Red Hat Data Grid 8.4

Data Grid Security Guide

Enable and configure Data Grid security

Last Updated: 2024-04-19

Red Hat Data Grid 8.4 Data Grid Security Guide

Enable and configure Data Grid security

Legal Notice

Copyright © 2024 Red Hat, Inc.

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

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

Abstract

Protect your Data Grid deployments from network intruders. Restrict data access to authorized users.

Table of Contents

RED HAT DATA GRID

Data Grid is a high-performance, distributed in-memory data store.

Schemaless data structure

Flexibility to store different objects as key-value pairs.

Grid-based data storage

Designed to distribute and replicate data across clusters.

Elastic scaling

Dynamically adjust the number of nodes to meet demand without service disruption.

Data interoperability

Store, retrieve, and query data in the grid from different endpoints.

DATA GRID DOCUMENTATION

Documentation for Data Grid is available on the Red Hat customer portal.

- Data Grid 8.4 Documentation
- Data Grid 8.4 Component Details
- Supported Configurations for Data Grid 8.4
- Data Grid 8 Feature Support
- Data Grid Deprecated Features and Functionality

DATA GRID DOWNLOADS

Access the Data Grid Software Downloads on the Red Hat customer portal.

NOTE

You must have a Red Hat account to access and download Data Grid software.

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message .

CHAPTER 1. SECURITY AUTHORIZATION WITH ROLE-BASED ACCESS CONTROL

Role-based access control (RBAC) capabilities use different permissions levels to restrict user interactions with Data Grid.

NOTE

For information on creating users and configuring authorization specific to remote or embedded caches, see:

- Configuring user roles and permissions with Data Grid Server
- Programmatically configuring user roles and permissions

1.1. DATA GRID USER ROLES AND PERMISSIONS

Data Grid includes several roles that provide users with permissions to access caches and Data Grid resources.

Role	Permissions	Description
admin	ALL	Superuser with all permissions including control of the Cache Manager lifecycle.
deployer	ALL_READ, ALL_WRITE, LISTEN, EXEC, MONITOR, CREATE	Can create and delete Data Grid resources in addition to **application** permissions.
application	ALL_READ, ALL_WRITE, LISTEN, EXEC, MONITOR	Has read and write access to Data Grid resources in addition to **observer** permissions. Can also listen to events and execute server tasks and scripts.
observer	ALL_READ, MONITOR	Has read access to Data Grid resources in addition to **monitor** permissions.
monitor	MONITOR	Can view statistics via JMX and the **metrics** endpoint.

Additional resources

- org.infinispan.security.AuthorizationPermission Enum
- Data Grid configuration schema reference

1.1.1. Permissions

User roles are sets of permissions with different access levels.

Table 1.1. Cache Manager permissions

Permission	Function	Description
CONFIGURATION	**defineConfiguration**	Defines new cache configurations.
LISTEN	**addListener**	Registers listeners against a Cache Manager.
LIFECYCLE	**stop**	Stops the Cache Manager.
CREATE	**createCache**, **removeCache**	Create and remove container resources such as caches, counters, schemas, and scripts.
MONITOR	**getStats**	Allows access to JMX statistics and the **metrics** endpoint.
ALL	-	Includes all Cache Manager permissions.

Table 1.2. Cache permissions

Permission	Function	Description
READ	**get**, **contains**	Retrieves entries from a cache.
WRITE	**put**, **putIfAbsent**, **replace**, **remove**, **evict**	Writes, replaces, removes, evicts data in a cache.
EXEC	**distexec**, **streams**	Allows code execution against a cache.
LISTEN	**addListener**	Registers listeners against a cache.
BULK_READ	**keySet**, **values**, **entrySet**, **query**	Executes bulk retrieve operations.
BULK_WRITE	**clear**, **putAll**	Executes bulk write operations.
LIFECYCLE	**start**, **stop**	Starts and stops a cache.

ADMIN	**getVersion**, **addInterceptor***, **removeInterceptor**, **getInterceptorChain**, **getEvictionManager**, **getComponentRegistry**, **getDistributionManager**, **getAuthorizationManager**, **evict**, **getRpcManager**, **getCacheConfiguration**, **getCacheManager**, **getInvocationContextContainer**, **setAvailability**, **getDataContainer**, **getStats**, **getXAResource**	Allows access to underlying components and internal structures.
MONITOR	**getStats**	Allows access to JMX statistics and the **metrics** endpoint.
ALL	-	Includes all cache permissions.
ALL_READ	-	Combines the READ and BULK_READ permissions.
ALL_WRITE	-	Combines the WRITE and BULK_WRITE permissions.

Additional resources

- Data Grid Security API

1.1.2. Role and permission mappers

Data Grid implements users as a collection of principals. Principals represent either an individual user identity, such as a username, or a group to which the users belong. Internally, these are implemented with the **javax.security.auth.Subject** class.

To enable authorization, the principals must be mapped to role names, which are then expanded into a set of permissions.

Data Grid includes the **PrincipalRoleMapper** API for associating security principals to roles, and the **RolePermissionMapper** API for associating roles with specific permissions.

Data Grid provides the following role and permission mapper implementations:

Cluster role mapper

Stores principal to role mappings in the cluster registry.

Cluster permission mapper

Stores role to permission mappings in the cluster registry. Allows you to dynamically modify user roles and permissions.

Identity role mapper

Uses the principal name as the role name. The type or format of the principal name depends on the source. For example, in an LDAP directory the principal name could be a Distinguished Name (DN).

Common name role mapper

Uses the Common Name (CN) as the role name. You can use this role mapper with an LDAP directory or with client certificates that contain Distinguished Names (DN); for example **cn=managers,ou=people,dc=example,dc=com** maps to the **managers** role.

1.1.2.1. Mapping users to roles and permissions in Data Grid

Consider the following user retrieved from an LDAP server, as a collection of DNs:

```
CN=myapplication,OU=applications,DC=mycompany
CN=dataprocessors,OU=groups,DC=mycompany
CN=finance,OU=groups,DC=mycompany
```

Using the **Common name role mapper**, the user would be mapped to the following roles:

```
dataprocessors
finance
```

Data Grid has the following role definitions:

```
dataprocessors: ALL_WRITE ALL_READ
finance: LISTEN
```

The user would have the following permissions:

```
ALL_WRITE ALL_READ LISTEN
```

Additional resources

- Data Grid Security API
- org.infinispan.security.PrincipalRoleMapper
- org.infinispan.security.RolePermissionMapper
- org.infinispan.security.mappers.IdentityRoleMapper
- org.infinispan.security.mappers.CommonNameRoleMapper

1.1.3. Configuring role mappers

Data Grid enables the cluster role mapper and cluster permission mapper by default. To use a different implementation for role mapping, you must configure the role mappers.

Procedure

1. Open your Data Grid configuration for editing.
2. Declare the role mapper as part of the security authorization in the Cache Manager configuration.

3. Save the changes to your configuration.

With embedded caches you can programmatically configure role and permission mappers with the **principalRoleMapper()** and **rolePermissionMapper()** methods.

Role mapper configuration

XML

```
<cache-container>
  <security>
    <authorization>
      <common-name-role-mapper />
    </authorization>
  </security>
</cache-container>
```

JSON

```
{
  "infinispan" : {
    "cache-container" : {
      "security" : {
        "authorization" : {
          "common-name-role-mapper": {}
        }
      }
    }
  }
}
```

YAML

```
infinispan:
  cacheContainer:
    security:
      authorization:
        commonNameRoleMapper: ~
```

Additional resources

- Data Grid configuration schema reference

1.2. CONFIGURING CACHES WITH SECURITY AUTHORIZATION

Add security authorization to caches to enforce role-based access control (RBAC). This requires Data Grid users to have a role with a sufficient level of permission to perform cache operations.

Prerequisites

- Create Data Grid users and either grant them with roles or assign them to groups.

Procedure

1. Open your Data Grid configuration for editing.
2. Add a **security** section to the configuration.
3. Specify roles that users must have to perform cache operations with the **authorization** element.
   You can implicitly add all roles defined in the Cache Manager or explicitly define a subset of roles.
4. Save the changes to your configuration.

Implicit role configuration

The following configuration implicitly adds every role defined in the Cache Manager:

XML

```xml
<distributed-cache>
  <security>
    <authorization/>
  </security>
</distributed-cache>
```

JSON

```json
{
  "distributed-cache": {
    "security": {
      "authorization": {
        "enabled": true
      }
    }
  }
}
```

YAML

```yaml
distributedCache:
  security:
    authorization:
      enabled: true
```

Explicit role configuration

The following configuration explicitly adds a subset of roles defined in the Cache Manager. In this case Data Grid denies cache operations for any users that do not have one of the configured roles.

XML

```xml
<distributed-cache>
  <security>
    <authorization roles="admin supervisor"/>
  </security>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "security": {
      "authorization": {
        "enabled": true,
        "roles": ["admin","supervisor"]
      }
    }
  }
}
```

YAML

```
distributedCache:
  security:
    authorization:
      enabled: true
      roles: ["admin","supervisor"]
```

CHAPTER 2. SECURITY REALMS

Security realms integrate Data Grid Server deployments with the network protocols and infrastructure in your environment that control access and verify user identities.

2.1. CREATING SECURITY REALMS

Add security realms to Data Grid Server configuration to control access to deployments. You can add one or more security realm to your configuration.

NOTE

When you add security realms to your configuration, Data Grid Server automatically enables the matching authentication mechanisms for the Hot Rod and REST endpoints.

Prerequisites

- Add socket bindings to your Data Grid Server configuration as required.
- Create keystores, or have a PEM file, to configure the security realm with TLS/SSL encryption. Data Grid Server can also generate keystores at startup.
- Provision the resources or services that the security realm configuration relies on. For example, if you add a token realm, you need to provision OAuth services.

This procedure demonstrates how to configure multiple property realms. Before you begin, you need to create properties files that add users and assign permissions with the Command Line Interface (CLI). Use the **user create** commands as follows:

```
user create <username> -p <changeme> -g <role> \
    --users-file=application-users.properties \
    --groups-file=application-groups.properties

user create <username> -p <changeme> -g <role> \
    --users-file=management-users.properties \
    --groups-file=management-groups.properties
```

TIP

Run **user create --help** for examples and more information.

NOTE

Adding credentials to a properties realm with the CLI creates the user only on the server instance to which you are connected. You must manually synchronize credentials in a properties realm to each node in the cluster.

Procedure

1. Open your Data Grid Server configuration for editing.
2. Use the **security-realms** element in the **security** configuration to contain create multiple security realms.

3. Add a security realm with the **security-realm** element and give it a unique name with the **name** attribute.
   To follow the example, create one security realm named **application-realm** and another named **management-realm**.

4. Provide the TLS/SSL identify for Data Grid Server with the **server-identities** element and configure a keystore as required.

5. Specify the type of security realm by adding one the following elements or fields:

   - **properties-realm**
   - **ldap-realm**
   - **token-realm**
   - **truststore-realm**

6. Specify properties for the type of security realm you are configuring as appropriate.
   To follow the example, specify the ***.properties** files you created with the CLI using the **path** attribute on the **user-properties** and **group-properties** elements or fields.

7. If you add multiple different types of security realm to your configuration, include the **distributed-realm** element or field so that Data Grid Server uses the realms in combination with each other.

