parameter specifies the distinguished name of the context to use to begin the search from.

  - The (optional) ***`user-dn`*** parameter specifies which attribute to read for the user's distinguished name after the user is located. The default value for this parameter is **dn**.

- The **`outbound-connections`** element specifies the name of the connection used to connect to the LDAP. directory.

- The **`ldap`** element specifies the properties of the outgoing LDAP connection.

  - The ***`name`*** parameter specifies the unique name used to reference this connection.

  - The ***`url`*** parameter specifies the URL used to establish the LDAP connection.

  - The ***`search-dn`*** parameter specifies the distinguished name of the user to authenticate and to perform the searches.

  - The ***`search-credential`*** parameter specifies the password required to connect to LDAP as the ***`search-dn`***.

  - The (optional) ***`initial-context-factory`*** parameter allows the overriding of the initial context factory. the default value of this parameter is **`com.sun.jndi.ldap.LdapCtxFactory`**.

Report a bug

### 2.6.7.4. Configure Hot Rod Authentication (X.509)

The `X.509` certificate can be installed at the node, and be made available to other nodes for authentication purposes for inbound and outbound SSL connections. This is enabled using the ***<server-identities/>*** element of a security realm definition, which defines how a server appears to external applications. This element can be used to configure a password to be used when establishing a remote connection, as well as the loading of an `X.509` key.

The following example shows how to install an `X.509` certificate on the node.

```
<security-realm name="ApplicationRealm">
  <server-identities>
    <ssl protocol="...">
      <keystore path="..." relative-to="..." keystore-password="..."
alias="..." key-password="..." />
    </ssl>
  </server-identities>

  [... authentication/authorization ...]

 </security-realms>
```

In the provided example, the SSL element contains the <keystore/> element, which is used to define how to load the key from the file-based keystore. The following parameters ave available for this element.

**Table 2.4. <server-identities/> Options**

| Parameter | Mandatory/Optional | Description |
|---|---|---|
| ***path*** | Mandatory | This is the path to the keystore, this can be an absolute path or relative to the next attribute. |
| ***relative-to*** | Optional | The name of a service representing a path the keystore is relative to. |
| ***keystore-password*** | Mandatory | The password required to open the keystore. |
| ***alias*** | Optional | The alias of the entry to use from the keystore - for a keystore with multiple entries in practice the first usable entry is used but this should not be relied on and the alias should be set to guarantee which entry is used. |
| ***key-password*** | Optional | The password to load the key entry, if omitted the keystore-password will be used instead. |

**NOTE**

If the following error occurs, specify a ***key-password*** as well as an ***alias*** to ensure only one key is loaded.

```
UnrecoverableKeyException: Cannot recover key
```

Report a bug

## 2.7. ACTIVE DIRECTORY AUTHENTICATION (NON-KERBEROS)

See Example 2.2, “Example of JBoss EAP LDAP login module configuration” for a non-Kerberos Active Directory Authentication configuration example.

Report a bug

## 2.8. ACTIVE DIRECTORY AUTHENTICATION USING KERBEROS (GSSAPI)

When using Red Hat JBoss Data Grid with Microsoft Active Directory, data security can be enabled via Kerberos authentication. To configure Kerberos authentication for Microsoft Active Directory, use the following procedure.

**Procedure 2.6. Configure Kerberos Authentication for Active Directory (Library Mode)**

1. Configure JBoss EAP server to authenticate itself to Kerberos. This can be done by configuring a dedicated security domain, for example:

```
<security-domain name="ldap-service" cache-type="default">
    <authentication>
        <login-module code="Kerberos" flag="required">
            <module-option name="storeKey" value="true"/>
            <module-option name="useKeyTab" value="true"/>
            <module-option name="refreshKrb5Config" value="true"/>
            <module-option name="principal"
value="ldap/localhost@INFINISPAN.ORG"/>
            <module-option name="keyTab"
value="${basedir}/keytab/ldap.keytab"/>
            <module-option name="doNotPrompt" value="true"/>
        </login-module>
    </authentication>
</security-domain>
```

2. The security domain for authentication must be configured correctly for JBoss EAP, an application must have a valid Kerberos ticket. To initiate the Kerberos ticket, you must reference another security domain using

```
<module-option name="usernamePasswordDomain" value="krb-admin"/>
```

