Oracle® Fusion Middleware

Securing Oracle Coherence



26 (26.1.0.0.0)
G56904-01
November 2026



Oracle Fusion Middleware Securing Oracle Coherence, 26 (26.1.0.0.0)

G56904-01

Copyright © 2008, 2026, Oracle and/or its affiliates.

Primary Author: Oracle Corporation

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Oracle customer access to and use of Oracle support services will be pursuant to the terms and conditions specified in their Oracle order for the applicable services.

Contents

Preface

Audience i
Documentation Accessibility i
Diversity and Inclusion ii
Related Documents ii
Conventions ii

1 Introduction to Oracle Coherence Security

Conceptual Overview of Oracle Coherence Security 1
Understanding Security Profiles 2
Coherence Security Quick Start 3
Overview of Security Configuration 4

2 Enabling General Security Measures

Using the Java Security Manager 1
- Enable the Java Security Manager 2
- Specify Permissions 2
- Programmatically Specifying Local Permissions 3

Using Host-Based Authorization 3
- Overview of Host-Based Authorization 4
- Specify Cluster Member Authorized Hosts 4
- Specify Extend Client Authorized Hosts 5
- Use a Filter Class to Determine Authorization 5

Managing Rogue Clients 6

3 Using TLS to Secure Communication

Configuring TLS with Protected Password Sources 3
Verifying and Rotating TLS Identities 5
Overview of SSL/TLS 6
Coherence Socket Providers 7
- Configuring the Identity Manager 8

Copyright © 2008, 2026, Oracle and/or its affiliates.

Configuring a Trust Manager 10
Resolving the Socket Provider URL 11
Using a Socket Provider in Configuration 11
Configure a Socket Provider at Runtime 13
Using SSL to Secure Cluster Communication 14
Cluster Communication Using mTLS 15
Cluster Communication with One-Way SSL 16
Using SSL to Secure Extend and gRPC Client Communication 17
Configuring a Cluster-Side Extend Proxy SSL Socket Provider 17
Configuring the Cluster-Side gRPC Proxy SSL Socket Provider 20
Configuring a Java Extend or gRPC Client SSL Socket Provider 21
Configure a Default Socket Provider for a Cache Configuration File 25
Configuring a .NET Client-Side Stream Provider 27
Securing the C++ Client with SSL/TLS 28
Using SSL to Secure Federation Communication 29
Federation with mTLS 30
Federation with One-Way SSL 31
Coherence PeerX509 Algorithm 32
Specifying a Global Socket Provider 32
Specifying Passwords in Socket Provider Configuration 34
Specify Plain Text Passwords 35
Passwords From Java System Properties 35
Reading Passwords From a URL 35
Custom Password Providers 36
Controlling Cipher Suite and Protocol Version Usage 37
Using Host Name Verification 37
Using the Default Coherence Host Name Verifier 38
Using a Custom Host Name Verifier 39
Configuring Client Authentication 39
Using Private Key and Certificate Files 41
Configuring an Identity Manager 41
Configuring a Trust Manager 42
Using Custom Keystore, Private Key, and Certificate Loaders 42
Using the Custom KeyStore Loader 42
Using the Custom PrivateKey Loader 45
Using a Custom Certificate Loader 46
Using Refreshable KeyStores, Private Keys, and Certificates 48
Configuring a Refresh Policy 49

4 Using an Access Controller

Authenticating Credentials with JAAS 2

Copyright © 2008, 2026, Oracle and/or its affiliates.

Using Authenticated Subjects with Modern Java 4
Overview of Using an Access Controller 5
Proof of Identity 7
Configuring the JAAS LoginModule 7
Configuring a JAAS Callback Handler 8
Using a Custom JAAS LoginModule 9
Enable the Access Controller 10
Using the Default Access Controller Implementation 10
The Key Store Type and Encryption Algorithm 11
The Default Access Controller Keystore 11
Create a Permissions File 12
Enable Security Audit Logs 12
Configure the Default Access Controller 12
Using a Custom Access Controller Implementation 14

5 Authorizing Access to Server-Side Operations

Implementing an Account Storage Authorizer 1
Overview of Access Control Authorization 3
Creating Access Control Authorization Implementations 4
Declaring Access Control Authorization Implementations 5
Enabling Access Control Authorization on a Partitioned Cache 6

6 Securing Extend Client Connections

Using Identity Tokens to Restrict Client Connections 1
Overview of Using Identity Tokens 2
Creating a Custom Identity Transformer 2
Enabling a Custom Identity Transformer 3
Creating a Custom Identity Asserter 3
Enabling a Custom Identity Asserter 4
Using Custom Security Types 4
Understanding Custom Identity Token Interoperability 5
Associating Identities with Extend Services 5
Implementing Extend Client Authorization 5
Overview of Extend Client Authorization 6
Create Authorization Interceptor Classes 7
Enable Authorization Interceptor Classes 9

7 Securing Serialization and Remote Execution

Developing Classified Application Types 3

Copyright © 2008, 2026, Oracle and/or its affiliates.

Generating Application Security Configuration 5
Classifying Remote Executable Objects 8
Choosing a Lambda Serialization Policy 14

8 Limiting Resource Use and Protecting Persisted Data

Configuring Serialization Container Limits 1
Transaction Bridges and Persistence Metadata 3
Protecting Memcached Values and Native Indexes 5

9 Securing gRPC Client Connections

Configuring a Secure gRPC Proxy 1
Connecting a Java gRPC Client Securely 2
gRPC Serializer and Diagnostic Policy 4

10 Securing Oracle Coherence REST

Constraining REST Resources and Expressions 1
REST JSON and XML Input Policy 3
Overview of Securing Oracle Coherence REST 4
Using HTTP Basic Authentication with Oracle Coherence REST 4
Specify Basic Authentication for an HTTP Acceptor 4
Specify a Login Module 4
Using SSL Authentication With Oracle Coherence REST 5
Specify Basic Authentication for an HTTP Acceptor 5
Configure an HTTP Acceptor SSL Socket Provider 6
Access Secured REST Services 7
Using SSL and HTTP Basic Authentication with Oracle Coherence REST 10
Implementing Authorization For Oracle Coherence REST 10

11 Securing Management and Health Services

Management Authentication and Compatibility 1
Securing JMX Connectors and Reporter 4
Securing Health Administration 5
About Securing Oracle Coherence HTTP Management Server 5
Basic Authentication for Coherence HTTP Management Server HTTP Acceptor 6
Specify the Basic Authentication for Coherence HTTP Management Server HTTP Acceptor 6
Specify a Coherence HTTP Management Server Login Module 6
Using SSL Authentication With Oracle Coherence HTTP Management Server 7

Copyright © 2008, 2026, Oracle and/or its affiliates.

