Security Guide for {brandname} 10.0

Table of Contents

Chapter 1. Security

Security within {brandname} is implemented at several layers:

- within the core library, to provide coarse-grained access control to CacheManagers, Caches and data
- over remote protocols, to obtain credentials from remote clients and to secure the transport using encryption
- between nodes in a cluster, so that only authorized nodes can join and to secure the transport using encryption

In order to maximize compatibility and integration, {brandname} uses widespread security standards where possible and appropriate, such as X.509 certificates, SSL/TLS encryption and Kerberos/GSSAPI. Also, to avoid pulling in any external dependencies and to increase the ease of integration with third party libraries and containers, the implementation makes use of any facilities provided by the standard Java security libraries (JAAS, JSSE, JCA, JCE, SASL, etc). For this reason, the {brandname} core library only provides interfaces and a set of basic implementations.

1.1. Embedded Security

Applications interact with {brandname} using its API within the same JVM. The two main components which are exposed by the {brandname} API are CacheManagers and Caches. If an application wants to interact with a secured CacheManager and Cache, it should provide an identity which {brandname}'s security layer will validate against a set of required roles and permissions. If the identity provided by the user application has sufficient permissions, then access will be granted, otherwise an exception indicating a security violation will be thrown. The identity is represented by the javax.security.auth.Subject class which is a wrapper around multiple Principals, e.g. a user and all the groups it belongs to. Since the Principal name is dependent on the owning system (e.g. a Distinguished Name in LDAP), {brandname} needs to be able to map Principal names to roles. Roles, in turn, represent one or more permissions. The following diagram shows the relationship between the various elements:

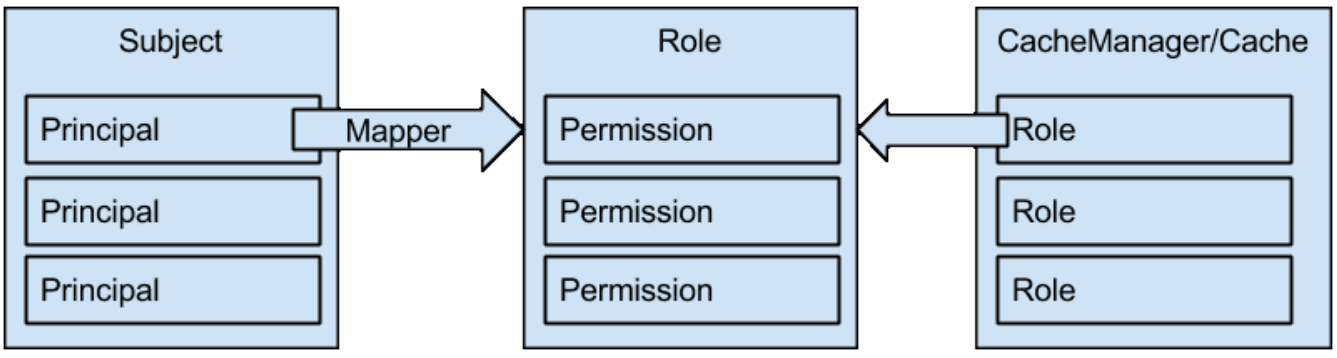

Figure 1. Roles/Permissions mapping

1.1.1. Embedded Permissions

Access to a cache manager or a cache is controlled by using a list of required permissions. Permissions are concerned with the type of action that is performed on one of the above entities

and not with the type of data being manipulated. Some of these permissions can be narrowed to specifically named entities, where applicable (e.g. a named cache). Depending on the type of entity, there are different types of permission available:

Cache Manager permissions

- CONFIGURATION (defineConfiguration): whether a new cache configuration can be defined
- LISTEN (addListener): whether listeners can be registered against a cache manager
- LIFECYCLE (stop): whether the cache manager can be stopped
- ALL: a convenience permission which includes all of the above

Cache permissions

- READ (get, contains): whether entries can be retrieved from the cache
- WRITE (put, putIfAbsent, replace, remove, evict): whether data can be written/replaced/removed/evicted from the cache
- EXEC (distexec, streams): whether code execution can be run against the cache
- LISTEN (addListener): whether listeners can be registered against a cache
- BULK_READ (keySet, values, entrySet, query): whether bulk retrieve operations can be executed
- BULK_WRITE (clear, putAll): whether bulk write operations can be executed
- LIFECYCLE (start, stop): whether a cache can be started / stopped
- ADMIN (getVersion, addInterceptor*, removeInterceptor, getInterceptorChain, getEvictionManager, getComponentRegistry, getDistributionManager, getAuthorizationManager, evict, getRpcManager, getCacheConfiguration, getCacheManager, getInvocationContextContainer, setAvailability, getDataContainer, getStats, getXAResource): whether access to the underlying components/internal structures is allowed
- ALL: a convenience permission which includes all of the above
- ALL_READ: combines READ and BULK_READ
- ALL_WRITE: combines WRITE and BULK_WRITE