. This points to the standard Kerberos login module described in Step 3.

```
<security-domain name="ispn-admin" cache-type="default">
```

```
    <authentication>
        <login-module code="SPNEGO" flag="requisite">
            <module-option name="password-stacking"
value="useFirstPass"/>
            <module-option name="serverSecurityDomain" value="ldap-
service"/>
            <module-option name="usernamePasswordDomain" value="krb-
admin"/>
        </login-module>
        <login-module code="AdvancedAdLdap" flag="required">
            <module-option name="password-stacking"
value="useFirstPass"/>
            <module-option name="bindAuthentication"
value="GSSAPI"/>
            <module-option name="jaasSecurityDomain" value="ldap-
service"/>
            <module-option name="java.naming.provider.url"
value="ldap://localhost:389"/>
            <module-option name="baseCtxDN"
value="ou=People,dc=infinispan,dc=org"/>
            <module-option name="baseFilter" value="
(krb5PrincipalName={0})"/>
            <module-option name="rolesCtxDN"
value="ou=Roles,dc=infinispan,dc=org"/>
            <module-option name="roleFilter" value="(member={1})"/>
            <module-option name="roleAttributeID" value="cn"/>
        </login-module>
    </authentication>
</security-domain>
```

3. The security domain authentication configuration described in the previous step points to the following standard Kerberos login module:

```
<security-domain name="krb-admin" cache-type="default">
    <authentication>
        <login-module code="Kerberos" flag="required">
            <module-option name="useKeyTab" value="true"/>
            <module-option name="principal"
value="admin@INFINISPAN.ORG"/>
            <module-option name="keyTab"
value="${basedir}/keytab/admin.keytab"/>
        </login-module>
    </authentication>
</security-domain>
```

Report a bug

# CHAPTER 3. THE SECURITY AUDIT LOGGER

Red Hat JBoss Data Grid includes a logger to audit security logs for the cache, specifically whether a cache or a cache manager operation was allowed or denied for various operations.

The default audit logger is **`org.infinispan.security.impl.DefaultAuditLogger`**. This logger outputs audit logs using the available logging framework (for example, JBoss Logging) and provides results at the **TRACE** level and the **`AUDIT`** category.

To send the **`AUDIT`** category to either a log file, a JMS queue, or a database, use the appropriate log appender.

Report a bug

## 3.1. CONFIGURE THE SECURITY AUDIT LOGGER (LIBRARY MODE)

Use the following to declaratively configure the audit logger in Red Hat JBoss Data Grid:

```
<infinispan>
  ...
 <global-security>
  <authorization audit-logger =
"org.infinispan.security.impl.DefaultAuditLogger">
     ...
  </authorization>
 </global-security>
 ...
</infinispan>
```

Use the following to programatically configure the audit logger in JBoss Data Grid:

```
GlobalConfigurationBuilder global = new GlobalConfigurationBuilder();
   global.security()
      .authorization()
         .auditLogger(new DefaultAuditLogger());
```

Report a bug

## 3.2. CONFIGURE THE SECURITY AUDIT LOGGER (REMOTE CLIENT-SERVER MODE)

Use the following code to configure the audit logger in Red Hat JBoss Data Grid Remote Client-Server Mode.

To use a different audit logger, specify it in the ***`<authorization>`*** element. The ***`<authorization>`*** element must be within the ***`<cache-container>`*** element in the Infinispan subsystem (in the **`standalone.xml`** configuration file).

```
<cache-container name="local" default-cache="default">
 <security>
 <authorization audit-
logger="org.infinispan.security.impl.DefaultAuditLogger">
 <identity-role-mapper/>
```

```
<role name="admin" permissions="ALL"/>
<role name="reader" permissions="READ"/>
<role name="writer" permissions="WRITE"/>
<role name="supervisor" permissions="ALL_READ ALL_WRITE"/>
</authorization>
</security>
<local-cache name="default" start="EAGER">
<locking isolation="NONE" acquire-timeout="30000" concurrency-
level="1000" striping="false"/>
<transaction mode="NONE"/>
<security>
<authorization roles="admin reader writer supervisor"/>
</security>
</local-cache>
```

**NOTE**

The default audit logger for server mode is **`org.jboss.as.clustering.infinispan.subsystem.ServerAuditLogger`** which sends the log messages to the server audit log. See the *Management Interface Audit Logging* chapter in the JBoss Enterprise Application Platform *Administration and Configuration Guide* for more information.