8. Configure Data Grid Server endpoints to use the security realm with the with the **security-realm** attribute.

9. Save the changes to your configuration.

Multiple property realms

XML

```xml
<server xmlns="urn:infinispan:server:14.0">
  <security>
    <security-realms>
      <security-realm name="application-realm">
        <properties-realm groups-attribute="Roles">
          <user-properties path="application-users.properties"/>
          <group-properties path="application-groups.properties"/>
        </properties-realm>
      </security-realm>
      <security-realm name="management-realm">
        <properties-realm groups-attribute="Roles">
          <user-properties path="management-users.properties"/>
          <group-properties path="management-groups.properties"/>
        </properties-realm>
      </security-realm>
    </security-realms>
  </security>
</server>
```

JSON

```
{
  "server": {
    "security": {
      "security-realms": [{
        "name": "management-realm",
        "properties-realm": {
          "groups-attribute": "Roles",
          "user-properties": {
            "digest-realm-name": "management-realm",
            "path": "management-users.properties"
          },
          "group-properties": {
            "path": "management-groups.properties"
          }
        }
      }, {
        "name": "application-realm",
        "properties-realm": {
          "groups-attribute": "Roles",
          "user-properties": {
            "digest-realm-name": "application-realm",
            "path": "application-users.properties"
          },
          "group-properties": {
            "path": "application-groups.properties"
          }
        }
      }]
    }
  }
}
```

YAML

```
server:
  security:
    securityRealms:
      - name: "management-realm"
        propertiesRealm:
          groupsAttribute: "Roles"
          userProperties:
            digestRealmName: "management-realm"
            path: "management-users.properties"
          groupProperties:
            path: "management-groups.properties"
      - name: "application-realm"
        propertiesRealm:
          groupsAttribute: "Roles"
          userProperties:
            digestRealmName: "application-realm"
            path: "application-users.properties"
          groupProperties:
            path: "application-groups.properties"
```

2.2. SETTING UP KERBEROS IDENTITIES

Add Kerberos identities to a security realm in your Data Grid Server configuration to use *keytab* files that contain service principal names and encrypted keys, derived from Kerberos passwords.

Prerequisites

- Have Kerberos service account principals.

> **NOTE**
>
> *keytab* files can contain both user and service account principals. However, Data Grid Server uses service account principals only which means it can provide identity to clients and allow clients to authenticate with Kerberos servers.

In most cases, you create unique principals for the Hot Rod and REST endpoints. For example, if you have a "datagrid" server in the "INFINISPAN.ORG" domain you should create the following service principals:

- **hotrod/datagrid@INFINISPAN.ORG** identifies the Hot Rod service.
- **HTTP/datagrid@INFINISPAN.ORG** identifies the REST service.

Procedure

1. Create keytab files for the Hot Rod and REST services.

   Linux

   ```
   ktutil
   ktutil:  addent -password -p datagrid@INFINISPAN.ORG -k 1 -e aes256-cts
   Password for datagrid@INFINISPAN.ORG: [enter your password]
   ktutil:  wkt http.keytab
   ktutil:  quit
   ```

   Microsoft Windows

   ```
   ktpass -princ HTTP/datagrid@INFINISPAN.ORG -pass * -mapuser
   INFINISPAN\USER_NAME
   ktab -k http.keytab -a HTTP/datagrid@INFINISPAN.ORG
   ```

2. Copy the keytab files to the **server/conf** directory of your Data Grid Server installation.
3. Open your Data Grid Server configuration for editing.
4. Add a **server-identities** definition to the Data Grid server security realm.
5. Specify the location of keytab files that provide service principals to Hot Rod and REST connectors.
6. Name the Kerberos service principals.
7. Save the changes to your configuration.

Kerberos identity configuration

XML

```xml
<server xmlns="urn:infinispan:server:14.0">
  <security>
    <security-realms>
      <security-realm name="kerberos-realm">
        <server-identities>
          <!-- Specifies a keytab file that provides a Kerberos identity. -->
          <!-- Names the Kerberos service principal for the Hot Rod endpoint. -->
          <!-- The required="true" attribute specifies that the keytab file must be present when the server
starts. -->
          <kerberos keytab-path="hotrod.keytab"
                    principal="hotrod/datagrid@INFINISPAN.ORG"
                    required="true"/>
          <!-- Specifies a keytab file and names the Kerberos service principal for the REST endpoint. -->
          <kerberos keytab-path="http.keytab"
                    principal="HTTP/localhost@INFINISPAN.ORG"
                    required="true"/>
        </server-identities>
      </security-realm>
    </security-realms>
  </security>
  <endpoints>
    <endpoint socket-binding="default"
              security-realm="kerberos-realm">
     <hotrod-connector>
        <authentication>
           <sasl server-name="datagrid"
                 server-principal="hotrod/datagrid@INFINISPAN.ORG"/>
        </authentication>
     </hotrod-connector>
     <rest-connector>
       <authentication server-principal="HTTP/localhost@INFINISPAN.ORG"/>
     </rest-connector>
   </endpoint>
  </endpoints>
</server>
```

JSON

```json
{
  "server": {
    "security": {
      "security-realms": [{
        "name": "kerberos-realm",
        "server-identities": [{
          "kerberos": {
            "principal": "hotrod/datagrid@INFINISPAN.ORG",
            "keytab-path": "hotrod.keytab",
            "required": true
          },
          "kerberos": {
```

```
          "principal": "HTTP/localhost@INFINISPAN.ORG",
          "keytab-path": "http.keytab",
          "required": true
        }
      }]
    }]
  },
  "endpoints": {
    "endpoint": {
      "socket-binding": "default",
      "security-realm": "kerberos-realm",
      "hotrod-connector": {
        "authentication": {
          "security-realm": "kerberos-realm",
          "sasl": {
            "server-name": "datagrid",
            "server-principal": "hotrod/datagrid@INFINISPAN.ORG"
          }
        }
      },
      "rest-connector": {
        "authentication": {
          "server-principal": "HTTP/localhost@INFINISPAN.ORG"
        }
      }
    }
  }
 }
}
```

YAML

```
server:
  security:
    securityRealms:
      - name: "kerberos-realm"
        serverIdentities:
          - kerberos:
              principal: "hotrod/datagrid@INFINISPAN.ORG"
              keytabPath: "hotrod.keytab"
              required: "true"
          - kerberos:
              principal: "HTTP/localhost@INFINISPAN.ORG"
              keytabPath: "http.keytab"
              required: "true"
  endpoints:
    endpoint:
      socketBinding: "default"
      securityRealm: "kerberos-realm"
      hotrodConnector:
        authentication:
          sasl:
            serverName: "datagrid"
            serverPrincipal: "hotrod/datagrid@INFINISPAN.ORG"
      restConnector:
```

```
      authentication:
        securityRealm: "kerberos-realm"
        serverPrincipal" : "HTTP/localhost@INFINISPAN.ORG"
```

2.3. PROPERTY REALMS

Property realms use property files to define users and groups.

- **users.properties** contains Data Grid user credentials. Passwords can be pre-digested with the **DIGEST-MD5** and **DIGEST** authentication mechanisms.
- **groups.properties** associates users with roles and permissions.

NOTE

You can avoid authentication issues that relate to a property file by using the Data Grid CLI to enter the correct security realm name to the file. You can find the correct security realm name of your Data Grid Server by opening the **infinispan.xml** file and navigating to the **<security-realm name>** property. When you copy a property file from one Data Grid Server to another, make sure that the security realm name appropriates to the correct authentication mechanism for the target endpoint.

users.properties

```
myuser=a_password
user2=another_password
```

groups.properties

```
myuser=supervisor,reader,writer
user2=supervisor
```

Property realm configuration

XML

```
<server xmlns="urn:infinispan:server:14.0">
  <security>
    <security-realms>
      <security-realm name="default">
        <!-- groups-attribute configures the "groups.properties" file to contain security authorization roles.
-->
        <properties-realm groups-attribute="Roles">
          <user-properties path="users.properties"
                           relative-to="infinispan.server.config.path"
                           plain-text="true"/>
          <group-properties path="groups.properties"
                            relative-to="infinispan.server.config.path"/>
        </properties-realm>
      </security-realm>
    </security-realms>
  </security>
</server>
```

▪

JSON

```
{
  "server": {
    "security": {
      "security-realms": [{
        "name": "default",
        "properties-realm": {
          "groups-attribute": "Roles",
          "user-properties": {
            "digest-realm-name": "default",
            "path": "users.properties",
            "relative-to": "infinispan.server.config.path",
            "plain-text": true
          },
          "group-properties": {
            "path": "groups.properties",
            "relative-to": "infinispan.server.config.path"
          }
        }
      }]
    }
  }
}
```

YAML

```
server:
  security:
    securityRealms:
      - name: "default"
        propertiesRealm:
          # groupsAttribute configures the "groups.properties" file
          # to contain security authorization roles.
          groupsAttribute: "Roles"
          userProperties:
            digestRealmName: "default"
            path: "users.properties"
            relative-to: 'infinispan.server.config.path'
            plainText: "true"
          groupProperties:
            path: "groups.properties"
            relative-to: 'infinispan.server.config.path'
```

2.4. LDAP REALMS

LDAP realms connect to LDAP servers, such as OpenLDAP, Red Hat Directory Server, Apache Directory Server, or Microsoft Active Directory, to authenticate users and obtain membership information.

NOTE

LDAP servers can have different entry layouts, depending on the type of server and deployment. It is beyond the scope of this document to provide examples for all possible configurations.

2.4.1. LDAP connection properties

Specify the LDAP connection properties in the LDAP realm configuration.

The following properties are required:

url	Specifies the URL of the LDAP server. The URL should be in format **ldap://hostname:port** or **ldaps://hostname:port** for secure connections using TLS.
principal	Specifies a distinguished name (DN) of a valid user in the LDAp server. The DN uniquely identifies the user within the LDAP directory structure.
credential	Corresponds to the password associated with the principal mentioned above.

IMPORTANT

The principal for LDAP connections must have necessary privileges to perform LDAP queries and access specific attributes.

TIP

Enabling **connection-pooling** significantly improves the performance of authentication to LDAP servers. The connection pooling mechanism is provided by the JDK. For more information see Connection Pooling Configuration and Java Tutorials: Pooling.

2.4.2. LDAP realm user authentication methods

Configure the user authentication method in the LDAP realm.

The LDAP realm can authenticate users in two ways:

Hashed password comparison	by comparing the hashed password stored in a user's password attribute (usually **userPassword**)
Direct verification	by authenticating against the LDAP server using the supplied credentials Direct verification is the only approach that works with Active Directory, because access to the **password** attribute is forbidden.

IMPORTANT

You cannot use endpoint authentication mechanisms that performs hashing with the **direct-verification** attribute, since this method requires having the password in clear text. As a result you must use the **BASIC** authentication mechanism with the REST endpoint and **PLAIN** with the Hot Rod endpoint to integrate with Active Directory Server. A more secure alternative is to use Kerberos, which allows the **SPNEGO**, **GSSAPI**, and **GS2-KRB5** authentication mechanisms.

The LDAP realm searches the directory to find the entry which corresponds to the authenticated user. The **rdn-identifier** attribute specifies an LDAP attribute that finds the user entry based on a provided identifier, which is typically a username; for example, the **uid** or **sAMAccountName** attribute. Add **search-recursive="true"** to the configuration to search the directory recursively. By default, the search for the user entry uses the **(rdn_identifier={0})** filter. You can specify a different filter using the **filter-name** attribute.

2.4.3. Mapping user entries to their associated groups

In the LDAP realm configuration, specify the **attribute-mapping** element to retrieve and associate all groups that a user is a member of.

The membership information is stored typically in two ways:

- Under group entries that usually have class **groupOfNames** or **groupOfUniqueNames** in the **member** attribute. This is the default behavior in most LDAP installations, except for Active Directory. In this case, you can use an attribute filter. This filter searches for entries that match the supplied filter, which locates groups with a **member** attribute equal to the user's DN. The filter then extracts the group entry's CN as specified by **from**, and adds it to the user's **Roles**.
- In the user entry in the **memberOf** attribute. This is typically the case for Active Directory. In this case you should use an attribute reference such as the following:
  <attribute-reference reference="memberOf" from="cn" to="Roles" />

  This reference gets all **memberOf** attributes from the user's entry, extracts the CN as specified by **from**, and adds them to the user's groups (**Roles** is the internal name used to map the groups).