Configure a Coherence HTTP Management Acceptor SSL Socket Provider 7

12 Securing Oracle Coherence Metrics

Configuring an Authenticated TLS Metrics Scrape 1
About Securing Oracle Coherence Metrics 2
Basic Authentication for Coherence Metrics Http Acceptor 2
Specify Basic Authentication for Coherence Metrics HTTP Acceptor 2
Specify a Coherence Metrics Login Module 2
Specify Basic Authentication for a Coherence Metrics HTTP Client 3
Using SSL Authentication With Oracle Coherence Metrics 3
Configure a Coherence Metrics HTTP Acceptor SSL Socket Provider 3

13 Securing Cluster Peers and Federation

Configuring Peer Proof Prerequisites 1

14 Securing Coherence RAG

RAG Configuration and Model Download Policy 1
Restricting RAG Document Imports 2

15 Securing Oracle Coherence in Oracle WebLogic Server

Overview of Securing Oracle Coherence in Oracle WebLogic Server 1
Securing Coherence using SSL/TLS 2
Extended Usage Certificates 3
Configure Coherence Cluster Traffic Using mTLS 3
Configure Coherence Cluster Traffic Using One-Way SSL/TLS 3
Using a Custom Coherence Operational Configuration File 4
Configure the Coherence Global Socket Provider 5
WebLogic Server Secured Production Mode 7
Configure Coherence for One-Way SSL/TLS in Secured Production Mode 7
Disable Coherence SSL/TLS in Secured Production Mode 8
Securing Oracle Coherence Cluster Membership 9
Enabling the Oracle Coherence Security Framework 9
Specifying an Identity for Use by the Security Framework 9
Authorizing Oracle Coherence Caches and Services 10
Specifying Cache Authorization 11
Specifying Service Authorization 11
Securing Extend Client Access with Identity Tokens 12
Enabling Identity Transformers for Use in Oracle WebLogic Server 13

Copyright © 2008, 2026, Oracle and/or its affiliates.

Enabling Identity Asserters for Use in Oracle WebLogic Server 13

16 Upgrading and Operating a Secured Deployment

Rolling Upgrade to Hardened Security 1
Assembling Secure Deployment Recipes 3

A Security Configuration Reference

B Security Behavior Matrix

C Security Errors and Diagnostic Signals

Copyright © 2008, 2026, Oracle and/or its affiliates.

Preface

Securing Oracle Coherence explains key security concepts and provides instructions for implementing various levels of security for Oracle Coherence clusters, Oracle Coherence REST, and Oracle Coherence*Extend clients.

This preface includes the following sections:

- Audience
- Documentation Accessibility
- Diversity and Inclusion
- Related Documents
- Conventions

Audience

This guide is intended for the following audiences:

- Primary Audience – Application developers and operators who want to secure an Oracle Coherence cluster and secure Oracle Coherence*Extend client communication with the cluster
- Secondary Audience – System architects who want to understand the options and architecture for securing an Oracle Coherence cluster and Oracle Coherence*Extend clients

The audience must be familiar with Oracle Coherence, Oracle Coherence REST, and Oracle Coherence*Extend to use this guide effectively. In addition, users must be familiar with Java and Secure Socket Layer (SSL). The examples in this guide require the installation and use of the Oracle Coherence product, including Oracle Coherence*Extend. The use of an integrated development environment (IDE) is not required, but it is recommended to facilitate working through the examples.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customer access to and use of Oracle support services will be pursuant to the terms and conditions specified in their Oracle order for the applicable services.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Related Documents

For more information, see the following documents in the Oracle Coherence documentation set:

- *Administering HTTP Session Management with Oracle Coherence*Web*
- *Administering Oracle Coherence*
- *Developing Applications with Oracle Coherence*
- *Developing Remote Clients for Oracle Coherence*
- *Installing Oracle Coherence*
- *Integrating Oracle Coherence*
- *Managing Oracle Coherence*
- *Java API Reference for Oracle Coherence*
- *C++ API Reference for Oracle Coherence*
- *.NET API Reference for Oracle Coherence*
- *Release Notes for Oracle Coherence*

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Copyright © 2008, 2026, Oracle and/or its affiliates.

1

Introduction to Oracle Coherence Security

Choose explicit controls for cluster members, external clients, management, application code, and persisted data.
This guide explains how to configure and operate Coherence security across its distributed data and administrative surfaces. Start with Understanding Security Profiles and Coherence Security Quick Start, then follow the recipes for the listeners and application behavior your deployment uses.

Treat a cluster member as trusted application infrastructure. Restrict membership and member-side code deployment. Authenticate external clients, authorize their operations, classify remotely supplied executable objects, and constrain the data and resource boundaries described in this guide. Protect management and optional components independently.

Deployment and version scope

Starting with Coherence 26.1.0.0, an unset `coherence.security.mode` selects the hardened security profile. Select `compatibility` explicitly if an upgrade plan needs a temporary compatibility phase. Match configuration to the deployed distribution, JDK, client version, and optional modules, and consult the release/patch documentation for another code line. Newer Java APIs used by an example have explicit prerequisites; older-JDK compatibility material is labeled accordingly.

The security profile does not provide every service's transport and identity setup. Use Security Behavior Matrix to distinguish profile-sensitive defaults, mode-independent protections, and separately configured policy. Use Security Configuration Reference for ownership and override details.

Start from the deployment task

- For new deployment, assemble the baseline, TLS, identity, authorization, and application-artifact configuration before exposing listeners.
- For an existing deployment, follow Rolling Upgrade to Hardened Security and exercise compatibility and hardened workflows before changing production policy.
- For a failed request, identify the boundary and signal in Security Errors and Diagnostic Signals before loosening a setting.
- Conceptual Overview of Oracle Coherence Security
- Understanding Security Profiles
- Coherence Security Quick Start
- Overview of Security Configuration

Conceptual Overview of Oracle Coherence Security

Understand the separate trust decisions made for connections, subjects, application types, operations, and stored bytes.
Security configuration should answer five questions: which peer may connect, which identity made a request, which operation that identity may perform, which application types/behavior a

Copyright © 2008, 2026, Oracle and/or its affiliates.

receiver may accept, and which stored input a process may consume. Apply the corresponding controls together.