Report a bug

## 3.3. CUSTOM AUDIT LOGGERS

Users can implement custom audit loggers in Red Hat JBoss Data Grid Library and Remote Client-Server Mode. The custom logger must implement the **`org.infinispan.security.AuditLogger`** interface. If no custom logger is provided, the default logger (**`DefaultAuditLogger`**) is used.

Report a bug

# CHAPTER 4. SECURITY FOR CLUSTER TRAFFIC

## 4.1. NODE AUTHENTICATION AND AUTHORIZATION (REMOTE CLIENT-SERVER MODE)

Security can be enabled at node level via SASL protocol, which enables node authentication against a security realm. This requires nodes to authenticate each other when joining or merging with a cluster. For detailed information about security realms, see Section 2.6.1, "About Security Realms".

The following example depicts the <sasl /> element, which leverages the SASL protocol. Both **DIGEST-MD5** or **GSSAPI** mechanisms are currently supported.

**Example 4.1. Configure SASL Authentication**

```
<management>
    <security-realms>
        <!-- Additional configuration information here -->
        <security-realm name="ClusterRealm">
            <authentication>
                <properties path="cluster-users.properties" relative-
to="jboss.server.config.dir"/>
                </authentication>
                <authorization>
                    <properties path="cluster-roles.properties"
relative-to="jboss.server.config.dir"/>
                </authorization>
            </security-realm>
        </security-realms>
        <!-- Additional configuration information here -->
    </security-realms>
</management>

<stack name="udp">
    <!-- Additional configuration information here -->
    <sasl mech="DIGEST-MD5" security-realm="ClusterRealm" cluster-
role="cluster">
        <property name="client_name">node1</property>
        <property name="client_password">password</property>
    </sasl>
    <!-- Additional configuration information here -->
</stack>
```

In the provided example, the nodes use the **DIGEST-MD5** mechanism to authenticate against the **ClusterRealm**. In order to join, nodes must have the ***cluster*** role.

The ***cluster-role*** attribute determines the role all nodes must belong to in the security realm in order to **JOIN** or **MERGE** with the cluster. Unless it has been specified, the ***cluster-role*** attribute is the name of the clustered ***<cache-container>*** by default. Each node identifies itself using the ***client-name*** property. If none is specified, the hostname on which the server is running will be used.

This name can also be overridden by specifying the **jboss.node.name** system property that can be overridden on the command line. For example:

```
$ clustered.sh -Djboss.node.name=node001
```

**NOTE**

JGroups AUTH protocol is not integrated with security realms, and its use is not advocated for Red Hat JBoss Data Grid.

Report a bug

### 4.1.1. Configure Node Authentication for Cluster Security (DIGEST-MD5)

The following example demonstrates how to use **DIGEST-MD5** with a properties-based security realm, with a dedicated realm for cluster node.

**Example 4.2. Using the DIGEST-MD5 Mechanism**

```
<management>
         <security-realms>
             <security-realm name="ClusterRealm">
                 <authentication>
                     <properties path="cluster-users.properties"
relative-to="jboss.server.config.dir"/>
                 </authentication>
                 <authorization>
                     <properties path="cluster-roles.properties"
relative-to="jboss.server.config.dir"/>
                 </authorization>
             </security-realm>
         </security-realms>
</management>
<subsystem xmlns="urn:infinispa:server:jgroups:6.1" default-
stack="${jboss.default.jgroups.stack:udp}">
     <stack name="udp">
         <transport type="UDP" socket-binding="jgroups-udp"/>
         <protocol type="PING"/>
         <protocol type="MERGE2"/>
         <protocol type="FD_SOCK" socket-binding="jgroups-udp-fd"/>
         <protocol type="FD_ALL"/>
         <protocol type="pbcast.NAKACK"/>
         <protocol type="UNICAST2"/>
         <protocol type="pbcast.STABLE"/>
         <protocol type="pbcast.GMS"/>
         <protocol type="UFC"/>
         <protocol type="MFC"/>
         <protocol type="FRAG2"/>
         <protocol type="RSVP"/>
         <sasl security-realm="ClusterRealm" mech="DIGEST-MD5">
             <property name="client_password>...</property>
         </sasl>
     </stack>
</subsystem>
<subsystem xmlns="urn:infinispan:server:core:6.1" default-cache-
container="clustered">
     <cache-container name="clustered" default-cache="default">
```

```
         <transport executor="infinispan-transport" lock-
timeout="60000" stack="udp"/>
         <!-- various clustered cache definitions here -->
     </cache-container>
</subsystem>
```