2.4.4. LDAP realm configuration reference

XML

```
<server xmlns="urn:infinispan:server:14.0">
  <security>
    <security-realms>
      <security-realm name="ldap-realm">
        <!-- Specifies connection properties. -->
        <ldap-realm url="ldap://my-ldap-server:10389"
                    principal="uid=admin,ou=People,dc=infinispan,dc=org"
                    credential="strongPassword"
                    connection-timeout="3000"
                    read-timeout="30000"
                    connection-pooling="true"
                    referral-mode="ignore"
                    page-size="30"
                    direct-verification="true">
          <!-- Defines how principals are mapped to LDAP entries. -->
          <identity-mapping rdn-identifier="uid"
                            search-dn="ou=People,dc=infinispan,dc=org"
                            search-recursive="false">
            <!-- Retrieves all the groups of which the user is a member. -->
            <attribute-mapping>
              <attribute from="cn" to="Roles"
                         filter="(&amp;(objectClass=groupOfNames)(member={1}))"
                         filter-dn="ou=Roles,dc=infinispan,dc=org"/>
            </attribute-mapping>
          </identity-mapping>
        </ldap-realm>
      </security-realm>
    </security-realms>
  </security>
</server>
```

JSON

```
{
  "server": {
    "security": {
      "security-realms": [{
        "name": "ldap-realm",
        "ldap-realm": {
          "url": "ldap://my-ldap-server:10389",
          "principal": "uid=admin,ou=People,dc=infinispan,dc=org",
          "credential": "strongPassword",
          "connection-timeout": "3000",
          "read-timeout": "30000",
          "connection-pooling": "true",
          "referral-mode": "ignore",
          "page-size": "30",
          "direct-verification": "true",
          "identity-mapping": {
            "rdn-identifier": "uid",
            "search-dn": "ou=People,dc=infinispan,dc=org",
            "search-recursive": "false",
            "attribute-mapping": [{
              "from": "cn",
              "to": "Roles",
              "filter": "(&(objectClass=groupOfNames)(member={1}))",
              "filter-dn": "ou=Roles,dc=infinispan,dc=org"
            }]
          }
        }
      }]
    }
  }
}
```

YAML

```
server:
  security:
    securityRealms:
      - name: ldap-realm
        ldapRealm:
          url: 'ldap://my-ldap-server:10389'
          principal: 'uid=admin,ou=People,dc=infinispan,dc=org'
          credential: strongPassword
          connectionTimeout: '3000'
          readTimeout: '30000'
          connectionPooling: true
          referralMode: ignore
          pageSize: '30'
          directVerification: true
          identityMapping:
            rdnIdentifier: uid
            searchDn: 'ou=People,dc=infinispan,dc=org'
            searchRecursive: false
            attributeMapping:
              - filter: '(&(objectClass=groupOfNames)(member={1}))'
                filterDn: 'ou=Roles,dc=infinispan,dc=org'
                from: cn
                to: Roles
```

2.4.4.1. LDAP realm principal rewriting

Principals obtained by SASL authentication mechanisms such as **GSSAPI**, **GS2-KRB5** and **Negotiate** usually include the domain name, for example **myuser@INFINISPAN.ORG**. Before using these principals in LDAP queries, it is necessary to transform them to ensure their compatibility. This process is called rewriting.

Data Grid includes the following transformers:

case-principal-transformer	rewrites the principal to either all uppercase or all lowercase. For example **MyUser** would be rewritten as **MYUSER** in uppercase mode and **myuser** in lowercase mode.
common-name-principal-transformer	rewrites principals in the LDAP Distinguished Name format (as defined by RFC 4514). It extracts the first attribute of type **CN** (commonName). For example, **DN=CN=myuser,OU=myorg,DC=mydomain** would be rewritten as**myuser**.
regex-principal-transformer	rewrites principals using a regular expression with capturing groups, allowing, for example, for extractions of any substring.

2.4.4.2. LDAP principal rewriting configuration reference

Case principal transformer

XML

```xml
<server xmlns="urn:infinispan:server:14.0">
  <security>
    <security-realms>
      <security-realm name="ldap-realm">
        <ldap-realm url="ldap://${org.infinispan.test.host.address}:10389"
                    principal="uid=admin,ou=People,dc=infinispan,dc=org"
                    credential="strongPassword">
          <name-rewriter>
            <!-- Defines a rewriter that transforms usernames to lowercase -->
            <case-principal-transformer uppercase="false"/>
          </name-rewriter>
          <!-- further configuration omitted -->
        </ldap-realm>
      </security-realm>
    </security-realms>
  </security>
</server>
```

JSON

```json
{
  "server": {
    "security": {
      "security-realms": [{
        "name": "ldap-realm",
        "ldap-realm": {
          "principal": "uid=admin,ou=People,dc=infinispan,dc=org",
          "url": "ldap://${org.infinispan.test.host.address}:10389",
          "credential": "strongPassword",
          "name-rewriter": {
            "case-principal-transformer": {
              "uppercase": false
            }
          }
        }
      }]
    }
  }
}
```

YAML

```yaml
server:
  security:
    securityRealms:
      - name: "ldap-realm"
        ldapRealm:
          principal: "uid=admin,ou=People,dc=infinispan,dc=org"
          url: "ldap://${org.infinispan.test.host.address}:10389"
          credential: "strongPassword"
          nameRewriter:
            casePrincipalTransformer:
              uppercase: false
          # further configuration omitted
```

-

Common name principal transformer

XML

```
<server xmlns="urn:infinispan:server:14.0">
  <security>
    <security-realms>
      <security-realm name="ldap-realm">
        <ldap-realm url="ldap://${org.infinispan.test.host.address}:10389"
                    principal="uid=admin,ou=People,dc=infinispan,dc=org"
                    credential="strongPassword">
          <name-rewriter>
            <!-- Defines a rewriter that obtains the first CN from a DN -->
            <common-name-principal-transformer />
          </name-rewriter>
          <!-- further configuration omitted -->
        </ldap-realm>
      </security-realm>
    </security-realms>
  </security>
</server>
```

JSON

```
{
  "server": {
    "security": {
      "security-realms": [{
        "name": "ldap-realm",
        "ldap-realm": {
          "principal": "uid=admin,ou=People,dc=infinispan,dc=org",
          "url": "ldap://${org.infinispan.test.host.address}:10389",
          "credential": "strongPassword",
          "name-rewriter": {
            "common-name-principal-transformer": {}
          }
        }
      }]
    }
  }
}
```

YAML

```
server:
  security:
    securityRealms:
      - name: "ldap-realm"
        ldapRealm:
          principal: "uid=admin,ou=People,dc=infinispan,dc=org"
          url: "ldap://${org.infinispan.test.host.address}:10389"
          credential: "strongPassword"
        nameRewriter:
          commonNamePrincipalTransformer: ~
        # further configuration omitted
```

Regex principal transformer

XML

```
<server xmlns="urn:infinispan:server:14.0">
  <security>
    <security-realms>
      <security-realm name="ldap-realm">
        <ldap-realm url="ldap://${org.infinispan.test.host.address}:10389"
                    principal="uid=admin,ou=People,dc=infinispan,dc=org"
                    credential="strongPassword">
          <name-rewriter>
            <!-- Defines a rewriter that extracts the username from the principal using a regular
expression. -->
            <regex-principal-transformer pattern="(.*)@INFINISPAN\.ORG"
                                         replacement="$1"/>
          </name-rewriter>
          <!-- further configuration omitted -->
        </ldap-realm>
      </security-realm>
    </security-realms>
  </security>
</server>
```

JSON

```
{
  "server": {
    "security": {
      "security-realms": [{
        "name": "ldap-realm",
        "ldap-realm": {
          "principal": "uid=admin,ou=People,dc=infinispan,dc=org",
          "url": "ldap://${org.infinispan.test.host.address}:10389",
          "credential": "strongPassword",
          "name-rewriter": {
            "regex-principal-transformer": {
              "pattern": "(.*)@INFINISPAN\\.ORG",
              "replacement": "$1"
            }
          }
        }
      }]
    }
  }
}
```

YAML

```
server:
  security:
    securityRealms:
      - name: "ldap-realm"
        ldapRealm:
          principal: "uid=admin,ou=People,dc=infinispan,dc=org"
          url: "ldap://${org.infinispan.test.host.address}:10389"
          credential: "strongPassword"
          nameRewriter:
            regexPrincipalTransformer:
              pattern: (.*)@INFINISPAN\.ORG
              replacement: "$1"
          # further configuration omitted
```

2.4.4.3. LDAP user and group mapping process with Data Grid

This example illustrates the process of loading and internally mapping LDAP users and groups to Data Grid subjects. The following is a LDIF (LDAP Data Interchange Format) file, which describes multiple LDAP entries:

LDIF

```
# Users

dn: uid=root,ou=People,dc=infinispan,dc=org
objectclass: top
objectclass: uidObject
objectclass: person
uid: root
cn: root
sn: root
userPassword: strongPassword

# Groups

dn: cn=admin,ou=Roles,dc=infinispan,dc=org
objectClass: top
objectClass: groupOfNames
cn: admin
description: the Infinispan admin group
member: uid=root,ou=People,dc=infinispan,dc=org

dn: cn=monitor,ou=Roles,dc=infinispan,dc=org
objectClass: top
objectClass: groupOfNames
cn: monitor
description: the Infinispan monitor group
member: uid=root,ou=People,dc=infinispan,dc=org
```

The **root** user is a member of the **admin** and **monitor** groups.

When a request to authenticate the user **root** with the password **strongPassword** is made on one of the endpoints, the following operations are performed:

- The username is optionally rewritten using the chosen principal transformer.

- The realm searches within the **ou=People,dc=infinispan,dc=org** tree for an entry whose **uid** attribute is equal to **root** and finds the entry with DN **uid=root,ou=People,dc=infinispan,dc=org**, which becomes the user principal.
- The realm searches within the **u=Roles,dc=infinispan,dc=org** tree for entries of **objectClass=groupOfNames** that include **uid=root,ou=People,dc=infinispan,dc=org** in the **member** attribute. In this case it finds two entries: **cn=admin,ou=Roles,dc=infinispan,dc=org** and **cn=monitor,ou=Roles,dc=infinispan,dc=org**. From these entries, it extracts the **cn** attributes which become the group principals.

The resulting subject will therefore look like:

- NamePrincipal: **uid=root,ou=People,dc=infinispan,dc=org**
- RolePrincipal: **admin**
- RolePrincipal: **monitor**

At this point, the global authorization mappers are applied on the above subject to convert the principals into roles. The roles are then expanded into a set of permissions, which are validated against the requested cache and operation.

2.5. TOKEN REALMS

Token realms use external services to validate tokens and require providers that are compatible with RFC-7662 (OAuth2 Token Introspection), such as Red Hat SSO.

Token realm configuration

XML

```
<server xmlns="urn:infinispan:server:14.0">
  <security>
    <security-realms>
      <security-realm name="token-realm">
        <!-- Specifies the URL of the authentication server. -->
        <token-realm name="token"
                     auth-server-url="https://oauth-server/auth/">
          <!-- Specifies the URL of the token introspection endpoint. -->
          <oauth2-introspection introspection-url="https://oauth-
server/auth/realms/infinispan/protocol/openid-connect/token/introspect"
                                client-id="infinispan-server"
                                client-secret="1fdca4ec-c416-47e0-867a-3d471af7050f"/>
        </token-realm>
      </security-realm>
    </security-realms>
  </security>
</server>
```

JSON

```
{
  "server": {
    "security": {
      "security-realms": [{
        "name": "token-realm",
        "token-realm": {
          "auth-server-url": "https://oauth-server/auth/",
          "oauth2-introspection": {
            "client-id": "infinispan-server",
            "client-secret": "1fdca4ec-c416-47e0-867a-3d471af7050f",
            "introspection-url": "https://oauth-server/auth/realms/infinispan/protocol/openid-
connect/token/introspect"
          }
        }
      }]
    }
  }
}
```

YAML

```
server:
  security:
    securityRealms:
      - name: token-realm
        tokenRealm:
          authServerUrl: 'https://oauth-server/auth/'
          oauth2Introspection:
            clientId: infinispan-server
            clientSecret: '1fdca4ec-c416-47e0-867a-3d471af7050f'
            introspectionUrl: 'https://oauth-server/auth/realms/infinispan/protocol/openid-
connect/token/introspect'
```

2.6. TRUST STORE REALMS

Trust store realms use certificates, or certificates chains, that verify Data Grid Server and client identities when they negotiate connections.

Keystores

Contain server certificates that provide a Data Grid Server identity to clients. If you configure a keystore with server certificates, Data Grid Server encrypts traffic using industry standard SSL/TLS protocols.