Some permissions might need to be combined with others in order to be useful. For example, suppose you want to allow only "supervisors" to be able to run stream operations, while "standard" users can only perform puts and gets, you would define the following mappings:

```
<role name="standard" permission="READ WRITE" />
<role name="supervisors" permission="READ WRITE EXEC BULK"/>
```

1.1.2. Embedded API

When a DefaultCacheManager has been constructed with security enabled using either the programmatic or declarative configuration, it returns a SecureCache which will check the security context before invoking any operations on the underlying caches. A SecureCache also makes sure that applications cannot retrieve lower-level insecure objects (such as DataContainer). In Java,

executing code with a specific identity usually means wrapping the code to be executed within a PrivilegedAction:

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

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

If you are using Java 8, the above call can be simplified to:

```
Security.doAs(mySubject, PrivilegedAction<String>() -> cache.put("key", "value"));
```

Notice the use of Security.doAs() in place of the typical Subject.doAs(). While in {brandname} you can use either, unless you really need to modify the AccessControlContext for reasons specific to your application's security model, using Security.doAs() provides much better performance. If you need the current Subject, use the following:

```
Security.getSubject();
```

which will automatically retrieve the Subject either from the {brandname}'s context or from the AccessControlContext.

{brandname} also fully supports running under a full-blown SecurityManager. The {brandname} distribution contains an example security.policy file which you should customize with the appropriate paths before supplying it to your JVM.

1.1.3. Embedded Configuration

There are two levels of configuration: global and per-cache. The global configuration defines the set of roles/permissions mappings while each cache can decide whether to enable authorization checks and the required roles.

Programmatic

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

Declarative

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

</infinispan>
```

Role Mappers

In order to convert the Principals in a Subject into a set of roles to be used when authorizing, a suitable PrincipalRoleMapper must be specified in the global configuration. {brandname} comes

with 3 mappers and also allows you to provide a custom one:

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

1.2. Security Audit

{brandname} offers a pluggable audit logger which tracks whether a cache or a cache manager operation was allowed or denied. The audit logger is configured at the cache container authorization level:

Programmatic

```
GlobalConfigurationBuilder global = new GlobalConfigurationBuilder();
global
   .authorization()
      .auditLogger(new LoggingAuditLogger());
```

Declarative

```
<infinispan>
   <cache-container default-cache="secured">
      <security>
         <authorization audit-logger="org.infinispan.security.impl.LoggingAuditLogger
">
            ...
         </authorization>
      </security>
      ...
   </cache-container>
</infinispan>
```

In embedded mode the default audit logger is org.infinispan.security.impl.NullAuditLogger which does nothing. {brandname} also comes with the org.infinispan.security.impl.LoggingAuditLogger which outputs audit logs through the available logging framework (e.g. Log4J) at level TRACE and category AUDIT. These logs look like:

```
[ALLOW|DENY] user READ cache[defaultCache]
```

Using an appropriate logging appender it is possible to send the AUDIT category either to a log file, a JMS queue, a database, etc. The user which is included in the log above is the name of the first non-java.security.acl.Group principal in the Subject.

1.3. Cluster security

JGroups can be configured so that nodes need to authenticate each other when joining / merging. The authentication uses SASL and is setup by adding the SASL protocol to your JGroups XML configuration above the GMS protocol, as follows:

```
<SASL mech="DIGEST-MD5"
    client_name="node_user"
    client_password="node_password"
    server_callback_handler_class=
"org.example.infinispan.security.JGroupsSaslServerCallbackHandler"
    client_callback_handler_class=
"org.example.infinispan.security.JGroupsSaslClientCallbackHandler"
    sasl_props="com.sun.security.sasl.digest.realm=test_realm" />
```

In the above example, the SASL mech will be DIGEST-MD5. Each node will need to declare the user and password it will use when joining the cluster. The behaviour of a node differs depending on whether it is the coordinator or any other node. The coordinator acts as the SASL server, whereas joining/merging nodes act as SASL clients. Therefore two different CallbackHandlers are required, the server_callback_handler_class will be used by the coordinator, and the client_callback_handler_class will be used by the other nodes. The SASL protocol in JGroups is only concerned with the authentication process. If you wish to implement node authorization, you can do so within the server callback handler, by throwing an Exception. The following example shows how this can be done:

```
public class AuthorizingServerCallbackHandler implements CallbackHandler {

    @Override
    public void handle(Callback[] callbacks) throws IOException,
UnsupportedCallbackException {
        for (Callback callback : callbacks) {
            ...
            if (callback instanceof AuthorizeCallback) {
                AuthorizeCallback acb = (AuthorizeCallback) callback;
                UserProfile user = UserManager.loadUser(acb.getAuthenticationID());
                if (!user.hasRole("myclusterrole")) {
                    throw new SecurityException("Unauthorized node " +user);
                }
            }
            ...
        }
    }
}
```

Chapter 2. Security

2.1. General concepts

2.1.1. Authorization

Just like embedded mode, the server supports cache authorization using the same configuration, e.g.:

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

2.1.2. Server Realms

{brandname} Server security is built around the features provided by the underlying server realm and security domains. Security Realms are used by the server to provide authentication and authorization information for both the management and application interfaces.

Security Realm configuration

```
<server xmlns="urn:jboss:domain:2.1">
   ...
   <management>
        ...
        <security-realm name="ApplicationRealm">
           <authentication>
              <properties path="application-users.properties" relative-to=
"jboss.server.config.dir"/>
           </authentication>
           <authorization>
              <properties path="application-roles.properties" relative-to=
"jboss.server.config.dir"/>
           </authorization>
        </security-realm>
        ...
    </management>
    ...
</server>
```

{brandname} Server comes with an add-user.sh script (add-user.bat for Windows) to ease the process of adding new user/role mappings to the above property files. An example invocation for adding a user to the ApplicationRealm with an initial set of roles:

./bin/add-user.sh -a -u myuser -p "qwer1234!" -ro supervisor,reader,writer

It is also possible to authenticate/authorize against alternative sources, such as LDAP, JAAS, etc. Refer to the WildFly security realms guide on how to configure the Security Realms. Bear in mind that the choice of authentication mechanism you select for the protocols limits the type of authentication sources, since the credentials must be in a format supported by the algorithm itself (e.g. pre-digested passwords for the digest algorithm)

2.2. Security Audit

The {brandname} subsystem security audit by default sends audit logs to the audit manager configured at the server level. Refer to the WildFly security subsystem guide on how to configure the server audit manager. Alternatively you can also set your custom audit logger by using the same configuration as for embedded mode. Refer to the Security chapter in the user guide for details.

2.3. Hot Rod authentication

The Hot Rod protocol supports authentication by leveraging the SASL mechanisms. The supported SASL mechanisms (usually shortened as mechs) are:

- PLAIN - This is the most insecure mech, since credentials are sent over the wire in plain-text format, however it is the simplest to get to work. In combination with encryption (i.e. TLS) it can be used safely

- DIGEST-MD5 - This mech hashes the credentials before sending them over the wire, so it is more secure than PLAIN
- GSSAPI - This mech uses Kerberos tickets, and therefore requires the presence of a properly configured Kerberos Domain Controller (such as Microsoft Active Directory)
- EXTERNAL - This mech obtains credentials from the underlying transport (i.e. from a X.509 client certificate) and therefore requires encryption using client-certificates to be enabled.

The following configuration enables authentication against ApplicationRealm, using the DIGEST-MD5 SASL mechanism and only enables the **auth** QoP (see SASL Quality of Protection):

Hot Rod connector configuration

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

Notice the server-name attribute: it is the name that the server declares to incoming clients and therefore the client configuration must match. It is particularly important when using GSSAPI as it is equivalent to the Kerberos service name. You can specify multiple mechanisms and they will be attempted in order.

2.3.1. SASL Quality of Protection

While the main purpose of SASL is to provide authentication, some mechanisms also support integrity and privacy protection, also known as Quality of Protection (or qop). During authentication negotiation, ciphers are exchanged between client and server, and they can be used to add checksums and encryption to all subsequent traffic. You can tune the required level of qop as follows:

QOP	Description
auth	Authentication only
auth-int	Authentication with integrity protection
auth-conf	Authentication with integrity and privacy protection

2.3.2. SASL Policies

You can further refine the way a mechanism is chosen by tuning the SASL policies. This will effectively include / exclude mechanisms based on whether they match the desired policies.

Policy	Description
forward-secrecy	Specifies that the selected SASL mechanism must support forward secrecy between sessions. This means that breaking into one session will not automatically provide information for breaking into future sessions.

Policy	Description
pass-credentials	Specifies that the selected SASL mechanism must require client credentials.
no-plain-text	Specifies that the selected SASL mechanism must not be susceptible to simple plain passive attacks.
no-active	Specifies that the selected SASL mechanism must not be susceptible to active (non-dictionary) attacks. The mechanism might require mutual authentication as a way to prevent active attacks.
no-dictionary	Specifies that the selected SASL mechanism must not be susceptible to passive dictionary attacks.
no-anonymous	Specifies that the selected SASL mechanism must not accept anonymous logins.

Each policy's value is either "true" or "false". If a policy is absent, then the chosen mechanism need not have that characteristic (equivalent to setting the policy to "false"). One notable exception is the **no-anonymous** policy which, if absent, defaults to true, thus preventing anonymous connections.

It is possible to have mixed anonymous and authenticated connections to the endpoint, delegating actual access logic to cache authorization configuration. To do so, set the **no-anonymous** policy to false and turn on cache authorization.

The following configuration selects all available mechanisms, but effectively only enables GSSAPI, since it is the only one that respects all chosen policies:

Hot Rod connector policies