In the provided example, supposing the hostnames of the various nodes are **node001**, **node002**, **node003**, the ***cluster-users.properties*** will contain:

- **node001=/<node001passwordhash>/**
- **node002=/<node002passwordhash>/**
- **node003=/<node003passwordhash>/**

The ***cluster-roles.properties*** will contain:

- node001=clustered
- node002=clustered
- node003=clustered

To generate these values, the following **add-users.sh** script can be used:

```
$ add-user.sh -up cluster-users.properties -gp cluster-roles.properties -r
ClusterRealm -u node001 -g clustered -p <password>
```

The **MD5** password hash of the node must also be placed in the " ***client_password***" property of the <sasl/> element.

```
<property name="client_password>...</property>
```

**NOTE**

To increase security, it is recommended that this password be stored using a Vault. For more information about vault expressions, see the *Red Hat Enterprise Application Platform Security Guide*

Once node security has been set up as discussed here, the cluster coordinator will validate each **JOIN**ing and **MERGE**ing node's credentials against the realm before letting the node become part of the cluster view.

Report a bug

### 4.1.2. Configure Node Authentication for Cluster Security (GSSAPI/Kerberos)

When using the **GSSAPI** mechanism, the ***client_name*** is used as the name of a Kerberos-enabled login module defined within the security domain subsystem. For a full procedure on how to do this, see Section 2.6.7.1, “Configure Hot Rod Authentication (GSSAPI/Kerberos)” .

**Example 4.3. Using the Kerberos Login Module**

```
<security-domain name="krb-node0" cache-type="default">
    <authentication>
        <login-module code="Kerberos" flag="required">
            <module-option name="storeKey" value="true"/>
            <module-option name="useKeyTab" value="true"/>
            <module-option name="refreshKrb5Config" value="true"/>
            <module-option name="principal"
value="jgroups/node0/clustered@INFINISPAN.ORG"/>
            <module-option name="keyTab"
value="${jboss.server.config.dir}/keytabs/jgroups_node0_clustered.keytab
"/>
            <module-option name="doNotPrompt" value="true"/>
        </login-module>
    </authentication>
</security-domain>
```

The following property must be set in the <sasl/> element to reference it:

```
<sasl <!-- Additional configuration information here --> >
     <property name="login_module_name">
       <!-- Additional configuration information here -->
     </property>
</sasl>
```

As a result, the **`authentication`** section of the security realm is ignored, as the nodes will be validated against the Kerberos Domain Controller. The **`authorization`** configuration is still required, as the node principal must belong to the required cluster-role.

In all cases, it is recommended that a shared authorization database, such as LDAP, be used to validate node membership in order to simplify administration.

By default, the principal of the joining node must be in the following format:

```
jgroups/$NODE_NAME/$CACHE_CONTAINER_NAME@REALM
```

Report a bug

## 4.2. CONFIGURE NODE SECURITY IN LIBRARY MODE

In Library mode, node authentication is configured directly in the JGroups configuration. JGroups can be configured so that nodes must authenticate each other when joining or merging with a cluster. The authentication uses SASL and is enabled by adding the **SASL** protocol to your JGroups XML configuration.

SASL relies on JAAS notions, such as **`CallbackHandlers`**, to obtain certain information necessary for the authentication handshake. Users must supply their own **`CallbackHandlers`** on both client and server sides.

**IMPORTANT**

The **JAAS** API is only available when configuring user authentication and authorization, and is not available for node security.