Boundary	Principal controls	Practical limit
Cluster member and federation peer	Network isolation, TLS/mTLS, approved participant/endpoints, supported peer proof	A cluster name or self-advertised address is not cryptographic identity.
External client/proxy	TLS, configured identity assertion, client/protocol policy	A raw subject or accepted object type does not prove a user login.
Cache operation	Service permissions, storage authorization, application policy	An allowed executable class does not grant a caller a role.
Deserialization/execution	Class and format validation, serializer limits, executable classification, lambda policy	Trusted classes can still execute application code with member process privileges.
Management/observability	Dedicated auth/transport/bind policy, target/operation/resource checks	Cache client security does not automatically cover an administrative listener.
Persistent/imported data	Protected paths, typed metadata, bounds, import validation	Structural checks are not cryptographic authenticity of all stored bytes.

Inventory enabled surfaces

Inventory cluster/federation transport and discovery, NameService, Extend cache and InvocationService proxies, Concurrent proxy/task endpoints, gRPC, REST, management HTTP, metrics, health, JMX/RMI/JMXMP, Memcached, and optional RAG APIs. For each enabled surface record the bind address, reachable networks, TLS/trust source, identity policy, authorization policy, serializer, and verification procedure.

Disable unused listeners. Keep storage and proxy/member roles deliberate. Host allowlists constrain location, and client suspect/bounds policies limit particular malformed or abusive traffic; neither replaces authentication or workload admission. See Enabling General Security Measures and the relevant protocol chapter.

Understanding Security Profiles

Use the security profile to select compatibility-sensitive hardening behavior independently of the Coherence runtime mode.
Coherence has separate settings for runtime mode and security profile. Set these properties before starting Coherence. To change the selected profile, update the startup configuration and restart the affected processes as part of a coordinated deployment.

Table 1-1 Runtime Mode and Security Profile

Property	Purpose	Values and Default
`coherence.mode`	Selects the runtime and license mode.	`eval`, `dev`, or `prod`. The aliases `development` and `production` are also accepted. The default is `dev`.
`coherence.security.mode`	Selects the security hardening profile.	`compatibility` or `hardened`. When the property is unset, the profile is `compatibility`.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Compatibility Profile

The compatibility profile preserves prior behavior for compatibility-sensitive checks. Some checks emit bounded warnings or diagnostic counters when an operation would be rejected under the hardened profile. Exercise the application's normal workflows and use those signals to identify configuration or application changes before enabling hardened behavior.

Protections that apply in both profiles, including built-in serialization deny rules and malformed-input checks, remain active. Explicitly configured policies also retain their own enforcement rules.

Hardened Profile

The hardened profile enforces compatibility-sensitive hardening checks. These include serialization allowlisting, remote-executable classification, and finite default serializer container limits. Applications may need generated security configuration, executable declarations, or other configuration changes before their workflows pass these checks.

For example, add the following JVM options before the application main class or `-jar` argument to select production runtime mode and the hardened security profile:

```
-Dcoherence.mode=prod
-Dcoherence.security.mode=hardened
```

To retain the compatibility profile while selecting production runtime mode, use:

```
-Dcoherence.mode=prod
-Dcoherence.security.mode=compatibility
```

Setting `coherence.mode=prod` alone leaves the security profile at its default. Invalid values for either property cause configuration resolution to fail.

Configure the Required Security Controls

Configure TLS, authentication, and authorization for each exposed service according to the deployment's requirements. The hardened profile does not configure TLS or install a storage authorizer, and some service policies require their own explicit settings. Review each service's defaults and overrides.

For an existing deployment, complete the patch or version upgrade using the existing configuration and compatibility profile. Prepare and test application artifacts, credentials, and service policies before a coordinated change to hardened behavior. Include every member and client that participates in the affected workflow.

See Coherence Security Quick Start for the security configuration starting points.

Coherence Security Quick Start

Start securing a Coherence deployment by protecting local resources, configuring transport and identity, authorizing operations, and preparing application security configuration. Coherence security combines input protections, security-profile defaults, and controls that you configure for individual services. Choose the controls needed for every exposed service and test both permitted access and the requests that should be rejected.

Copyright © 2008, 2026, Oracle and/or its affiliates.

- Protect configuration files, application artifacts, persistence directories, keystores, and credential sources with operating-system permissions. Restrict listener access to the required networks and disable services that the deployment does not use.
- Configure TLS for cluster and external-client communication. Configure trust and identity material, hostname verification, and client certificates where mutual TLS is required. See Using SSL to Secure Cluster Communication, Using SSL to Secure Extend and gRPC Client Communication, and Using SSL Authentication With Oracle Coherence REST.
- Configure authentication and identity validation for the services that clients use. For Extend, configure identity transformers and asserters appropriate to the application's authentication system. See Securing Extend Client Access with Identity Tokens.
- Configure authorization for the operations and resources each identity may access. Use a storage access authorizer where storage-side operation checks are required, and configure any necessary client-service authorization. See Authorizing Access to Server-Side Operations and Implementing Extend Client Authorization.
- Prepare application serialization and remote-execution policy. Include the required generated `META-INF/coherence/security-config.xml` in each artifact that contributes serialized application types or remote executables. Classify the application objects that execute remotely. Test the workflows that use processors, aggregators, filters, extractors, and remote tasks.
- Prepare the required application and service configuration before starting Coherence. Starting with 26.1.0.0, `coherence.security.mode` defaults to `hardened`; `coherence.mode=prod` remains the independent runtime/license selection. If an existing deployment needs a temporary compatibility phase, explicitly select `coherence.security.mode=compatibility` before upgrading. See Understanding Security Profiles and Rolling Upgrade to Hardened Security.

Apply the same review to management, metrics, health administration, and optional application endpoints. Configure their authentication, transport, and network exposure explicitly, and verify access from the intended clients and monitoring tools.