Trust stores

Contain client certificates, or certificate chains, that clients present to Data Grid Server. Client trust stores are optional and allow Data Grid Server to perform client certificate authentication.

Client certificate authentication

You must add the **require-ssl-client-auth="true"** attribute to the endpoint configuration if you want Data Grid Server to validate or authenticate client certificates.

Trust store realm configuration

XML

```
<server xmlns="urn:infinispan:server:14.0">
```

```
  <security>
    <security-realms>
      <security-realm name="trust-store-realm">
        <server-identities>
          <ssl>
            <!-- Provides an SSL/TLS identity with a keystore that contains server certificates. -->
            <keystore path="server.p12"
                      relative-to="infinispan.server.config.path"
                      keystore-password="secret"
                      alias="server"/>
            <!-- Configures a trust store that contains client certificates or part of a certificate chain. -->
            <truststore path="trust.p12"
                        relative-to="infinispan.server.config.path"
                        password="secret"/>
          </ssl>
        </server-identities>
        <!-- Authenticates client certificates against the trust store. If you configure this, the trust store
must contain the public certificates for all clients. -->
        <truststore-realm/>
      </security-realm>
    </security-realms>
  </security>
</server>
```

JSON

```
{
  "server": {
    "security": {
      "security-realms": [{
        "name": "trust-store-realm",
        "server-identities": {
          "ssl": {
            "keystore": {
              "path": "server.p12",
              "relative-to": "infinispan.server.config.path",
              "keystore-password": "secret",
              "alias": "server"
            },
            "truststore": {
              "path": "trust.p12",
              "relative-to": "infinispan.server.config.path",
              "password": "secret"
            }
          }
        },
        "truststore-realm": {}
      }]
    }
  }
}
```

YAML

```
server:
  security:
    securityRealms:
      - name: "trust-store-realm"
        serverIdentities:
          ssl:
            keystore:
              path: "server.p12"
              relative-to: "infinispan.server.config.path"
              keystore-password: "secret"
              alias: "server"
            truststore:
              path: "trust.p12"
              relative-to: "infinispan.server.config.path"
              password: "secret"
        truststoreRealm: ~
```

2.7. DISTRIBUTED SECURITY REALMS

Distributed realms combine multiple different types of security realms. When users attempt to access the Hot Rod or REST endpoints, Data Grid Server uses each security realm in turn until it finds one that can perform the authentication.

Distributed realm configuration

XML

```
<server xmlns="urn:infinispan:server:14.0">
  <security>
    <security-realms>
      <security-realm name="distributed-realm">
        <ldap-realm url="ldap://my-ldap-server:10389"
                    principal="uid=admin,ou=People,dc=infinispan,dc=org"
                    credential="strongPassword">
          <identity-mapping rdn-identifier="uid"
                            search-dn="ou=People,dc=infinispan,dc=org"
                            search-recursive="false">
            <attribute-mapping>
              <attribute from="cn" to="Roles"
                         filter="(&amp;(objectClass=groupOfNames)(member={1}))"
                         filter-dn="ou=Roles,dc=infinispan,dc=org"/>
            </attribute-mapping>
          </identity-mapping>
        </ldap-realm>
        <properties-realm groups-attribute="Roles">
          <user-properties path="users.properties"
                           relative-to="infinispan.server.config.path"/>
          <group-properties path="groups.properties"
                            relative-to="infinispan.server.config.path"/>
        </properties-realm>
        <distributed-realm/>
      </security-realm>
    </security-realms>
  </security>
</server>
```

JSON

```
{
  "server": {
    "security": {
      "security-realms": [{
        "name": "distributed-realm",
        "ldap-realm": {
          "principal": "uid=admin,ou=People,dc=infinispan,dc=org",
          "url": "ldap://my-ldap-server:10389",
          "credential": "strongPassword",
          "identity-mapping": {
            "rdn-identifier": "uid",
            "search-dn": "ou=People,dc=infinispan,dc=org",
            "search-recursive": false,
            "attribute-mapping": {
              "attribute": {
                "filter": "(&(objectClass=groupOfNames)(member={1}))",
                "filter-dn": "ou=Roles,dc=infinispan,dc=org",
                "from": "cn",
                "to": "Roles"
              }
            }
          }
        },
        "properties-realm": {
          "groups-attribute": "Roles",
          "user-properties": {
            "digest-realm-name": "distributed-realm",
            "path": "users.properties"
          },
          "group-properties": {
            "path": "groups.properties"
          }
        },
        "distributed-realm": {}
      }]
    }
  }
}
```

YAML

```
server:
  security:
    securityRealms:
      - name: "distributed-realm"
        ldapRealm:
          principal: "uid=admin,ou=People,dc=infinispan,dc=org"
          url: "ldap://my-ldap-server:10389"
      credential: "strongPassword"
      identityMapping:
        rdnIdentifier: "uid"
        searchDn: "ou=People,dc=infinispan,dc=org"
        searchRecursive: "false"
        attributeMapping:
          attribute:
            filter: "(&(objectClass=groupOfNames)(member={1}))"
            filterDn: "ou=Roles,dc=infinispan,dc=org"
            from: "cn"
            to: "Roles"
    propertiesRealm:
      groupsAttribute: "Roles"
      userProperties:
        digestRealmName: "distributed-realm"
        path: "users.properties"
      groupProperties:
        path: "groups.properties"
    distributedRealm: ~
```

CHAPTER 3. ENDPOINT AUTHENTICATION MECHANISMS

Data Grid Server can use custom SASL and HTTP authentication mechanisms for Hot Rod and REST endpoints.

3.1. DATA GRID SERVER AUTHENTICATION

Authentication restricts user access to endpoints as well as the Data Grid Console and Command Line Interface (CLI).

Data Grid Server includes a "default" security realm that enforces user authentication. Default authentication uses a property realm with user credentials stored in the **server/conf/users.properties** file. Data Grid Server also enables security authorization by default so you must assign users with permissions stored in the **server/conf/groups.properties** file.

TIP

Use the **user create** command with the Command Line Interface (CLI) to add users and assign permissions. Run **user create --help** for examples and more information.

3.2. CONFIGURING DATA GRID SERVER AUTHENTICATION MECHANISMS

You can explicitly configure Hot Rod and REST endpoints to use specific authentication mechanisms. Configuring authentication mechanisms is required only if you need to explicitly override the default mechanisms for a security realm.

NOTE

Each **endpoint** section in your configuration must include **hotrod-connector** and **rest-connector** elements or fields. For example, if you explicitly declare a **hotrod-connector** you must also declare a **rest-connector** even if it does not configure an authentication mechanism.

Prerequisites

- Add security realms to your Data Grid Server configuration as required.

Procedure

1. Open your Data Grid Server configuration for editing.
2. Add an **endpoint** element or field and specify the security realm that it uses with the **security-realm** attribute.
3. Add a **hotrod-connector** element or field to configure the Hot Rod endpoint.
   a. Add an **authentication** element or field.
   b. Specify SASL authentication mechanisms for the Hot Rod endpoint to use with the **sasl mechanisms** attribute.
   c. If applicable, specify SASL quality of protection settings with the **qop** attribute.

   d. Specify the Data Grid Server identity with the **server-name** attribute if necessary.

4. Add a **rest-connector** element or field to configure the REST endpoint.

   a. Add an **authentication** element or field.

   b. Specify HTTP authentication mechanisms for the REST endpoint to use with the **mechanisms** attribute.

5. Save the changes to your configuration.

Authentication mechanism configuration

The following configuration specifies SASL mechanisms for the Hot Rod endpoint to use for authentication:

XML

```xml
<server xmlns="urn:infinispan:server:14.0">
  <endpoints>
    <endpoint socket-binding="default"
              security-realm="my-realm">
      <hotrod-connector>
        <authentication>
          <sasl mechanisms="SCRAM-SHA-512 SCRAM-SHA-384 SCRAM-SHA-256
                            SCRAM-SHA-1 DIGEST-SHA-512 DIGEST-SHA-384
                            DIGEST-SHA-256 DIGEST-SHA DIGEST-MD5 PLAIN"
                server-name="infinispan"
                qop="auth"/>
        </authentication>
      </hotrod-connector>
      <rest-connector>
        <authentication mechanisms="DIGEST BASIC"/>
      </rest-connector>
    </endpoint>
  </endpoints>
</server>
```

JSON

```json
{
  "server": {
    "endpoints": {
      "endpoint": {
        "socket-binding": "default",
        "security-realm": "my-realm",
        "hotrod-connector": {
          "authentication": {
            "security-realm": "default",
            "sasl": {
              "server-name": "infinispan",
              "mechanisms": ["SCRAM-SHA-512", "SCRAM-SHA-384", "SCRAM-SHA-256", "SCRAM-SHA-1", "DIGEST-SHA-512", "DIGEST-SHA-384", "DIGEST-SHA-256", "DIGEST-SHA", "DIGEST-MD5", "PLAIN"],
              "qop": ["auth"]
            }
          }
        },
        "rest-connector": {
          "authentication": {
            "mechanisms": ["DIGEST", "BASIC"],
            "security-realm": "default"
          }
        }
      }
    }
  }
}
```

YAML

```
server:
  endpoints:
    endpoint:
      socketBinding: "default"
      securityRealm: "my-realm"
      hotrodConnector:
        authentication:
          securityRealm: "default"
          sasl:
            serverName: "infinispan"
            mechanisms:
              - "SCRAM-SHA-512"
              - "SCRAM-SHA-384"
              - "SCRAM-SHA-256"
              - "SCRAM-SHA-1"
              - "DIGEST-SHA-512"
              - "DIGEST-SHA-384"
              - "DIGEST-SHA-256"
              - "DIGEST-SHA"
              - "DIGEST-MD5"
              - "PLAIN"
            qop:
              - "auth"
      restConnector:
        authentication:
          mechanisms:
            - "DIGEST"
            - "BASIC"
          securityRealm: "default"
```

3.2.1. Disabling authentication

In local development environments or on isolated networks you can configure Data Grid to allow unauthenticated client requests. When you disable user authentication you should also disable authorization in your Data Grid security configuration.

Procedure

1. Open your Data Grid Server configuration for editing.

2. Remove the **security-realm** attribute from the **endpoints** element or field.

3. Remove any **authorization** elements from the **security** configuration for the **cache-container** and each cache configuration.

4. Save the changes to your configuration.

XML

```
<server xmlns="urn:infinispan:server:14.0">
  <endpoints socket-binding="default"/>
</server>
```

JSON

```
{
  "server": {
    "endpoints": {
      "endpoint": {
        "socket-binding": "default"
      }
    }
  }
}
```

YAML

```
server:
  endpoints:
    endpoint:
      socketBinding: "default"
```

3.3. DATA GRID SERVER AUTHENTICATION MECHANISMS

Data Grid Server automatically configures endpoints with authentication mechanisms that match your security realm configuration. For example, if you add a Kerberos security realm then Data Grid Server enables the **GSSAPI** and **GS2-KRB5** authentication mechanisms for the Hot Rod endpoint.

IMPORTANT

Currently, you cannot use the Lightweight Directory Access Protocol (LDAP) protocol with the **DIGEST** or **SCRAM** authentication mechanisms, because these mechanisms require access to specific hashed passwords.

Hot Rod endpoints

Data Grid Server enables the following SASL authentication mechanisms for Hot Rod endpoints when your configuration includes the corresponding security realm:

Security realm	SASL authentication mechanism
Property realms and LDAP realms	SCRAM-*, DIGEST-*, **SCRAM-***
Token realms	OAUTHBEARER
Trust realms	EXTERNAL
Kerberos identities	GSSAPI, GS2-KRB5
SSL/TLS identities	PLAIN

REST endpoints

Data Grid Server enables the following HTTP authentication mechanisms for REST endpoints when your configuration includes the corresponding security realm:

Security realm	HTTP authentication mechanism
Property realms and LDAP realms	DIGEST
Token realms	BEARER_TOKEN
Trust realms	CLIENT_CERT
Kerberos identities	SPNEGO
SSL/TLS identities	BASIC

3.3.1. SASL authentication mechanisms

Data Grid Server supports the following SASL authentications mechanisms with Hot Rod endpoints:

Authentication mechanism	Description	Security realm type	Related details
PLAIN	Uses credentials in plain-text format. You should use **PLAIN** authentication with encrypted connections only.	Property realms and LDAP realms	Similar to the **BASIC** HTTP mechanism.