NOTE

In the provided example, **CallbackHandler** classes are examples only, and not contained in the Red Hat JBoss Data Grid release. Users must provide the appropriate **CallbackHandler** classes for their specific LDAP implementation.

**Example 4.4. Setting Up SASL Authentication in JGroups**

```
<SASL mech="DIGEST-MD5"
    client_name="node_user"
    client_password="node_password"

server_callback_handler_class="org.example.infinispan.security.JGroupsSa
slServerCallbackHandler"

client_callback_handler_class="org.example.infinispan.security.JGroupsSa
slClientCallbackHandler"
    sasl_props="com.sun.security.sasl.digest.realm=test_realm" />
```

The above example uses the **DIGEST-MD5** mechanism. Each node must declare the user and password it will use when joining the cluster.

IMPORTANT

The SASL protocol must be placed before the GMS protocol in order for authentication to take effect.

The following example demonstrates how to implement a **CallbackHandler** class. In this example, login and password are checked against values provided via Java properties when JBoss Data Grid is started, and authorization is checked against ***role*** which is defined in the class ( **"test_user"**).

**Example 4.5. Callback Handler Class**

```
public class SaslPropAuthUserCallbackHandler implements CallbackHandler
{

   private static final String APPROVED_USER = "test_user";

   private final String name;
   private final char[] password;
   private final String realm;

   public SaslPropAuthUserCallbackHandler() {
      this.name = System.getProperty("sasl.username");
      this.password = System.getProperty("sasl.password").toCharArray();
      this.realm = System.getProperty("sasl.realm");
   }

   @Override
   public void handle(Callback[] callbacks) throws IOException,
UnsupportedCallbackException {
      for (Callback callback : callbacks) {
         if (callback instanceof PasswordCallback) {
            ((PasswordCallback) callback).setPassword(password);
         } else if (callback instanceof NameCallback) {
            ((NameCallback) callback).setName(name);
         } else if (callback instanceof AuthorizeCallback) {
            AuthorizeCallback authorizeCallback = (AuthorizeCallback)
callback;
            if
(APPROVED_USER.equals(authorizeCallback.getAuthorizationID())) {
               authorizeCallback.setAuthorized(true);
            } else {
               authorizeCallback.setAuthorized(false);
            }
         } else if (callback instanceof RealmCallback) {
            RealmCallback realmCallback = (RealmCallback) callback;
            realmCallback.setText(realm);
         } else {
            throw new UnsupportedCallbackException(callback);
         }
      }
   }

}
```

For authentication, specify the **`javax.security.auth.callback.NameCallback`** and **`javax.security.auth.callback.PasswordCallback`** callbacks

For authorization, specify the callbacks required for authentication, as well as specifying the **`javax.security.sasl.AuthorizeCallback`** callback.

Report a bug

### 4.2.1. Configure Node Authentication for Library Mode (DIGEST-MD5)

The behavior of a node differs depending on whether it is the coordinator node or any other node. The coordinator acts as the SASL server, with the joining or merging nodes behaving as SASL clients. When using the DIGEST-MD5 mechanism in Library mode, the server and client callback must be specified so that the server and client are aware of how to obtain the credentials. Therefore, two **`CallbackHandlers`** are required:

- The **`server_callback_handler_class`** is used by the coordinator.
- The **`client_callback_handler_class`** is used by other nodes.

The following example demonstrates these **`CallbackHandlers`**.

**Example 4.6. Callback Handlers**

```
<SASL mech="DIGEST-MD5"
      client_name="node_name"
      client_password="node_password"

client_callback_handler_class="${CLIENT_CALLBACK_HANDLER_IN_CLASSPATH}"
```

```
server_callback_handler_class="${SERVER_CALLBACK_HANDLER_IN_CLASSPATH}"
      sasl_props="com.sun.security.sasl.digest.realm=test_realm"
/>
```

JGroups is designed so that all nodes are able to act as coordinator or client depending on cluster behavior, so if the current coordinator node goes down, the next node in the succession chain will become the coordinator. Given this behavior, both server and client callback handlers must be identified within SASL for Red Hat JBoss Data Grid implementations.

Report a bug

### 4.2.2. Configure Node Authentication for Library Mode (GSSAPI)

When performing node authentication in Library mode using the GSSAPI mechanism, the ***`login_module_name`*** parameter must be specified instead of ***`callback`***.