Use the deployment recipes in Assembling Secure Deployment Recipes to combine these controls. Before changing an existing deployment, follow Rolling Upgrade to Hardened Security. For the gRPC client path, see Securing gRPC Client Connections.

Overview of Security Configuration

Place security configuration in the owning descriptor or artifact and verify its effective value on each process.
Coherence security settings belong to several configuration sources. A setting in the wrong descriptor, a resource absent from the receiving classpath, or an overridden default can leave a deployment behaving differently from the intended policy.

Source	Responsibility
Operational override	Member/network/socket providers, password providers, security framework, identity transformer/asserter, storage-authorizer declarations, peer-provider configuration.
Cache configuration	Cache mappings and schemes, backing-map authorizer references, proxy acceptors, remote initiators/channels, serializers and service-local limits.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Source	Responsibility
`META-INF/coherence/security-config.xml`	Per-artifact allowed application classes, executable classes, and lambda-target classification. All visible resources are validated and merged.
POF configuration	Supported type IDs and serializers for application/product values; explicit compatibility choices.
REST configuration	Published resources, named/direct queries, expression aliases, marshallers, and approved aggregator/processor registrations.
JAAS configuration	Named login-module entries, provider options, and the credential validation contract.
Credentials and trust sources	Private keys, certificates, trust anchors, password-provider material, and component-specific API credentials.

Select `coherence.override` and `coherence.cacheconfig` deliberately for each process. Review system-property-backed XML defaults and explicit overrides. Do not assume a property's documented shipped default applies after an application supplies its own configuration without the shipped marker.

Generate application security resources during the build and verify the JAR actually deployed. Keep configuration, code, native binaries, and credentials consistent across the relevant clients and members. Use Security Configuration Reference and the owning feature's precedence rules rather than applying one global override rule to every feature.

Copyright © 2008, 2026, Oracle and/or its affiliates.

2

Enabling General Security Measures

Protect processes, local artifacts, listeners, and credentials before applying feature-specific policy.
Run members and administrative tools with dedicated, minimally privileged service identities. Restrict write access to application JARs, configuration, native libraries, persistence, snapshots, reports, and credential sources. A process that can replace trusted code or configuration can change the effective security policy.

Bind listeners only where needed and restrict network access to the intended members, clients, administrators, and scrapers. Configure and test each surface in Conceptual Overview of Oracle Coherence Security. Use supported operating-system/container resource and isolation controls for the member process and external data loaders.

Deployment baseline

- Keep a recorded configuration and artifact inventory, including optional modules and native binaries.
- Use TLS trust, matching peer names, and the required identity policy before transmitting credentials.
- Configure service/storage authorization and application serialization policy explicitly.
- Give persistence and report output dedicated protected directories; review restore/import permissions.
- Use bounded logs and operational identifiers. Avoid logging passwords, bearer tokens, complete authorization headers, private keys, or arbitrary payload values.
- Exercise successful and rejected operations before exposing a new listener or activating a stronger profile.

Current JDK security

Java Security Manager instructions apply only to older JDKs that still support it. Beginning with JDK 24 it is permanently disabled. Use Using Authenticated Subjects with Modern Java for current subject context, and configure Coherence's own authorization APIs independently. Keep legacy policy files only for the supported deployment that actually uses them.

- Using the Java Security Manager
- Using Host-Based Authorization
- Managing Rogue Clients

Using the Java Security Manager

Recognize legacy Java policy instructions and use current application and process controls on modern JDKs.
The Java Security Manager is permanently disabled beginning with JDK 24. Enabling it at startup is an error on those JDKs. Java policy files and related checks must not be presented as the security prerequisite for a current Coherence deployment.

Copyright © 2008, 2026, Oracle and/or its affiliates.

The child topics are retained for supported older-JDK deployments that explicitly use the legacy mechanism. Check the exact JDK and Coherence release before using their flags or permissions. Do not add Security Manager flags to a modern JVM to repair a Coherence authorization failure.

JAAS login, authenticated subject context, Coherence access controllers, and `StorageAccessAuthorizer` have separate responsibilities. Use the supported operating-system/container boundary for process resource access and Using Authenticated Subjects with Modern Java for current Java examples.

The Security Manager Is Permanently Disabled

- Enable the Java Security Manager
- Specify Permissions
- Programmatically Specifying Local Permissions

Enable the Java Security Manager

These Java Security Manager instructions apply only to a supported older JDK that still implements that mechanism. The Security Manager is permanently disabled beginning with JDK 24. Use the current Coherence/application authorization and process isolation controls on newer JDKs.

To enable the Java security manager and use the *COHERENCE_HOME* `/lib/security/security.policy` file, set the following properties on a cluster member:

1. Set the `java.security.manager` property to enable the Java security manager. For example:

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

2. Set the `java.security.policy` property to the location of the policy file. For example:

   ```
   -Djava.security.manager
   -Djava.security.policy=/coherence/lib/security/security.policy
   ```

3. Set the `coherence.home` system property to *COHERENCE_HOME* . For example:

   ```
   -Djava.security.manager
   -Djava.security.policy=/coherence/lib/security/security.policy
   -Dcoherence.home=/coherence
   ```

> **Note**
>
> The security policy file assumes that the default Java Runtime Environment (JRE) security permissions have been granted. Therefore, you must be careful to use a single equal sign (=) and not two equal signs (==) when setting the `java.security.policy` system property.

Specify Permissions

These Java Security Manager instructions apply only to a supported older JDK that still implements that mechanism. The Security Manager is permanently disabled beginning with JDK 24. Use the current Coherence/application authorization and process isolation controls on newer JDKs.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Modify the *COHERENCE_HOME* `/lib/security/security.policy` file to include additional permissions as required. See Permissions in the Java Development Kit (JDK) in *Java SE Security*.

To specify additional permissions in the `security.policy` file:

1. Edit the `security.policy` file and add a permission for a resource. For example, the following permission grants access to the `coherence.jar` library:

```
grant codeBase "file:${coherence.home}/lib/coherence.jar"
    {
            permission java.security.AllPermission;
    };
```

2. When you declare binaries, sign the binaries using the JDK `jarsigner` tool. The following example signs the `coherence.jar` resource declared in the previous step:

```
jarsigner -keystore ./keystore.jks -storepass password coherence.jar admin
```

   Add the signer in the permission declaration. For example, modify the original permission as follows to add the `admin` signer.

```
grant SignedBy "admin" codeBase "file:${coherence.home}/lib/coherence.jar"
    {
            permission java.security.AllPermission;
    };
```

3. Use operating system mechanisms to protect all relevant files from malicious modifications.

Programmatically Specifying Local Permissions

Scope legacy local Java permission checks to JDKs that support them and use current Coherence authorization for application policy.
Local Java permission-policy examples depend on the Java Security Manager contract of the deployed JDK. That contract is unavailable on JDK 24 and later. Retain older permission configuration only for a supported older-JDK environment that explicitly requires it.

