Security Guide for Infinispan 12.0

Table of Contents

Chapter 1. Infinispan Security

Infinispan provides security for components as well as data across different layers:

- Within the core library to provide role-based access control (RBAC) to CacheManagers, Cache instances, and stored data.
- Over remote protocols to authenticate client requests and encrypt network traffic.
- Across nodes in clusters to authenticate new cluster members and encrypt the cluster transport.

The Infinispan core library uses standard Java security libraries such as JAAS, JSSE, JCA, JCE, and SASL to ease integration and improve compatibility with custom applications and container environments. For this reason, the Infinispan core library provides only interfaces and a set of basic implementations.

Infinispan servers support a wide range of security standards and mechanisms to readily integrate with enterprise-level security frameworks.

Chapter 2. Configuring Infinispan Authorization

Authorization restricts the ability to perform operations with Infinispan and access data. You assign users with roles that have different permission levels.

2.1. Infinispan Authorization

Infinispan lets you configure authorization to secure Cache Managers and cache instances. When user applications or clients attempt to perform an operation on secured Cached Managers and caches, they must provide an identity with a role that has sufficient permissions to perform that operation.

For example, you configure authorization on a specific cache instance so that invoking `Cache.get()` requires an identity to be assigned a role with read permission while `Cache.put()` requires a role with write permission.

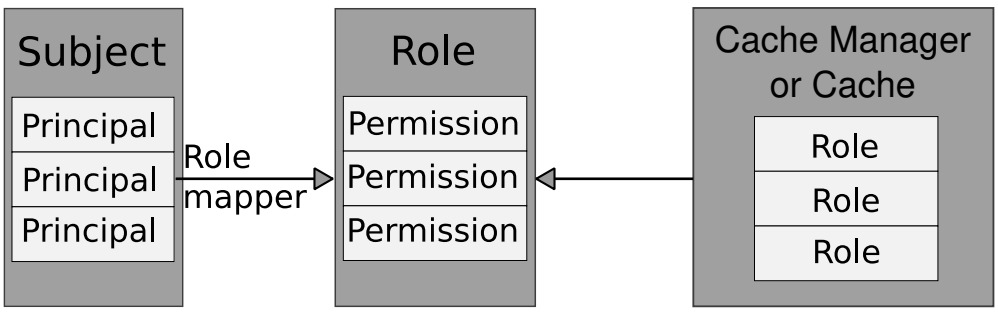
In this scenario, if a user application or client with the `reader` role attempts to write an entry, Infinispan denies the request and throws a security exception. If a user application or client with the `writer` role sends a write request, Infinispan validates authorization and issues a token for subsequent operations.

Identity to Role Mapping

Identities are security Principals of type `java.security.Principal`. Subjects, implemented with the `javax.security.auth.Subject` class, represent a group of security Principals. In other words, a Subject represents a user and all groups to which it belongs.

Infinispan uses role mappers so that security principals correspond to roles, which represent one or more permissions.

The following image illustrates how security principals map to roles:



2.1.1. Permissions

Permissions control access to Cache Managers and caches by restricting the actions that you can perform. Permissions can also apply to specific entities such as named caches.

Table 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.
ALL	-	Includes all Cache Manager permissions.

Table 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.
ALL	-	Includes all cache permissions.
ALL_READ	-	Combines the READ and BULK_READ permissions.
ALL_WRITE	-	Combines the WRITE and BULK_WRITE permissions.

Combining permissions

You might need to combine permissions so that they are useful. For example, to allow "supervisors" to run stream operations but restrict "standard" users to puts and gets only, you can define the following mappings:

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

Reference

- Infinispan Security API

2.1.2. Role Mappers

Infinispan includes a `PrincipalRoleMapper` API that maps security Principals in a Subject to authorization roles. There are two role mappers available by default:

IdentityRoleMapper

Uses the Principal name as the role name.

- Java class: `org.infinispan.security.mappers.IdentityRoleMapper`
- Declarative configuration: `<identity-role-mapper />`

CommonNameRoleMapper

Uses the Common Name (CN) as the role name if the Principal name is a Distinguished Name (DN). For example the `cn=managers,ou=people,dc=example,dc=com` DN maps to the `managers` role.

- Java class: `org.infinispan.security.mappers.CommonRoleMapper`
- Declarative configuration: `<common-name-role-mapper />`

You can also use custom role mappers that implement the `org.infinispan.security.PrincipalRoleMapper` interface. To configure custom role mappers declaratively, use: `<custom-role-mapper class="my.custom.RoleMapper" />`

Reference

- Infinispan Security API
- org.infinispan.security.PrincipalRoleMapper

2.2. Programmatically Configuring Authorization

When using Infinispan as an embedded library, you can configure authorization with the `GlobalSecurityConfigurationBuilder` and `ConfigurationBuilder` classes.