This login module is used to obtain a valid Kerberos ticket, which is used to authenticate a client to the server. The ***`server_name`*** must also be specified, as the client principal is constructed as **`jgroups/$server_name@REALM`**.

**Example 4.7. Specifying the login module and server on the coordinator node**

```
<SASL mech="GSSAPI"
         server_name="node0/clustered"
         login_module_name="krb-node0"

server_callback_handler_class="org.infinispan.test.integration.security.
utils.SaslPropCallbackHandler" />
```

On the coordinator node, the ***`server_callback_handler_class`*** must be specified for node authorization. This will determine if the authenticated joining node has permission to join the cluster.

**NOTE**

The server principal is always constructed as **`jgroups/server_name`**, therefore the server principal in Kerberos must also be **`jgroups/server_name`**. For example, if the server name in Kerberos is **`jgroups/node1/mycache`**, then the server name must be **`node1/mycache`**.

Report a bug

### 4.2.3. Node Authorization in Library Mode

The **SASL** protocol in JGroups is concerned only with the authentication process. To implement node authorization, you can do so within the server callback handler by throwing an Exception.

The following example demonstrates this.

**Example 4.8. Implementing Node Authorization**

```
public class AuthorizingServerCallbackHandler implements CallbackHandler
{

@Override
public void handle(Callback[] callbacks) throws IOException,
UnsupportedCallbackException {
        for (Callback callback : callbacks) {
            <!-- Additional configuration information here -->
            if (callback instanceof AuthorizeCallback) {
                AuthorizeCallback acb = (AuthorizeCallback) callback;
                if
(!"myclusterrole".equals(acb.getAuthenticationID()))) {
                    throw new SecurityException("Unauthorized node "
+user);
            }
            <!-- Additional configuration information here -->
        }
    }
}
```

Report a bug

## 4.3. JGROUPS ENCRYPT

JGroups includes the **ENCRYPT** protocol to provide encryption for cluster traffic.

By default, encryption only encrypts the message body; it does not encrypt message headers. To encrypt the entire message, including all headers, as well as destination and source addresses, the property ***encrypt_entire_message*** must be `true`. **ENCRYPT** must also be below any protocols with headers that must be encrypted.

The **ENCRYPT** layer is used to encrypt and decrypt communication in JGroups. The JGroups **ENCRYPT** protocol can be used in two ways:

- Configured with a secretKey in a keystore.
- Configured with algorithms and key sizes.

Each message is identified as encrypted with a specific encryption header identifying the encrypt header and an MD5 digest identifying the version of the key being used to encrypt and decrypt messages.

Report a bug

### 4.3.1. ENCRYPT Configured with a secretKey in a Key Store

The **ENCRYPT** class can be configured with a secretKey in a keystore so it is usable at any layer in JGroups without requiring a coordinator. This also provides protection against passive monitoring without the complexity of key exchange.

This method can be used by inserting the encryption layer at any point in the JGroups stack, which will encrypt all events of a type MSG that have a non-nill message buffer. The following is an example of the entry in this form:

```
<ENCRYPT key_store_name="defaultStore.keystore"
store_password="${VAULT::VAULT_BLOCK::ATTRIBUTE_NAME::ENCRYPTED_VALUE}"
alias="myKey"/>
```

The secretKey must be already generated in a keystore file in order to use the **ENCRYPT** layer in this manner. The directory containing the keystore file must be on the application classpath.

**NOTE**

The secretKey keystore file cannot be created using the keytool application shipped with JDK. The **KeyStoreGenerator** java file is included in the demo package that can be used to generate a suitable keystore.

Report a bug

### 4.3.2. ENCRYPT Using a Key Store

**ENCRYPT** uses store type JCEKS. To generate a keystore compatible with JCEKS, use the following command line options:

```
$ keytool -genseckey -alias myKey -keypass changeit -storepass changeit -
keyalg Blowfish -keysize 56 -keystore defaultStore.keystore -storetype
JCEKS
```

**ENCRYPT** can then be configured by adding the following information to the JGroups file used by the application.

```
<ENCRYPT key_store_name="defaultStore.keystore"

store_password="${VAULT::VAULT_BLOCK::ATTRIBUTE_NAME::ENCRYPTED_VALUE}"
         alias="myKey"/>
```

Standard Java properties can also be used in the configuration, and it is possible to pass the path to JGroups configuration via the **-D** option during start up.