For current applications, define the resource and operation policy in the appropriate Coherence/application authorizer and establish a verified subject through the chosen identity system. Use operating-system and container controls for file, process, and network privileges. A successful subject login does not install a JVM sandbox.

See Using an Access Controller, Authorizing Access to Server-Side Operations, and Using Authenticated Subjects with Modern Java.

Using Host-Based Authorization

Host-based authorization is a type of access control that allows you to specify which hosts (based on host name or IP address) can connect to a cluster. The feature is available for both cluster member connections and extend client connections.
This section includes the following topics:

- Overview of Host-Based Authorization
- Specify Cluster Member Authorized Hosts
- Specify Extend Client Authorized Hosts
- Use a Filter Class to Determine Authorization

Copyright © 2008, 2026, Oracle and/or its affiliates.

Overview of Host-Based Authorization

Host-based authorization uses the host name and IP address of a cluster member or extend client to determine whether a connection to the cluster is allowed. Specific host names, addresses, and address ranges can be defined. For custom processing, a custom filter can be created to validate hosts.

Host-based authorization is ideal for environments where known hosts with relatively static network addresses are joining or accessing the cluster. In dynamic environments, or when updating a DNS server, IP addresses can change and cause a cluster member or extend client to fail authorization. Cache operations may not complete if cluster members or extend clients are no longer authorized. Extend clients are more likely to have access problems because of their transient nature.

When using host-based authorization, consider the dynamic nature of the network environment. The need to reconfigure the list of authorized hosts may become impractical. If possible, always use a range of IP addresses instead of using a specific host name. Or, create a custom filter that is capable of resolving address that have changed. If host-based authorization becomes impractical, consider using extend client identity tokens or SSL. See Using Identity Tokens to Restrict Client Connections and Using TLS to Secure Communication, respectively.

Specify Cluster Member Authorized Hosts

The default behavior of a cluster allows any host to connect to the cluster and become a cluster member. Host-based authorization changes this behavior to allow only hosts with specific host names or IP addresses to connect to the cluster.

Configure authorized hosts in an operational override file using the `<authorized-hosts>` element within the `<cluster-config>` element. Enter specific addresses using the `<host-address>` element or a range of addresses using the `<host-range>` element. The `<host-address>` and `<host-range>` elements support an `id` attribute for uniquely identifying multiple elements. Additionally, an application can set the system property `coherence.authorized.hosts` with a comma-separated list of IP addresses and/or host names authorized as cluster members.

The following example configures a cluster to accept only cluster members whose IP address is either 192.168.0.5, 192.168.0.6, or within the range of 192.168.0.10 to 192.168.0.20 and 192.168.0.30 to 192.168.0.40.

```
<?xml version='1.0'?>

<coherence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://xmlns.oracle.com/coherence/coherence-operational-config"
   xsi:schemaLocation="http://xmlns.oracle.com/coherence/coherence-operational-config
   coherence-operational-config.xsd">
   <cluster-config>
      <authorized-hosts>
         <host-address id="1">192.168.0.5</host-address>
         <host-address id="2">192.168.0.6</host-address>
         <host-range id="1">
            <from-address>192.168.0.10</from-address>
            <to-address>192.168.0.20</to-address>
         </host-range>
         <host-range id="2">
            <from-address>192.168.0.30</from-address>
            <to-address>192.168.0.40</to-address>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
         </host-range>
      </authorized-hosts>
   </cluster-config>
</coherence>
```

Specify Extend Client Authorized Hosts

The default behavior of an extend proxy server allows any extend client to connect to the cluster. Host-based authorization changes this behavior to allow only hosts with specific host names or IP addresses to connect to the cluster.

Configure authorized hosts in a cache configuration file using the `<authorized-hosts>` element within the `<tcp-acceptor>` element of a proxy scheme definition. Enter specific addresses using the `<host-address>` element or a range of addresses using the `<host-range>` element. The `<host-address>` and `<host-range>` elements support an `id` attribute for uniquely identifying multiple elements. Additionally, an application can set the system property `coherence.extend.authorized.hosts` with a comma-separated list of IP addresses and/or host names authorized to join proxy as an extend client.

The following example configures an extend proxy to accept only client connections from clients whose IP address is either 192.168.0.5, 192.168.0.6, or within the range of 192.168.0.10 to 192.168.0.20 and 192.168.0.30 to 192.168.0.40.

```
<proxy-scheme>
   <service-name>ExtendTcpProxyService</service-name>
   <thread-count>5</thread-count>
   <acceptor-config>
      <tcp-acceptor>
         ...
         <authorized-hosts>
            <host-address id="1">192.168.0.5</host-address>
            <host-address id="2">192.168.0.6</host-address>
            <host-range id="1">
               <from-address>192.168.0.10</from-address>
               <to-address>192.168.0.20</to-address>
            </host-range>
            <host-range id="2">
               <from-address>192.168.0.30</from-address>
               <to-address>192.168.0.40</to-address>
            </host-range>
         </authorized-hosts>
         ...
      </tcp-acceptor>
   </acceptor-config>
   <autostart>true</autostart>
</proxy-scheme>
```

Use a Filter Class to Determine Authorization

A filter class determines whether to accept a particular host connection. Both extend client connections and cluster member connections support using filter classes. A filter class must implement the `com.tangosol.util.Filter` interface. The `evaluate()` method of the interface is passed the `java.net.InetAddress` of the host. Implementations should return `true` to accept the connection.

To enable a filter class, enter a fully qualified class name using the `<class-name>` element within the `<host-filter>` element. Set initialization parameters using the `<init-params>` element.

Copyright © 2008, 2026, Oracle and/or its affiliates.