Authentication mechanism	Description	Security realm type	Related details
DIGEST-*	Uses hashing algorithms and nonce values. Hot Rod connectors support **DIGEST-MD5**, **DIGEST-SHA**, **DIGEST-SHA-256**, **DIGEST-SHA-384**, and **DIGEST-SHA-512** hashing algorithms, in order of strength.	Property realms and LDAP realms	Similar to the **Digest** HTTP mechanism.
SCRAM-*	Uses *salt* values in addition to hashing algorithms and nonce values. Hot Rod connectors support **SCRAM-SHA**, **SCRAM-SHA-256**, **SCRAM-SHA-384**, and **SCRAM-SHA-512** hashing algorithms, in order of strength.	Property realms and LDAP realms	Similar to the **Digest** HTTP mechanism.
GSSAPI	Uses Kerberos tickets and requires a Kerberos Domain Controller. You must add a corresponding **kerberos** server identity in the realm configuration. In most cases, you also specify an **ldap-realm** to provide user membership information.	Kerberos realms	Similar to the **SPNEGO** HTTP mechanism.

Authentication mechanism	Description	Security realm type	Related details
GS2-KRB5	Uses Kerberos tickets and requires a Kerberos Domain Controller. You must add a corresponding **kerberos** server identity in the realm configuration. In most cases, you also specify an **ldap-realm** to provide user membership information.	Kerberos realms	Similar to the **SPNEGO** HTTP mechanism.
EXTERNAL	Uses client certificates.	Trust store realms	Similar to the **CLIENT_CERT** HTTP mechanism.
OAUTHBEARER	Uses OAuth tokens and requires a **token-realm** configuration.	Token realms	Similar to the **BEARER_TOKEN** HTTP mechanism.

3.3.2. SASL quality of protection (QoP)

If SASL mechanisms support integrity and privacy protection (QoP) settings, you can add them to your Hot Rod endpoint configuration with the **qop** attribute.

QoP setting	Description
auth	Authentication only.
auth-int	Authentication with integrity protection.
auth-conf	Authentication with integrity and privacy protection.

3.3.3. SASL policies

SASL policies provide fine-grain control over Hot Rod authentication mechanisms.

TIP

Data Grid cache authorization restricts access to caches based on roles and permissions. Configure cache authorization and then set **<no-anonymous value=false />** to allow anonymous login and delegate access logic to cache authorization.

Policy	Description	Default value
forward-secrecy	Use only SASL mechanisms that support forward secrecy between sessions. This means that breaking into one session does not automatically provide information for breaking into future sessions.	false
pass-credentials	Use only SASL mechanisms that require client credentials.	false
no-plain-text	Do not use SASL mechanisms that are susceptible to simple plain passive attacks.	false
no-active	Do not use SASL mechanisms that are susceptible to active, non-dictionary, attacks.	false
no-dictionary	Do not use SASL mechanisms that are susceptible to passive dictionary attacks.	false
no-anonymous	Do not use SASL mechanisms that accept anonymous logins.	true

SASL policy configuration

In the following configuration the Hot Rod endpoint uses the **GSSAPI** mechanism for authentication because it is the only mechanism that complies with all SASL policies:

XML

```
<server xmlns="urn:infinispan:server:14.0">
  <endpoints>
    <endpoint socket-binding="default"
              security-realm="default">
      <hotrod-connector>
        <authentication>
          <sasl mechanisms="PLAIN DIGEST-MD5 GSSAPI EXTERNAL"
                server-name="infinispan"
                qop="auth"
                policy="no-active no-plain-text"/>
        </authentication>
      </hotrod-connector>
      <rest-connector/>
    </endpoint>
  </endpoints>
</server>
```

JSON

```
{
  "server": {
    "endpoints" : {
      "endpoint" : {
        "socket-binding" : "default",
        "security-realm" : "default",
        "hotrod-connector" : {
          "authentication" : {
            "sasl" : {
              "server-name" : "infinispan",
              "mechanisms" : [ "PLAIN","DIGEST-MD5","GSSAPI","EXTERNAL" ],
              "qop" : [ "auth" ],
              "policy" : [ "no-active","no-plain-text" ]
            }
          }
        },
        "rest-connector" : ""
      }
    }
  }
}
```

YAML

```
server:
  endpoints:
    endpoint:
      socketBinding: "default"
      securityRealm: "default"
      hotrodConnector:
        authentication:
          sasl:
            serverName: "infinispan"
            mechanisms:
              - "PLAIN"
              - "DIGEST-MD5"
              - "GSSAPI"
              - "EXTERNAL"
            qop:
              - "auth"
            policy:
              - "no-active"
              - "no-plain-text"
      restConnector: ~
```

3.3.4. HTTP authentication mechanisms

Data Grid Server supports the following HTTP authentication mechanisms with REST endpoints:

Authentication mechanism	Description	Security realm type	Related details
BASIC	Uses credentials in plain-text format. You should use **BASIC** authentication with encrypted connections only.	Property realms and LDAP realms	Corresponds to the **Basic** HTTP authentication scheme and is similar to the **PLAIN** SASL mechanism.
DIGEST	Uses hashing algorithms and nonce values. REST connectors support **SHA-512**, **SHA-256** and **MD5** hashing algorithms.	Property realms and LDAP realms	Corresponds to the **Digest** HTTP authentication scheme and is similar to **DIGEST-*** SASL mechanisms.
SPNEGO	Uses Kerberos tickets and requires a Kerberos Domain Controller. You must add a corresponding **kerberos** server identity in the realm configuration. In most cases, you also specify an **ldap-realm** to provide user membership information.	Kerberos realms	Corresponds to the **Negotiate** HTTP authentication scheme and is similar to the **GSSAPI** and **GS2-KRB5** SASL mechanisms.
BEARER_TOKEN	Uses OAuth tokens and requires a **token-realm** configuration.	Token realms	Corresponds to the **Bearer** HTTP authentication scheme and is similar to **OAUTHBEARER** SASL mechanism.
CLIENT_CERT	Uses client certificates.	Trust store realms	Similar to the **EXTERNAL** SASL mechanism.

CHAPTER 4. CONFIGURING TLS/SSL ENCRYPTION

You can secure Data Grid Server connections using SSL/TLS encryption by configuring a keystore that contains public and private keys for Data Grid. You can also configure client certificate authentication if you require mutual TLS.

4.1. CONFIGURING DATA GRID SERVER KEYSTORES

Add keystores to Data Grid Server and configure it to present SSL/TLS certificates that verify its identity to clients. If a security realm contains TLS/SSL identities, it encrypts any connections to Data Grid Server endpoints that use that security realm.

Prerequisites

- Create a keystore that contains certificates, or certificate chains, for Data Grid Server.

Data Grid Server supports the following keystore formats: JKS, JCEKS, PKCS12/PFX and PEM. BKS, BCFKS, and UBER are also supported if the Bouncy Castle library is present. When using client hostname validation, according to the rules defined by the RFC 2818 specification, server certificates should include the **subjectAltName** extension of type **dNSName** and/or **iPAddress**.

IMPORTANT

In production environments, server certificates should be signed by a trusted Certificate Authority, either Root or Intermediate CA.

TIP

You can use PEM files as keystores if they contain both of the following:

- A private key in PKCS#1 or PKCS#8 format.
- One or more certificates.

You should also configure PEM file keystores with an empty password (**password=""**).

Procedure

1. Open your Data Grid Server configuration for editing.
2. Add the keystore that contains SSL/TLS identities for Data Grid Server to the **$RHDG_HOME/server/conf** directory.
3. Add a **server-identities** definition to the Data Grid Server security realm.
4. Specify the keystore file name with the **path** attribute.
5. Provide the keystore password and certificate alias with the **keystore-password** and **alias** attributes.
6. Save the changes to your configuration.

Next steps

Configure clients with a trust store so they can verify SSL/TLS identities for Data Grid Server.

Keystore configuration

XML

```
<server xmlns="urn:infinispan:server:14.0">
  <security>
    <security-realms>
      <security-realm name="default">
        <server-identities>
          <ssl>
            <!-- Adds a keystore that contains server certificates that provide SSL/TLS identities to clients.
-->
            <keystore path="server.p12"
                      relative-to="infinispan.server.config.path"
                      password="secret"
                      alias="my-server"/>
          </ssl>
        </server-identities>
      </security-realm>
    </security-realms>
  </security>
</server>
```

JSON

```
{
  "server": {
    "security": {
      "security-realms": [{
        "name": "default",
        "server-identities": {
          "ssl": {
            "keystore": {
              "alias": "my-server",
              "path": "server.p12",
              "password": "secret"
            }
          }
        }
      }]
    }
  }
}
```

YAML

```
server:
  security:
    securityRealms:
      - name: "default"
        serverIdentities:
          ssl:
            keystore:
```

```
          alias: "my-server"
          path: "server.p12"
          password: "secret"
```

Additional resources

- Configuring Hot Rod client encryption

4.1.1. Generating Data Grid Server keystores

Configure Data Grid Server to automatically generate keystores at startup.

IMPORTANT

Automatically generated keystores:

- Should not be used in production environments.
- Are generated whenever necessary; for example, while obtaining the first connection from a client.
- Contain certificates that you can use directly in Hot Rod clients.

Procedure

1. Open your Data Grid Server configuration for editing.
2. Include the **generate-self-signed-certificate-host** attribute for the **keystore** element in the server configuration.
3. Specify a hostname for the server certificate as the value.
4. Save the changes to your configuration.

Generated keystore configuration

XML

```
<server xmlns="urn:infinispan:server:14.0">
  <security>
    <security-realms>
      <security-realm name="generated-keystore">
        <server-identities>
          <ssl>
            <!-- Generates a keystore that includes a self-signed certificate with the specified hostname. --
>
            <keystore path="server.p12"
                      relative-to="infinispan.server.config.path"
                      password="secret"
                      alias="server"
                      generate-self-signed-certificate-host="localhost"/>
          </ssl>
        </server-identities>
      </security-realm>
    </security-realms>
  </security>
</server>
```

JSON

```
{
  "server": {
    "security": {
      "security-realms": [{
        "name": "generated-keystore",
        "server-identities": {
          "ssl": {
            "keystore": {
              "alias": "server",
              "generate-self-signed-certificate-host": "localhost",
              "path": "server.p12",
              "password": "secret"
            }
          }
        }
      }]
    }
  }
}
```

YAML

```
server:
  security:
    securityRealms:
      - name: "generated-keystore"
        serverIdentities:
          ssl:
            keystore:
              alias: "server"
              generateSelfSignedCertificateHost: "localhost"
              path: "server.p12"
              password: "secret"
```

4.1.2. Configuring TLS versions and cipher suites

When using SSL/TLS encryption to secure your deployment, you can configure Data Grid Server to use specific versions of the TLS protocol as well as specific cipher suites within the protocol.

Procedure

1. Open your Data Grid Server configuration for editing.
2. Add the **engine** element to the SSL configuration for Data Grid Server.
3. Configure Data Grid to use one or more TLS versions with the **enabled-protocols** attribute. Data Grid Server supports TLS version 1.2 and 1.3 by default. If appropriate you can set **TLSv1.3** only to restrict the security protocol for client connections. Data Grid does not recommend

enabling **TLSv1.1** because it is an older protocol with limited support and provides weak security. You should never enable any version of TLS older than 1.1.

> **WARNING**
>
> If you modify the SSL **engine** configuration for Data Grid Server you must explicitly configure TLS versions with the **enabled-protocols** attribute. Omitting the **enabled-protocols** attribute allows any TLS version.
>
> ```
> <engine enabled-protocols="TLSv1.3 TLSv1.2" />
> ```

4. Configure Data Grid to use one or more cipher suites with the **enabled-ciphersuites** attribute (for TLSv1.2 and below) and the **enabled-ciphersuites-tls13** attribute (for TLSv1.3). You must ensure that you set a cipher suite that supports any protocol features you plan to use; for example **HTTP/2 ALPN**.