```
<hotrod-connector socket-binding="hotrod" cache-container="default">
   <authentication security-realm="ApplicationRealm">
      <sasl server-name="myhotrodserver" mechanisms="PLAIN DIGEST-MD5 GSSAPI EXTERNAL"
qop="auth">
         <policy>
            <no-active value="true" />
            <no-anonymous value="true" />
            <no-plain-text value="true" />
         </policy<>
      </sasl>
   </authentication>
</hotrod-connector>
```

2.3.3. Using GSSAPI/Kerberos

If you want to use GSSAPI/Kerberos, setup and configuration differs. First we need to define a Kerberos login module using the security domain subsystem:

Security domain configuration

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

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

Next we need to modify the Hot Rod connector

Hot Rod connector configuration

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

2.4. Hot Rod and REST encryption (TLS/SSL)

Both Hot Rod and REST protocols support encryption using SSL/TLS with optional TLS/SNI support (Server Name Indication). To set this up you need to create a keystore using the keytool application which is part of the JDK to store your server certificate. Then add a <server-identities> element to your security realm:

Security Realm configuration for SSL

```
<security-realm name="ApplicationRealm">
    <server-identities>
        <ssl>
            <keystore path="keystore_server.jks" relative-to="jboss.server.config.dir"
keystore-password="secret" />
        </ssl>
    </server-identities>
</security-realm>
```

When using SNI support there might be multiple Security Realms configured.

It is also possible to generate development certificates on server startup. In order to do this, just specify `generate-self-signed-certificate-host` in the keystore element as shown below:

Generating Keystore automatically

```
<security-realm name="ApplicationRealm">
    <server-identities>
        <ssl>
            <keystore path="keystore_server.jks" relative-to="jboss.server.config.dir"
keystore-password="secret" generate-self-signed-certificate-host="localhost"/>
        </ssl>
    </server-identities>
</security-realm>
```

There are three basic principles that you should remember when using automatically generated keystores:

- They shouldn't be used on a production environment
- They are generated when necessary (e.g. while obtaining the first connection from the client)
- They contain also certificates so they might be used in a Hot Rod client directly

Next modify the <hotrod-connector> and/or <rest-connector> elements in the endpoint subsystem to require encryption. Optionally add SNI configuration:

Hot Rod connector SSL configuration

```
<hotrod-connector socket-binding="hotrod" cache-container="local">
    <encryption security-realm="ApplicationRealm" require-ssl-client-auth="false">
        <sni host-name="domain1" security-realm="Domain1ApplicationRealm" />
        <sni host-name="domain2" security-realm="Domain2ApplicationRealm" />
    </encryption>
</hotrod-connector>
<rest-connector socket-binding="rest" cache-container="local">
    <encryption security-realm="ApplicationRealm" require-ssl-client-auth="false">
        <sni host-name="domain1" security-realm="Domain1ApplicationRealm" />
        <sni host-name="domain2" security-realm="Domain2ApplicationRealm" />
    </encryption>
</rest-connector>
```

To configure the client In order to connect to the server using the Hot Rod protocol, the client needs a trust store containing the public key of the server(s) you are going to connect to, unless the key was signed by a Certification Authority (CA) trusted by the JRE.

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
    .addServer()
        .host("127.0.0.1")
        .port(11222)
     .security()
        .ssl()
           .enabled(true)
           .sniHostName("domain1")
           .trustStoreFileName("truststore_client.jks")
           .trustStorePassword("secret".toCharArray());
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
```

Additionally, you might also want to enable client certificate authentication (and optionally also allow the use of the EXTERNAL SASL mech to authenticate and authorize clients). To enable this you will need the security realm on the server to be able to trust incoming client certificates by adding a trust store:

```
<security-realm name="ApplicationRealm">
   <authentication>
      <truststore path="truststore_server.jks" relative-to="jboss.server.config.dir"
keystore-password="secret"/>
   </authentication>
   <server-identities>
       <ssl>
           <keystore path="keystore_server.jks" relative-to="jboss.server.config.dir"
keystore-password="secret" />
       </ssl>
   </server-identities>
</security-realm>
```

And then tell the connector to require a client certificate:

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

The client, at this point, will also need to specify a keyStore which contains its certificate on top of the trustStore which trusts the server certificate. See the Hot Rod client encryption

section to learn how.