The following example configures a filter named `MyFilter`, which determines if a host connection is allowed.

```
<authorized-hosts>
   <host-filter>
      <class-name>package.MyFilter</class-name>
         <init-params>
            <init-param>
               <param-name>sPolicy</param-name>
               <param-value>strict</param-value>
            </init-param>
         </init-params>
   </host-filter>
</authorized-hosts>
```

> **Note**
>
> If `<host-filter>` is provided, all `<host-address>` and `<host-range>` elements are superseded by that filter and have no effect. Setting the system properties `coherence.authorized.hosts` and `coherence.extend.authorized.hosts` is ignored if the corresponding `<authorized-hosts><host-filter>` has been provided, it takes complete precedence.

Managing Rogue Clients

You can use the suspect protocol to safeguard against rogue extend clients that operate outside of acceptable limits. Rogue clients are slow-to-respond clients or abusive clients that attempt to overuse a proxy— as is the case with denial of service attacks. In both cases, the potential exists for a proxy to run out of memory and become unresponsive.
The suspect algorithm monitors client connections looking for abnormally slow or abusive clients. When a rogue client connection is detected, the algorithm closes the connection to protect the proxy server from running out of memory. The protocol works by monitoring both the size (in bytes) and length (in messages) of the outgoing connection buffer backlog for a client. Different levels determine when a client is suspect, when it returns to normal, or when it is considered rogue.

Configure the suspect protocol within the `<tcp-acceptor>` element of a proxy scheme definition. See tcp-acceptor in *Developing Applications with Oracle Coherence*. The suspect protocol is enabled by default.

The following example demonstrates configuring the suspect protocol and is similar to the default settings. When the outgoing connection buffer backlog for a client reaches 10 MB or 10000 messages, the client is considered suspect and is monitored. If the connection buffer backlog for a client returns to 2 MB or 2000 messages, then the client is considered safe and the client is no longer monitored. If the connection buffer backlog for a client reaches 95 MB or 60000 messages, then the client is considered unsafe and the proxy closes the connection.

```
<proxy-scheme>
   <service-name>ExtendTcpProxyService</service-name>
   <thread-count>5</thread-count>
   <acceptor-config>
      <tcp-acceptor>
         ...
         <suspect-protocol-enabled>true</suspect-protocol-enabled>
         <suspect-buffer-size>10M</suspect-buffer-size>
         <suspect-buffer-length>10000</suspect-buffer-length>
         <nominal-buffer-size>2M</nominal-buffer-size>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
            <nominal-buffer-length>2000</nominal-buffer-length>
            <limit-buffer-size>95M</limit-buffer-size>
            <limit-buffer-length>60000</limit-buffer-length>
         </tcp-acceptor>
      </acceptor-config>
      <autostart>true</autostart>
   </proxy-scheme>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

3

Using TLS to Secure Communication

Configure identity, trust, peer-name verification, and certificate lifecycle for each Coherence transport.
TLS protects a connection using the socket provider selected for that connection. Inventory cluster transport, Extend, gRPC, REST, management, metrics, and optional listeners independently. A provider configured for one service does not automatically secure every listener.

Start with Configuring TLS with Protected Password Sources and Verifying and Rotating TLS Identities. The retained topics describe socket-provider placement, custom keystore/key/certificate loaders, password sources, certificate usages, and refresh in more detail.

One-way TLS and mutual TLS

Server-authenticated TLS presents the server identity to a client that trusts its issuer and verifies the server name. mTLS additionally requires the client to present an accepted certificate. Decide which certificate identity maps to an application subject and which application operations it authorizes; a successful handshake alone does not create a cache role policy.

For cluster transport, each member can act as both client and server. Provision appropriate certificate usages and names for both roles. Configure well-known addresses when using the SSL/TCP cluster provider; multicast sockets are not supported by that provider. Host allowlists complement cryptographic identity but do not provide it.

Trust algorithm choice

The Coherence `PeerX509` algorithm trusts exact certificates in its trust store and has narrower certificate-validation semantics than a general PKI trust manager. The retained PeerX509 topic documents that additional certificate attributes and signatures are not checked in the same way. Choose the algorithm whose actual checks match the deployment; sharing a trust-store file does not define those checks by itself.

Keep WebLogic secured production mode scoped to the WebLogic integration chapter. For standalone Coherence, explicitly select socket providers and the independent security profile.

- Configuring TLS with Protected Password Sources
- Verifying and Rotating TLS Identities
- Overview of SSL/TLS
  SSL/TLS is a security protocol that secures communication between entities (typically, clients and servers) over a network. SSL/TLS works by authenticating clients and servers using digital certificates and by encrypting and decrypting communication using unique keys that are associated with authenticated clients and servers.
- Coherence Socket Providers
  Coherence communication is configured using Socket Providers. The `<socket-providers>` section of the operational configuration file contains zero or more named `<socket-provider>` elements.

Copyright © 2008, 2026, Oracle and/or its affiliates.

- Resolving the Socket Provider URL
  Some elements in a socket provider configuration are URLs. For example, the `<url>` element within the `<key-store>` element.
- Using a Socket Provider in Configuration
  You can configure various places in Coherence configuration files using `<socket-provider>`. You can configure `<socket-provider>` in one of two ways: either a named reference to a named socket provider in the operational configuration file, or as an in-line socket-provider configuration.
- Using SSL to Secure Cluster Communication
  In a Coherence cluster, all the cluster members communicate with each other over TCP in a peer-to-peer network. Each JVM is both a server that receives connections from other cluster members and a client that connects to other cluster members.
- Using SSL to Secure Extend and gRPC Client Communication
  Oracle Coherence supports SSL to secure communication between Coherence Extend and gRPC clients and a cluster side Extend or gRPC proxy.
- Configure a Default Socket Provider for a Cache Configuration File
  You can configure a socket provider in the `<defaults>` section of the cache configuration file. This socket provider will then apply to all the schemes in that configuration file that do not specifically configure their own socket provider, such as any remote cache services, remote invocation services, proxy services, gRPC services, and so on.
- Configuring a .NET Client-Side Stream Provider
- Securing the C++ Client with SSL/TLS
  The Coherence C++ Extend Client does not officially support SSL/TLS. However, you can use one of the following options to work around this limitation to run C++ extend clients securely against an SSL/TLS enabled Coherence proxy server.
- Using SSL to Secure Federation Communication
  Oracle Coherence supports using SSL to secure communication between cluster participants in a federated cluster. Communication is secured between federated service members and requires SSL to be configured on each cluster participant.
- Coherence PeerX509 Algorithm
  Oracle Coherence includes a proprietary peer trust algorithm, PeerX509, which works by assuming trust (and only trust) of the certificates that are in the trust manager keystore. It also leverages the peer-to-peer protocol features of TCMP. Specifically, for the SSL negotiation to succeed, the certificate received must be the same as one of the certificates held by the trust manager .
- Specifying a Global Socket Provider
  You can configure a global socket provider in the Coherence operational configuration file. When set, every server or client socket that Coherence creates will use this configuration unless it has been overridden with a specific socket provider of its own.
- Specifying Passwords in Socket Provider Configuration
  Java keystores and private keys can be secured with credentials, typically a password. The socket provider configuration provides several ways to specify a password. It is up to the application developer to choose the most suitable approach based on the required level of security versus simplicity of configuration.
- Controlling Cipher Suite and Protocol Version Usage
- Using Host Name Verification
- Configuring Client Authentication
  You can use the `<client-auth>` element to specify whether a SSL/TLS socket provider should use one-way or two-way SSL/TLS authentication.

Copyright © 2008, 2026, Oracle and/or its affiliates.

- Using Private Key and Certificate Files
- Using Custom Keystore, Private Key, and Certificate Loaders
- Using Refreshable KeyStores, Private Keys, and Certificates

Configuring TLS with Protected Password Sources

Use named SSL socket providers and OS-protected password files rather than embedding secrets in XML.
Provision member/server PKCS12 identities, a PKCS12 trust store containing the approved CA material, and separate password files with permissions limited to the service account. Use certificates with the DNS/IP names and extended usages required by the actual connections. The example assumes the normal supported JCA algorithms named below.

This operational override defines an mTLS `server-tls` provider. Replace the paths with mounted deployment paths. The store, private key, and trust-store passwords are different configuration locations even when a deployment happens to use the same value.