5. Save the changes to your configuration.

SSL engine configuration

XML

```
<server xmlns="urn:infinispan:server:14.0">
  <security>
    <security-realms>
      <security-realm name="default">
        <server-identities>
          <ssl>
            <keystore path="server.p12"
                      relative-to="infinispan.server.config.path"
                      password="secret"
                      alias="server"/>
            <!-- Configures Data Grid Server to use specific TLS versions and cipher suites. -->
            <engine enabled-protocols="TLSv1.3 TLSv1.2"
                    enabled-
ciphersuites="TLS_DHE_RSA_WITH_AES_128_CBC_SHA,TLS_DHE_RSA_WITH_AES_128_CBC_
SHA256"
                    enabled-ciphersuites-
tls13="TLS_AES_256_GCM_SHA384:TLS_CHACHA20_POLY1305_SHA256:TLS_AES_128_GCM_S
HA256"/>
          </ssl>
        </server-identities>
      </security-realm>
    </security-realms>
  </security>
</server>
```

JSON

```
{
  "server": {
    "security": {
      "security-realms": [{
        "name": "default",
        "server-identities": {
          "ssl": {
            "keystore": {
              "alias": "server",
              "path": "server.p12",
              "password": "secret"
            },
            "engine": {
              "enabled-protocols": ["TLSv1.3"],
              "enabled-ciphersuites":
"TLS_DHE_RSA_WITH_AES_128_CBC_SHA,TLS_DHE_RSA_WITH_AES_128_CBC_SHA256",
              "enabled-ciphersuites-tls13":
"TLS_AES_256_GCM_SHA384:TLS_CHACHA20_POLY1305_SHA256:TLS_AES_128_GCM_SHA25
6"
            }
          }
        }
      }]
    }
  }
}
```

YAML

```
server:
  security:
    securityRealms:
      - name: "default"
        serverIdentities:
          ssl:
            keystore:
              alias: "server"
              path: "server.p12"
              password: "secret"
            engine:
              enabledProtocols:
                - "TLSv1.3"
              enabledCiphersuites: "TLS_AES_256_GCM_SHA384,TLS_AES_128_GCM_SHA256"
              enabledCiphersuitesTls13: "TLS_AES_256_GCM_SHA384"
```

4.2. CONFIGURING DATA GRID SERVER ON A SYSTEM WITH FIPS 140-2 COMPLIANT CRYPTOGRAPHY

FIPS (Federal Information Processing Standards) are standards and guidelines for US federal computer systems. Although FIPS are developed for use by the US federal government, many in the private sector voluntarily use these standards.

FIPS 140-2 defines security requirements for cryptographic modules. You can configure your Data Grid Server to use encryption ciphers that adhere to the FIPS 140-2 specification by using alternative JDK security providers.

Additional resources

- Java PKCS#11 cryptographic provider
- The Legion of the Bouncy Castle cryptographic provider

4.2.1. Configuring the PKCS11 cryptographic provider

You can configure the PKCS11 cryptographic provider by specifying the PKCS11 keystore with the **SunPKCS11-NSS-FIPS** provider.

Prerequisites

- Configure your system for FIPS mode. You can check if your system has FIPS Mode enabled by issuing the **fips-mode-setup --check** command in your Data Grid command-line Interface (CLI)
- Initialize the system-wide NSS database by using the **certutil** tool.
- Install the JDK with the **java.security** file configured to enable the **SunPKCS11** provider. This provider points to the NSS database and the SSL provider.
- Install a certificate in the NSS database.

NOTE

The OpenSSL provider requires a private key, but you cannot retrieve a private key from the PKCS#11 store. FIPS blocks the export of unencrypted keys from a FIPS-compliant cryptographic module, so you cannot use the OpenSSL provider for TLS when in FIPS mode. You can disable the OpenSSL provider at startup with the **-Dorg.infinispan.openssl=false** argument.

Procedure

1. Open your Data Grid Server configuration for editing.
2. Add a **server-identities** definition to the Data Grid Server security realm.
3. Specify the PKCS11 keystore with the **SunPKCS11-NSS-FIPS** provider.
4. Save the changes to your configuration.

Keystore configuration

XML

```
<server xmlns="urn:infinispan:server:14.0">
  <security>
    <security-realms>
      <security-realm name="default">
        <server-identities>
          <ssl>
          <!-- Adds a keystore that reads certificates from the NSS database. -->
          <keystore provider="SunPKCS11-NSS-FIPS" type="PKCS11"/>
        </ssl>
      </server-identities>
    </security-realm>
  </security-realms>
 </security>
</server>
```

JSON

```
{
 "server": {
  "security": {
   "security-realms": [{
    "name": "default",
    "server-identities": {
     "ssl": {
      "keystore": {
       "provider": "SunPKCS11-NSS-FIPS",
       "type": "PKCS11"
      }
     }
    }
   }]
  }
 }
}
```

YAML

```
server:
 security:
  securityRealms:
   - name: "default"
    serverIdentities:
     ssl:
      keystore:
       provider: "SunPKCS11-NSS-FIPS"
       type: "PKCS11"
```

4.2.2. Configuring the Bouncy Castle FIPS cryptographic provider

You can configure the Bouncy Castle FIPS (Federal Information Processing Standards) cryptographic provider in your Data Grid server’s configuration.

Prerequisites

- Configure your system for FIPS mode. You can check if your system has FIPS Mode enabled by issuing the **fips-mode-setup --check** command in your Data Grid command-line Interface (CLI).
- Create a keystore in BCFKS format that contains a certificate.

Procedure

1. Download the Bouncy Castle FIPS JAR file, and add the file to the **server/lib** directory of your Data Grid Server installation.
2. To install Bouncy Castle, issue the **install** command:

```
[disconnected]> install org.bouncycastle:bc-fips:1.0.2.3
```

3. Open your Data Grid Server configuration for editing.
4. Add a **server-identities** definition to the Data Grid Server security realm.
5. Specify the BCFKS keystore with the **BCFIPS** provider.
6. Save the changes to your configuration.

Keystore configuration

XML

```
<server xmlns="urn:infinispan:server:14.0">
  <security>
    <security-realms>
      <security-realm name="default">
        <server-identities>
          <ssl>
            <!-- Adds a keystore that reads certificates from the BCFKS keystore. -->
            <keystore path="server.bcfks" password="secret" alias="server" provider="BCFIPS"
type="BCFKS"/>
          </ssl>
        </server-identities>
      </security-realm>
    </security-realms>
  </security>
</server>
```

JSON

```
{
  "server": {
    "security": {
      "security-realms": [{
        "name": "default",
        "server-identities": {
          "ssl": {
            "keystore": {
              "path": "server.bcfks",
              "password": "secret",
              "alias": "server",
              "provider": "BCFIPS",
              "type": "BCFKS"
            }
          }
        }
      }]
      }
    }
  }
```

YAML

```
server:
  security:
    securityRealms:
      - name: "default"
        serverIdentities:
          ssl:
            keystore:
              path: "server.bcfks"
              password: "secret"
              alias: "server"
              provider: "BCFIPS"
              type: "BCFKS"
```

4.3. CONFIGURING CLIENT CERTIFICATE AUTHENTICATION

Configure Data Grid Server to use mutual TLS to secure client connections.

You can configure Data Grid to verify client identities from certificates in a trust store in two ways:

- Require a trust store that contains only the signing certificate, which is typically a Certificate Authority (CA). Any client that presents a certificate signed by the CA can connect to Data Grid.
- Require a trust store that contains all client certificates in addition to the signing certificate. Only clients that present a signed certificate that is present in the trust store can connect to Data Grid.

TIP

Alternatively to providing trust stores you can use shared system certificates.

Prerequisites

- Create a client trust store that contains either the CA certificate or all public certificates.
- Create a keystore for Data Grid Server and configure an SSL/TLS identity.

NOTE

PEM files can be used as trust stores provided they contain one or more certificates. These trust stores should be configured with an empty password: **password=""**.

Procedure

1. Open your Data Grid Server configuration for editing.
2. Add the **require-ssl-client-auth="true"** parameter to your **endpoints** configuration.
3. Add the client trust store to the **$RHDG_HOME/server/conf** directory.

4. Specify the **path** and **password** attributes for the **truststore** element in the Data Grid Server security realm configuration.

5. Add the **<truststore-realm/>** element to the security realm if you want Data Grid Server to authenticate each client certificate.

6. Save the changes to your configuration.

Next steps

- Set up authorization with client certificates in the Data Grid Server configuration if you control access with security roles and permissions.

- Configure clients to negotiate SSL/TLS connections with Data Grid Server.

Client certificate authentication configuration

XML

```
<server xmlns="urn:infinispan:server:14.0">
  <security>
    <security-realms>
      <security-realm name="trust-store-realm">
        <server-identities>
          <ssl>
            <!-- Provides an SSL/TLS identity with a keystore that
                contains server certificates. -->
            <keystore path="server.p12"
                        relative-to="infinispan.server.config.path"
                        keystore-password="secret"
                        alias="server"/>
            <!-- Configures a trust store that contains client certificates
                or part of a certificate chain. -->
            <truststore path="trust.p12"
                          relative-to="infinispan.server.config.path"
                          password="secret"/>
          </ssl>
        </server-identities>
        <!-- Authenticates client certificates against the trust store. If you configure this, the trust store
must contain the public certificates for all clients. -->
        <truststore-realm/>
      </security-realm>
    </security-realms>
  </security>
  <endpoints>
    <endpoint socket-binding="default"
              security-realm="trust-store-realm"
              require-ssl-client-auth="true">
      <hotrod-connector>
        <authentication>
          <sasl mechanisms="EXTERNAL"
                server-name="infinispan"
                qop="auth"/>
        </authentication>
      </hotrod-connector>
```

```
        <rest-connector>
          <authentication mechanisms="CLIENT_CERT"/>
        </rest-connector>
      </endpoint>
    </endpoints>
</server>
```

JSON

```
{
  "server": {
    "security": {
      "security-realms": [{
        "name": "trust-store-realm",
        "server-identities": {
          "ssl": {
            "keystore": {
              "path": "server.p12",
              "relative-to": "infinispan.server.config.path",
              "keystore-password": "secret",
              "alias": "server"
            },
            "truststore": {
              "path": "trust.p12",
              "relative-to": "infinispan.server.config.path",
              "password": "secret"
            }
          }
        },
        "truststore-realm": {}
      }]
    },
    "endpoints": [{
      "socket-binding": "default",
      "security-realm": "trust-store-realm",
      "require-ssl-client-auth": "true",
      "connectors": {
        "hotrod": {
          "hotrod-connector": {
            "authentication": {
              "sasl": {
                "mechanisms": "EXTERNAL",
                "server-name": "infinispan",
                "qop": "auth"
              }
            }
          },
          "rest": {
            "rest-connector": {
              "authentication": {
                "mechanisms": "CLIENT_CERT"
              }
            }
          }
        }
      }
    }]
  }
}
```

YAML

```
server:
  security:
    securityRealms:
      - name: "trust-store-realm"
        serverIdentities:
          ssl:
            keystore:
              path: "server.p12"
              relative-to: "infinispan.server.config.path"
              keystore-password: "secret"
              alias: "server"
            truststore:
              path: "trust.p12"
              relative-to: "infinispan.server.config.path"
              password: "secret"
        truststoreRealm: ~
  endpoints:
    socketBinding: "default"
    securityRealm: "trust-store-realm"
    requireSslClientAuth: "true"
    connectors:
      - hotrod:
          hotrodConnector:
            authentication:
              sasl:
                mechanisms: "EXTERNAL"
                serverName: "infinispan"
                qop: "auth"
      - rest:
          restConnector:
            authentication:
              mechanisms: "CLIENT_CERT"
```

Additional resources

- Configuring Hot Rod client encryption
- Using Shared System Certificates (Red Hat Enterprise Linux 7 Security Guide)

4.4. CONFIGURING AUTHORIZATION WITH CLIENT CERTIFICATES

Enabling client certificate authentication means you do not need to specify Data Grid user credentials in client configuration, which means you must associate roles with the Common Name (CN) field in the client certificate(s).