Procedure

1. Construct a `GlobalConfigurationBuilder` that enables authorization, specifies a role mapper, and defines a set of roles and permissions.

```
GlobalConfigurationBuilder global = new GlobalConfigurationBuilder();
global
   .security()
      .authorization().enable() ①
         .principalRoleMapper(new IdentityRoleMapper()) ②
         .role("admin") ③
            .permission(AuthorizationPermission.ALL)
         .role("reader")
            .permission(AuthorizationPermission.READ)
         .role("writer")
            .permission(AuthorizationPermission.WRITE)
         .role("supervisor")
            .permission(AuthorizationPermission.READ)
            .permission(AuthorizationPermission.WRITE)
            .permission(AuthorizationPermission.EXEC);
```

① Enables Infinispan authorization for the Cache Manager.

② Specifies an implementation of `PrincipalRoleMapper` that maps Principals to roles.

③ Defines roles and their associated permissions.

2. Enable authorization in the `ConfigurationBuilder` for caches to restrict access based on user roles.

```
ConfigurationBuilder config = new ConfigurationBuilder();
config
   .security()
      .authorization()
         .enable(); ①
```

① Implicitly adds all roles from the global configuration.

If you do not want to apply all roles to a cache, explicitly define the roles that are authorized for caches as follows:

```
ConfigurationBuilder config = new ConfigurationBuilder();
config
   .security()
      .authorization()
         .enable()
         .role("admin") ①
         .role("supervisor")
         .role("reader");
```

① Defines authorized roles for the cache. In this example, users who have the `writer` role only are not authorized for the "secured" cache. Infinispan denies any access requests from those users.

Reference

- org.infinispan.configuration.global.GlobalSecurityConfigurationBuilder
- org.infinispan.configuration.cache.ConfigurationBuilder

2.3. Declaratively Configuring Authorization

Configure authorization in your `infinispan.xml` file.

Procedure

1. Configure the global authorization settings in the `cache-container` that specify a role mapper, and define a set of roles and permissions.
2. Configure authorization for caches to restrict access based on user roles.

```
<infinispan>
   <cache-container default-cache="secured" name="secured">
      <security>
         <authorization> ①
            <identity-role-mapper /> ②
            <role name="admin" permissions="ALL" /> ③
            <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/> ④
         </security>
      </local-cache>
   </cache-container>
</infinispan>
```

① Enables Infinispan authorization for the Cache Manager.

② Specifies an implementation of `PrincipalRoleMapper` that maps Principals to roles.

③ Defines roles and their associated permissions.

④ Implicitly adds all roles from the global configuration.

If you do not want to apply all roles to a cache, explicitly define the roles that are authorized for caches as follows:

```
<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 supervisor reader"/> ①
         </security>
      </local-cache>
   </cache-container>

</infinispan>
```

① Defines authorized roles for the cache. In this example, users who have the `writer` role only are not authorized for the "secured" cache. Infinispan denies any access requests from those users.

Reference

- Infinispan Configuration Schema Reference

2.4. Code Execution with Secure Caches

When you configure Infinispan authorization and then construct a `DefaultCacheManager`, it returns a `SecureCache` that checks the security context before invoking any operations on the underlying caches. A `SecureCache` also ensures that applications cannot retrieve lower-level insecure objects such as `DataContainer`. For this reason, you must execute code with an identity that has the required authorization.

In Java, executing code with a specific identity usually means wrapping the code to be executed within a `PrivilegedAction` as follows:

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

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

With Java 8, you can simplify the preceding call as follows:

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

The preceding call uses the `Security.doAs()` method instead of `Subject.doAs()`. You can use either method with Infinispan, however `Security.doAs()` provides better performance.

If you need the current Subject, use the following call to retrieve it from the Infinispan context or from the AccessControlContext:

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

Chapter 3. Securing JGroups

Configure JGroups to secure Infinispan clusters.

3.1. Configuring JGroups Authentication

Configure JGroups authentication to restrict Infinispan cluster membership. When joining or merging, nodes must authenticate with the cluster.

Procedure

- Add the SASL mechanism to your JGroups configuration before the `GMS` protocol, as in the following example:

```
<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" />
```

The preceding example uses `DIGEST-MD5` so that each node must declare valid credentials when joining a Infinispan cluster.

Within the cluster, coordinator nodes act as SASL servers. All other nodes act as SASL clients. For this reason you need two different `CallbackHandlers`, a server_callback_handler_class for the coordinator and a client_callback_handler_class for the other nodes.

Cluster authorization

To implement node authorization, configure the server callback handler to throw an exception as in the following example:

```
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);
                }
            }
            ...
        }
    }
}
```