```
<coherence xmlns="http://xmlns.oracle.com/coherence/coherence-operational-
config">
  <cluster-config>
    <socket-providers>
      <socket-provider id="server-tls">
        <ssl>
          <protocol>TLS</protocol>
          <identity-manager>
            <algorithm>SunX509</algorithm>
            <key-store>
              <url>file:/run/coherence/identity.p12</url>
              <password-provider><name>store-password</name></password-
provider>
              <type>PKCS12</type>
            </key-store>
            <password-provider><name>key-password</name></password-provider>
          </identity-manager>
          <trust-manager>
            <algorithm>PKIX</algorithm>
            <key-store>
              <url>file:/run/coherence/trust.p12</url>
              <password-provider><name>trust-password</name></password-
provider>
              <type>PKCS12</type>
            </key-store>
          </trust-manager>
          <hostname-verifier><action>default</action></hostname-verifier>
          <socket-provider>tcp</socket-provider>
          <client-auth>required</client-auth>
          <refresh-period>5m</refresh-period>
        </ssl>
      </socket-provider>
    </socket-providers>
    <password-providers>
      <password-provider id="store-password">
        <class-name>com.tangosol.net.URLPasswordProvider</class-name>
        <init-params>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
          <init-param><param-type>java.lang.String</param-type><param-
value>file:/run/secrets/coherence-store</param-value></init-param>
          <init-param><param-type>boolean</param-type><param-value>true</
param-value></init-param>
        </init-params>
      </password-provider>
      <password-provider id="key-password">
        <class-name>com.tangosol.net.URLPasswordProvider</class-name>
        <init-params>
          <init-param><param-type>java.lang.String</param-type><param-
value>file:/run/secrets/coherence-key</param-value></init-param>
          <init-param><param-type>boolean</param-type><param-value>true</
param-value></init-param>
        </init-params>
      </password-provider>
      <password-provider id="trust-password">
        <class-name>com.tangosol.net.URLPasswordProvider</class-name>
        <init-params>
          <init-param><param-type>java.lang.String</param-type><param-
value>file:/run/secrets/coherence-trust</param-value></init-param>
          <init-param><param-type>boolean</param-type><param-value>true</
param-value></init-param>
        </init-params>
      </password-provider>
    </password-providers>
  </cluster-config>
</coherence>
```

The second constructor argument makes the provider use only the first line. Without it, a trailing newline can become part of a password. Readable secret files are still secrets; file indirection removes passwords from XML but does not encrypt the files or authorize broad filesystem access.

Client and server variants

For a server-authenticated-only listener, retain the identity and explicitly choose `client-auth=none` where appropriate. A trust-only client provider contains the trust manager, hostname verifier, and underlying TCP provider, and does not need a private key. For mTLS clients, configure their own identity and trust under `client-tls`. Do not share a private identity across members merely to simplify copying the example.

Reference the named provider from the target Extend TCP acceptor/initiator, gRPC acceptor/channel, or HTTP acceptor. For cluster transport use `<cluster-config><unicast-listener><socket-provider>...</socket-provider></unicast-listener>` and the required well-known-address configuration. Set these choices consistently across members before replacing an unsecured transport.

The `<ssl>` configuration element retains its historical name even when the configured protocol is TLS. Use supported JDK/provider protocol and cipher policy; the element name is not a reason to enable obsolete SSL protocol versions.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Verifying and Rotating TLS Identities

Verify certificate trust and peer names separately, then rotate material with an explicit overlap and connection strategy.

Hostname verification

The default verifier checks the peer name against the supported certificate names. Configure it explicitly in secure recipes. In compatibility mode, an absent/null or unsafe `allow` verifier can permit a mismatch with a bounded would-reject warning. Hardened policy enforces the default and rejects an explicit unsafe allow configuration, subject to the documented shipped-provider fallback. A configured custom verifier retains its own contract.

Use the DNS name clients actually connect to in the certificate. A private CA or self-signed certificate still needs explicit trust and correct name validation. Do not treat an IP address, service alias, load-balancer address, and backend pod hostname as interchangeable certificate names.

Verification procedure

1. Connect using the intended provider, trust material, and DNS name, and perform a real application operation.
2. Replace trust with an unrelated test CA and confirm handshake failure.
3. Connect using a name absent from the certificate and confirm failure with the configured verifier.
4. For mTLS, test missing/untrusted client certificates as well as a valid one. Verify the expected application identity and a denied operation separately.
5. Repeat for each listener, client language, and load-balancer route used by the deployment.

Rotation and refresh

Existing custom keystore, private-key, and certificate loaders can be combined with a refresh period. A nonzero refresh period enables the configured refresh behavior; zero is the default disabled refresh interval. Check the loader's refreshability and the provider's failure behavior rather than assuming every custom source refreshes automatically.

Stage new CA trust on receivers before activating identities issued by that CA. Publish a complete matching key/certificate/store/password set atomically where the secret source allows it. Confirm successful refresh and a fresh connection using the new identity. Keep the old trust during the overlap needed by remaining peers, then retire it deliberately.

Refreshing an SSL context does not prove every existing long-lived connection renegotiates or that every revoked identity is disconnected. Plan connection replacement or rolling restarts when immediate retirement is required. Test expiry, missing/mismatched files, and a failed refresh before relying on last-known-good or fail-closed behavior from a particular loader.