Prerequisites

- Provide clients with a Java keystore that contains either their public certificates or part of the certificate chain, typically a public CA certificate.
- Configure Data Grid Server to perform client certificate authentication.

Procedure

1. Open your Data Grid Server configuration for editing.
2. Enable the **common-name-role-mapper** in the security authorization configuration.
3. Assign the Common Name (**CN**) from the client certificate a role with the appropriate permissions.
4. Save the changes to your configuration.

Client certificate authorization configuration

XML

```
<infinispan>
  <cache-container name="certificate-authentication" statistics="true">
    <security>
      <authorization>
        <!-- Declare a role mapper that associates the common name (CN) field in client certificate trust
stores with authorization roles. -->
        <common-name-role-mapper/>
        <!-- In this example, if a client certificate contains `CN=Client1` then clients with matching
certificates get ALL permissions. -->
        <role name="Client1" permissions="ALL"/>
      </authorization>
    </security>
  </cache-container>
</infinispan>
```

JSON

```
{
  "infinispan": {
    "cache-container": {
      "name": "certificate-authentication",
      "security": {
        "authorization": {
          "common-name-role-mapper": null,
          "roles": {
            "Client1": {
              "role": {
                "permissions": "ALL"
              }
            }
          }
        }
      }
      }
    }
  }
```

YAML

```
infinispan:
  cacheContainer:
    name: "certificate-authentication"
    security:
      authorization:
        commonNameRoleMapper: ~
        roles:
          Client1:
            role:
              permissions:
                - "ALL"
```

CHAPTER 5. STORING DATA GRID SERVER CREDENTIALS IN KEYSTORES

External services require credentials to authenticate with Data Grid Server. To protect sensitive text strings such as passwords, add them to a credential keystore rather than directly in Data Grid Server configuration files.

You can then configure Data Grid Server to decrypt passwords for establishing connections with services such as databases or LDAP directories.

IMPORTANT

Plain-text passwords in **$RHDG_HOME/server/conf** are unencrypted. Any user account with read access to the host filesystem can view plain-text passwords.

While credential keystores are password-protected store encrypted passwords, any user account with write access to the host filesystem can tamper with the keystore itself.

To completely secure Data Grid Server credentials, you should grant read-write access only to user accounts that can configure and run Data Grid Server.

5.1. SETTING UP CREDENTIAL KEYSTORES

Create keystores that encrypt credential for Data Grid Server access.

A credential keystore contains at least one alias that is associated with an encrypted password. After you create a keystore, you specify the alias in a connection configuration such as a database connection pool. Data Grid Server then decrypts the password for that alias from the keystore when the service attempts authentication.

You can create as many credential keystores with as many aliases as required.

NOTE

As a security best practice, keystores should be readable only by the user who runs the process for Data Grid Server.

Procedure

1. Open a terminal in **$RHDG_HOME**.
2. Create a keystore and add credentials to it with the **credentials** command.

   TIP

   By default, keystores are of type PKCS12. Run **help credentials** for details on changing keystore defaults.

   The following example shows how to create a keystore that contains an alias of "dbpassword" for the password "changeme". When you create a keystore you also specify a password to access the keystore with the **-p** argument.

   Linux

```
bin/cli.sh credentials add dbpassword -c changeme -p "secret1234!"
```

Microsoft Windows

```
bin\cli.bat credentials add dbpassword -c changeme -p "secret1234!"
```

3. Check that the alias is added to the keystore.

```
bin/cli.sh credentials ls -p "secret1234!"
dbpassword
```

4. Open your Data Grid Server configuration for editing.
5. Configure Data Grid to use the credential keystore.
   a. Add a **credential-stores** section to the **security** configuration.
   b. Specify the name and location of the credential keystore.
   c. Specify the password to access the credential keystore with the **clear-text-credential** configuration.

   NOTE

   Instead of adding a clear-text password for the credential keystore to your Data Grid Server configuration you can use an external command or masked password for additional security.

   You can also use a password in one credential store as the master password for another credential store.

6. Reference the credential keystore in configuration that Data Grid Server uses to connect with an external system such as a datasource or LDAP server.
   a. Add a **credential-reference** section.
   b. Specify the name of the credential keystore with the **store** attribute.
   c. Specify the password alias with the **alias** attribute.

   TIP

   Attributes in the **credential-reference** configuration are optional.

   - **store** is required only if you have multiple keystores.
   - **alias** is required only if the keystore contains multiple password aliases.

7. Save the changes to your configuration.

5.2. SECURING PASSWORDS FOR CREDENTIAL KEYSTORES

Data Grid Server requires a password to access credential keystores. You can add that password to Data Grid Server configuration in clear text or, as an added layer of security, you can use an external command for the password or you can mask the password.

Prerequisites

- Set up a credential keystore for Data Grid Server.

Procedure

Do one of the following:

- Use the **credentials mask** command to obscure the password, for example:

  ```
  bin/cli.sh credentials mask -i 100 -s pepper99 "secret1234!"
  ```

  Masked passwords use Password Based Encryption (PBE) and must be in the following format in your Data Grid Server configuration: <MASKED_VALUE;SALT;ITERATION>.

- Use an external command that provides the password as standard output.
  An external command can be any executable, such as a shell script or binary, that uses **java.lang.Runtime#exec(java.lang.String)**.
  If the command requires parameters, provide them as a space-separated list of strings.

5.3. CREDENTIAL KEYSTORE CONFIGURATION

You can add credential keystores to Data Grid Server configuration and use clear-text passwords, masked passwords, or external commands that supply passwords.

Credential keystore with a clear text password

XML

```
<server xmlns="urn:infinispan:server:14.0">
  <security>
    <credential-stores>
      <credential-store name="credentials" path="credentials.pfx">
        <clear-text-credential clear-text="secret1234!"/>
      </credential-store>
    </credential-stores>
  </security>
</server>
```

JSON

```
{
  "server": {
    "security": {
      "credential-stores": [{
        "name": "credentials",
        "path": "credentials.pfx",
        "clear-text-credential": {
          "clear-text": "secret1234!"
        }
```

```
      }]
    }
  }
}
```

YAML

```
server:
  security:
    credentialStores:
      - name: credentials
        path: credentials.pfx
        clearTextCredential:
          clearText: "secret1234!"
```

Credential keystore with a masked password

XML

```
<server xmlns="urn:infinispan:server:14.0">
  <security>
    <credential-stores>
      <credential-store name="credentials"
                        path="credentials.pfx">
        <masked-credential masked="1oTMDZ5JQj6DVepJviXMnX;pepper99;100"/>
      </credential-store>
    </credential-stores>
  </security>
</server>
```

JSON

```
{
  "server": {
    "security": {
      "credential-stores": [{
        "name": "credentials",
        "path": "credentials.pfx",
        "masked-credential": {
          "masked": "1oTMDZ5JQj6DVepJviXMnX;pepper99;100"
        }
      }]
    }
  }
}
```

YAML

```
server:
  security:
    credentialStores:
      - name: credentials
```

```
        path: credentials.pfx
        maskedCredential:
          masked: "1oTMDZ5JQj6DVepJviXMnX;pepper99;100"
```

External command passwords

XML

```xml
<server xmlns="urn:infinispan:server:14.0">
  <security>
    <credential-stores>
      <credential-store name="credentials"
                        path="credentials.pfx">
        <command-credential command="/path/to/executable.sh arg1 arg2"/>
      </credential-store>
    </credential-stores>
  </security>
</server>
```

JSON

```json
{
  "server": {
    "security": {
      "credential-stores": [{
        "name": "credentials",
        "path": "credentials.pfx",
        "command-credential": {
          "command": "/path/to/executable.sh arg1 arg2"
        }
      }]
    }
  }
}
```

YAML

```yaml
server:
  security:
    credentialStores:
      - name: credentials
        path: credentials.pfx
        commandCredential:
          command: "/path/to/executable.sh arg1 arg2"
```

5.4. CREDENTIAL KEYSTORE REFERENCES

After you add credential keystores to Data Grid Server you can reference them in connection configurations.

Datasource connections

XML

```
<server xmlns="urn:infinispan:server:14.0">
  <security>
    <credential-stores>
      <credential-store name="credentials"
                        path="credentials.pfx">
        <clear-text-credential clear-text="secret1234!"/>
      </credential-store>
    </credential-stores>
  </security>
  <data-sources>
    <data-source name="postgres"
                 jndi-name="jdbc/postgres">
      <!-- Specifies the database username in the connection factory. -->
      <connection-factory driver="org.postgresql.Driver"
                          username="dbuser"
                          url="${org.infinispan.server.test.postgres.jdbcUrl}">
        <!-- Specifies the credential keystore that contains an encrypted password and the alias for it. --
>
        <credential-reference store="credentials"
                              alias="dbpassword"/>
      </connection-factory>
      <connection-pool max-size="10"
                       min-size="1"
                       background-validation="1000"
                       idle-removal="1"
                       initial-size="1"
                       leak-detection="10000"/>
    </data-source>
  </data-sources>
</server>
```

JSON

```
{
  "server": {
    "security": {
      "credential-stores": [{
        "name": "credentials",
        "path": "credentials.pfx",
        "clear-text-credential": {
          "clear-text": "secret1234!"
        }
      }],
      "data-sources": [{
        "name": "postgres",
        "jndi-name": "jdbc/postgres",
        "connection-factory": {
          "driver": "org.postgresql.Driver",
          "username": "dbuser",
          "url": "${org.infinispan.server.test.postgres.jdbcUrl}",
          "credential-reference": {
            "store": "credentials",
            "alias": "dbpassword"
          }
        }
      }]
    }
  }
}
```

YAML

```
server:
  security:
    credentialStores:
      - name: credentials
        path: credentials.pfx
        clearTextCredential:
          clearText: "secret1234!"
  dataSources:
    - name: postgres
      jndiName: jdbc/postgres
      connectionFactory:
        driver: org.postgresql.Driver
        username: dbuser
        url: '${org.infinispan.server.test.postgres.jdbcUrl}'
        credentialReference:
          store: credentials
          alias: dbpassword
```

LDAP connections

XML

```
<server xmlns="urn:infinispan:server:14.0">
  <security>
    <credential-stores>
      <credential-store name="credentials"
                        path="credentials.pfx">
        <clear-text-credential clear-text="secret1234!"/>
      </credential-store>
    </credential-stores>
    <security-realms>
      <security-realm name="default">
        <!-- Specifies the LDAP principal in the connection factory. -->
        <ldap-realm name="ldap"
                    url="ldap://my-ldap-server:10389"
                    principal="uid=admin,ou=People,dc=infinispan,dc=org">
          <!-- Specifies the credential keystore that contains an encrypted password and the alias for it. -
->
          <credential-reference store="credentials"
                                alias="ldappassword"/>
        </ldap-realm>
      </security-realm>
    </security-realms>
  </security>
</server>
```

JSON

```
{
  "server": {
    "security": {
      "credential-stores": [{
        "name": "credentials",
        "path": "credentials.pfx",
        "clear-text-credential": {
          "clear-text": "secret1234!"
        }
      }],
      "security-realms": [{
        "name": "default",
        "ldap-realm": {
          "name": "ldap",
          "url": "ldap://my-ldap-server:10389",
          "principal": "uid=admin,ou=People,dc=infinispan,dc=org",
          "credential-reference": {
            "store": "credentials",
            "alias": "ldappassword"
          }
        }
      }]
    }
  }
}
```

YAML

```
server:
  security:
    credentialStores:
      - name: credentials
        path: credentials.pfx
        clearTextCredential:
          clearText: "secret1234!"
    securityRealms:
      - name: "default"
        ldapRealm:
          name: ldap
          url: 'ldap://my-ldap-server:10389'
          principal: 'uid=admin,ou=People,dc=infinispan,dc=org'
          credentialReference:
            store: credentials
            alias: ldappassword
```

CHAPTER 6. ENCRYPTING CLUSTER TRANSPORT

Secure cluster transport so that nodes communicate with encrypted messages. You can also configure Data Grid clusters to perform certificate authentication so that only nodes with valid identities can join.