The default, pre-configured JGroups files are packaged in **infinispan-embedded.jar**, alternatively, you can create your own configuration file. See the *Configure JGroups (Library Mode)* section in the *Red Hat JBoss Data Grid Administration and Configuration Guide*for instructions on how to set up JBoss Data Grid to use custom JGroups configurations in library mode.

In Remote Client-Server mode, the JGroups configuration is part of the main server configuration file.

**NOTE**

The ***defaultStore.keystore*** must be found in the classpath.

Report a bug

### 4.3.3. ENCRYPT Configured with Algorithms and Key Sizes

In this encryption mode, the coordinator selects the secretKey and distributes it to all peers. There is no keystore, and keys are distributed using a public/private key exchange. Instead, encryption occurs as follows:

1. The secret key is generated and distributed by the controller.
2. When a view change occurs, a peer requests the secret key by sending a key request with its own public key.
3. The controller encrypts the secret key with the public key, and sends it back to the peer.
4. The peer then decrypts and installs the key as its own secret key.

In this mode, the **ENCRYPT** layer must be placed above the GMS protocol in the configuration.

**Example 4.9. ENCRYPT Layer**

```
<config <!-- Additional configuration information here --> >
    <UDP />
    <PING />
    <MERGE2 />
    <FD />
    <VERIFY_SUSPECT />
    <ENCRYPT encrypt_entire_message="false"
             sym_init="128" sym_algorithm="AES/ECB/PKCS5Padding"
             asym_init="512" asym_algorithm="RSA"/>
    <pbcast.NAKACK />
    <UNICAST />
    <pbcast.STABLE />
    <FRAG2 />
    <pbcast.GMS />
</config>
```

In the provided example, the sequence numbers for the **NAKACK** and **UNICAST** protocols will be encrypted.

View changes that identify a new controller result in a new session key being generated and distributed to all peers. This is a substantial overhead in an application with high peer churn.

When encrypting an entire message, the message must be marshalled into a byte buffer before being encrypted, resulting in decreased performance.

Report a bug

### 4.3.4. ENCRYPT Configuration Parameters

The following table provides configuration parameters for the **ENCRYPT** JGroups protocol:

**Table 4.1. Configuration Parameters**

| Name | Description |
|---|---|
| alias | Alias used for recovering the key. Change the default. |

| Name | Description |
|---|---|
| asymAlgorithm | Cipher engine transformation for asymmetric algorithm. Default is RSA. |
| asymInit | Initial public/private key length. Default is 512. |
| asymProvider | Cryptographic Service Provider. Default is Bouncy Castle Provider. |
| encrypt_entire_message | |
| id | Give the protocol a different ID if needed so we can have multiple instances of it in the same stack. |
| keyPassword | Password for recovering the key. Change the default. |
| keyStoreName | File on classpath that contains keystore repository. |
| level | Sets the logger level (see javadocs). |
| name | Give the protocol a different name if needed so we can have multiple instances of it in the same stack. |
| stats | Determines whether to collect statistics (and expose them via JMX). Default is true. |
| storePassword | Password used to check the integrity/unlock the keystore. Change the default. It is recommended that passwords are stored using Vault. |
| symAlgorithm | Cipher engine transformation for symmetric algorithm. Default is AES. |
| symInit | Initial key length for matching symmetric algorithm. Default is 128. |

Report a bug

# APPENDIX A. REVISION HISTORY

| | | |
|---|---|---|
| **Revision 6.5.0-3** | **Mon Sep 14 2015** | **Christian Huffman** |
| Updating for 6.5.1 release. | | |
| **Revision 6.5.0-2** | **Tue 16 June 2015** | **Christian Huffman** |
| Updated for GA. | | |
| **Revision 6.5.0-1** | **Fri 8 May 2015** | **Christian Huffman** |
| Moved the Security Audit Logger to a separate chapter. | | |
| **Revision 6.5.0-0** | **Wed 6 May 2015** | **Christian Huffman** |
| Initial draft release. | | |