6.1. SECURING CLUSTER TRANSPORT WITH TLS IDENTITIES

Add SSL/TLS identities to a Data Grid Server security realm and use them to secure cluster transport. Nodes in the Data Grid Server cluster then exchange SSL/TLS certificates to encrypt JGroups messages, including RELAY messages if you configure cross-site replication.

Prerequisites

- Install a Data Grid Server cluster.

Procedure

1. Create a TLS keystore that contains a single certificate to identify Data Grid Server.
   You can also use a PEM file if it contains a private key in PKCS#1 or PKCS#8 format, a certificate, and has an empty password: **password=""**.

   NOTE

   If the certificate in the keystore is not signed by a public certificate authority (CA) then you must also create a trust store that contains either the signing certificate or the public key.

2. Add the keystore to the **$RHDG_HOME/server/conf** directory.

3. Add the keystore to a new security realm in your Data Grid Server configuration.

   IMPORTANT

   You should create dedicated keystores and security realms so that Data Grid Server endpoints do not use the same security realm as cluster transport.

```
<server xmlns="urn:infinispan:server:14.0">
  <security>
    <security-realms>
      <security-realm name="cluster-transport">
        <server-identities>
          <ssl>
            <!-- Adds a keystore that contains a certificate that provides SSL/TLS identity to
encrypt cluster transport. -->
            <keystore path="server.pfx"
                      relative-to="infinispan.server.config.path"
                      password="secret"
                      alias="server"/>
          </ssl>
        </server-identities>
      </security-realm>
    </security-realms>
  </security>
</server>
```

4. Configure cluster transport to use the security realm by specifying the name of the security realm with the **server:security-realm** attribute.

```
<infinispan>
  <cache-container>
    <transport server:security-realm="cluster-transport"/>
  </cache-container>
</infinispan>
```

Verification

When you start Data Grid Server, the following log message indicates that the cluster is using the security realm for cluster transport:

```
[org.infinispan.SERVER] ISPN080060: SSL Transport using realm <security_realm_name>
```

6.2. JGROUPS ENCRYPTION PROTOCOLS

To secure cluster traffic, you can configure Data Grid nodes to encrypt JGroups message payloads with secret keys.

Data Grid nodes can obtain secret keys from either:

- The coordinator node (asymmetric encryption).
- A shared keystore (symmetric encryption).

Retrieving secret keys from coordinator nodes

You configure asymmetric encryption by adding the **ASYM_ENCRYPT** protocol to a JGroups stack in your Data Grid configuration. This allows Data Grid clusters to generate and distribute secret keys.

IMPORTANT

When using asymmetric encryption, you should also provide keystores so that nodes can perform certificate authentication and securely exchange secret keys. This protects your cluster from man-in-the-middle (MitM) attacks.

Asymmetric encryption secures cluster traffic as follows:

1. The first node in the Data Grid cluster, the coordinator node, generates a secret key.
2. A joining node performs certificate authentication with the coordinator to mutually verify identity.
3. The joining node requests the secret key from the coordinator node. That request includes the public key for the joining node.
4. The coordinator node encrypts the secret key with the public key and returns it to the joining node.

5. The joining node decrypts and installs the secret key.

6. The node joins the cluster, encrypting and decrypting messages with the secret key.

Retrieving secret keys from shared keystores

You configure symmetric encryption by adding the **SYM_ENCRYPT** protocol to a JGroups stack in your Data Grid configuration. This allows Data Grid clusters to obtain secret keys from keystores that you provide.

1. Nodes install the secret key from a keystore on the Data Grid classpath at startup.

2. Node join clusters, encrypting and decrypting messages with the secret key.

Comparison of asymmetric and symmetric encryption

ASYM_ENCRYPT with certificate authentication provides an additional layer of encryption in comparison with **SYM_ENCRYPT**. You provide keystores that encrypt the requests to coordinator nodes for the secret key. Data Grid automatically generates that secret key and handles cluster traffic, while letting you specify when to generate secret keys. For example, you can configure clusters to generate new secret keys when nodes leave. This ensures that nodes cannot bypass certificate authentication and join with old keys.

SYM_ENCRYPT, on the other hand, is faster than **ASYM_ENCRYPT** because nodes do not need to exchange keys with the cluster coordinator. A potential drawback to **SYM_ENCRYPT** is that there is no configuration to automatically generate new secret keys when cluster membership changes. Users are responsible for generating and distributing the secret keys that nodes use to encrypt cluster traffic.

6.3. SECURING CLUSTER TRANSPORT WITH ASYMMETRIC ENCRYPTION

Configure Data Grid clusters to generate and distribute secret keys that encrypt JGroups messages.

Procedure

1. Create a keystore with certificate chains that enables Data Grid to verify node identity.

2. Place the keystore on the classpath for each node in the cluster.
   For Data Grid Server, you put the keystore in the $RHDG_HOME directory.

3. Add the **SSL_KEY_EXCHANGE** and **ASYM_ENCRYPT** protocols to a JGroups stack in your Data Grid configuration, as in the following example:

```
<infinispan>
  <jgroups>
    <!-- Creates a secure JGroups stack named "encrypt-tcp" that extends the default TCP
stack. -->
    <stack name="encrypt-tcp" extends="tcp">
      <!-- Adds a keystore that nodes use to perform certificate authentication. -->
      <!-- Uses the stack.combine and stack.position attributes to insert
SSL_KEY_EXCHANGE into the default TCP stack after VERIFY_SUSPECT2. -->
      <SSL_KEY_EXCHANGE keystore_name="mykeystore.jks"
                        keystore_password="changeit"
                        stack.combine="INSERT_AFTER"
                        stack.position="VERIFY_SUSPECT2"/>
      <!-- Configures ASYM_ENCRYPT -->
      <!-- Uses the stack.combine and stack.position attributes to insert ASYM_ENCRYPT into
the default TCP stack before pbcast.NAKACK2. -->
      <!-- The use_external_key_exchange = "true" attribute configures nodes to use the
`SSL_KEY_EXCHANGE` protocol for certificate authentication. -->
      <ASYM_ENCRYPT asym_keylength="2048"
               asym_algorithm="RSA"
               change_key_on_coord_leave = "false"
               change_key_on_leave = "false"
               use_external_key_exchange = "true"
               stack.combine="INSERT_BEFORE"
               stack.position="pbcast.NAKACK2"/>
    </stack>
  </jgroups>
  <cache-container name="default" statistics="true">
    <!-- Configures the cluster to use the JGroups stack. -->
    <transport cluster="${infinispan.cluster.name}"
            stack="encrypt-tcp"
            node-name="${infinispan.node.name:}"/>
  </cache-container>
</infinispan>
```

Verification

When you start your Data Grid cluster, the following log message indicates that the cluster is using the secure JGroups stack:

```
[org.infinispan.CLUSTER] ISPN000078: Starting JGroups channel cluster with stack
<encrypted_stack_name>
```

Data Grid nodes can join the cluster only if they use **ASYM_ENCRYPT** and can obtain the secret key from the coordinator node. Otherwise the following message is written to Data Grid logs:

```
[org.jgroups.protocols.ASYM_ENCRYPT] <hostname>: received message without encrypt header
from <hostname>; dropping it
```

Additional resources

- JGroups 4 Manual
- JGroups 4.2 Schema

6.4. SECURING CLUSTER TRANSPORT WITH SYMMETRIC ENCRYPTION

Configure Data Grid clusters to encrypt JGroups messages with secret keys from keystores that you provide.

Procedure

1. Create a keystore that contains a secret key.
2. Place the keystore on the classpath for each node in the cluster.
   For Data Grid Server, you put the keystore in the $RHDG_HOME directory.

3. Add the **SYM_ENCRYPT** protocol to a JGroups stack in your Data Grid configuration.

```
<infinispan>
  <jgroups>
    <!-- Creates a secure JGroups stack named "encrypt-tcp" that extends the default TCP stack. -->
    <stack name="encrypt-tcp" extends="tcp">
      <!-- Adds a keystore from which nodes obtain secret keys. -->
      <!-- Uses the stack.combine and stack.position attributes to insert SYM_ENCRYPT into the
default TCP stack after VERIFY_SUSPECT2. -->
      <SYM_ENCRYPT keystore_name="myKeystore.p12"
                   keystore_type="PKCS12"
                   store_password="changeit"
                   key_password="changeit"
                   alias="myKey"
                   stack.combine="INSERT_AFTER"
                   stack.position="VERIFY_SUSPECT2"/>
    </stack>
  </jgroups>
  <cache-container name="default" statistics="true">
    <!-- Configures the cluster to use the JGroups stack. -->
    <transport cluster="${infinispan.cluster.name}"
               stack="encrypt-tcp"
               node-name="${infinispan.node.name:}"/>
  </cache-container>
</infinispan>
```

Verification

When you start your Data Grid cluster, the following log message indicates that the cluster is using the secure JGroups stack:

```
[org.infinispan.CLUSTER] ISPN000078: Starting JGroups channel cluster with stack
<encrypted_stack_name>
```

Data Grid nodes can join the cluster only if they use **SYM_ENCRYPT** and can obtain the secret key from the shared keystore. Otherwise the following message is written to Data Grid logs:

```
[org.jgroups.protocols.SYM_ENCRYPT] <hostname>: received message without encrypt header from
<hostname>; dropping it
```

Additional resources

- JGroups 4 Manual
- JGroups 4.2 Schema

CHAPTER 7. DATA GRID PORTS AND PROTOCOLS

As Data Grid distributes data across your network and can establish connections for external client requests, you should be aware of the ports and protocols that Data Grid uses to handle network traffic.

If run Data Grid as a remote server then you might need to allow remote clients through your firewall. Likewise, you should adjust ports that Data Grid nodes use for cluster communication to prevent conflicts or network issues.

7.1. DATA GRID SERVER PORTS AND PROTOCOLS

Data Grid Server provides network endpoints that allow client access with different protocols.

Port	Protocol	Description
11222	TCP	Hot Rod and REST
11221	TCP	Memcached (disabled by default)

Single port

Data Grid Server exposes multiple protocols through a single TCP port, **11222**. Handling multiple protocols with a single port simplifies configuration and reduces management complexity when deploying Data Grid clusters. Using a single port also enhances security by minimizing the attack surface on the network.

Data Grid Server handles HTTP/1.1, HTTP/2, and Hot Rod protocol requests from clients via the single port in different ways.

HTTP/1.1 upgrade headers

Client requests can include the **HTTP/1.1 upgrade** header field to initiate HTTP/1.1 connections with Data Grid Server. Client applications can then send the **Upgrade: protocol** header field, where **protocol** is a server endpoint.

Application-Layer Protocol Negotiation (ALPN)/Transport Layer Security (TLS)

Client requests include Server Name Indication (SNI) mappings for Data Grid Server endpoints to negotiate protocols over a TLS connection.

NOTE

Applications must use a TLS library that supports the ALPN extension. Data Grid uses WildFly OpenSSL bindings for Java.

Automatic Hot Rod detection

Client requests that include Hot Rod headers automatically route to Hot Rod endpoints.

7.1.1. Configuring network firewalls for Data Grid traffic

Adjust firewall rules to allow traffic between Data Grid Server and client applications.

Procedure

On Red Hat Enterprise Linux (RHEL) workstations, for example, you can allow traffic to port **11222** with firewalld as follows:

```
# firewall-cmd --add-port=11222/tcp --permanent
success
# firewall-cmd --list-ports | grep 11222
11222/tcp
```

To configure firewall rules that apply across a network, you can use the nftables utility.

Reference

- Using and configuring firewalld
- Getting started with nftables

7.2. TCP AND UDP PORTS FOR CLUSTER TRAFFIC

Data Grid uses the following ports for cluster transport messages:

Default Port	Protocol	Description
7800	TCP/UDP	JGroups cluster bind port
46655	UDP	JGroups multicast

Cross-site replication

Data Grid uses the following ports for the JGroups RELAY2 protocol:

7900

For Data Grid clusters running on OpenShift.

7800

If using UDP for traffic between nodes and TCP for traffic between clusters.

7801

If using TCP for traffic between nodes and TCP for traffic between clusters.