OCSP/CRL behavior, Kubernetes mount replacement, and cloud secret-manager integration depend on the selected JDK/provider/loader. Configure and test the exact deployment combination. Do not claim revocation merely from a trust-store reload or assume a symbolic-link secret update is atomic across all its files.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Overview of SSL/TLS

SSL/TLS is a security protocol that secures communication between entities (typically, clients and servers) over a network. SSL/TLS works by authenticating clients and servers using digital certificates and by encrypting and decrypting communication using unique keys that are associated with authenticated clients and servers.

This section covers a brief description of SSL/TLS and some of the terms that will be used in the rest of this chapter.

Establishing Identity

The identity of an entity is established by using a digital certificate and public and private encryption keys. The digital certificate contains general information about the entity and contains the public encryption key embedded within it.

In Coherence, identity is controlled by an identity manager, which corresponds to an identity manager in a Java SSL context.

Establishing Trust

A digital certificate is verified by a Certificate Authority (CA) and signed using the CA's digital certificate. The CA's digital certificate establishes trust that the entity is authentic. When a connection is made, the received certificate is verified against the CA certificate's configured trust store or the JVM's default trust store.

In Coherence, trust is controlled by a trust manager configuration, which corresponds to a trust manager in a Java SSL context.

Encrypting and Decrypting Data

The digital certificate for an entity contains a public encryption key that is paired with a private encryption key. Certificates are passed between entities during an initial connection. Data is then encrypted using the public key. Data that is encrypted using the entity public key can only be decrypted using the entity private key. This ensures that only the entity that owns the public encryption key can decrypt the data.

One-Way Authentication and Two-Way Authentication

SSL communication between clients and servers is set up using either one-way or two-way authentication.

In one-way authentication, a server is required to identify itself to a client by sending its digital certificate for authentication. The client is not required to send the server a digital certificate and remains anonymous to the server.

In two-way authentication, both the client and the server must send their respective digital certificates to each other for mutual authentication. Two-way authentication provides stronger security by assuring that the identity on each side of the communication is known. Two-way TLS is also called mutual TLS (mTLS).

Oracle Coherence supports both one-way and two-way SSL. Configuration depends on various factors, such as whether this is for cluster membership, Extend or gRPC proxies, or Extend or gRPC clients.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Certificates With Extended Usage

You can add a extended usage field to restrict the uses of a certificate. The extended usage is typically one or more values. When securing Coherence communication, the extended usage must include the correct usage, typically either `serverAuth` or `clientAuth`.

The values of extended usage will differ depending on the type of Coherence communication and the specific SSL scenario. If an application only has access to certificates that are single use (that is, only `serverAuth` or only `clientAuth`), then this restricts which available SSL configurations can be used and whether mTLS or one-way TLS can be used.

The following list shows the extended usage required for certificates used to secure different parts of Coherence.

- Cluster Membership using mTLS requires both `serverAuth` and `clientAuth`
- Cluster Membership using one-way SSL requires `serverAuth`
- Extend or gRPC proxies require `serverAuth`
- Extend or gRPC clients require `clientAuth`
- Federation using mTLS requires both `serverAuth` and `clientAuth`
- Federation using one-way SSL requires `serverAuth`
- Management over REST HTTP endpoint requires `serverAuth`
- Metrics HTTP endpoint requires `serverAuth`
- Coherence REST HTTP proxies require `serverAuth`

Coherence has a custom trust protocol called PeerX509 which is similar to mTLS but does not validate extended usage. Socket providers configured with this algorithm will work with any extended usage certificate. See Coherence PeerX509 Algorithm.

Coherence Socket Providers

Coherence communication is configured using Socket Providers. The `<socket-providers>` section of the operational configuration file contains zero or more named `<socket-provider>` elements.

To name the `<socket-providers>` element, set its `id` attribute. For example, if you specify `<socket-providers id="ssl-config">`, the socket provider configuration is named `ssl-config` and it can then be referenced from other parts of the Coherence operational or cache configuration files.

There are different types of socket providers in Coherence, and to use SSL, an `<ssl>` socket provider needs to be configured. Depending on the required security scenario, there are several XML elements that can be added to the `<ssl>` element. The most common are `<identity-manager>` and `<trust-manager>`.

Example 3-1 shows a basic mTLS `<ssl>` socket provider that is configured with an `<identity-manager>` keystore named `server.jks` that holds the private key and certificate to establish this JVMs identity, and a `<trust-manager>` keystore named `trust.jks` that holds the CA certificate to validate the certificates of client connections.

Example 3-1 Basic mTLS socket provider configuration

```
<socket-provider id="ssl-config">
    <ssl>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
            <identity-manager>
                <key-store>
                    <url>file:server.jks</url>
                </key-store>
            </identity-manager>
            <trust-manager>
                <key-store>
                    <url>file:trust.jks</url>
                </key-store>
            </trust-manager>
        </ssl>
</socket-provider>
```

Although Example 3-1 uses Java keystores, it is also possible to configure a socket provider to directly use private key and certificate files as well as plug in custom providers that can obtain keystores, keys or certificates from any location. See Using Private Key and Certificate Files.

Additionally, Example 3-1 does not configure any passwords for keystores or keys. Configuring passwords is covered in Specifying Passwords in Socket Provider Configuration.

- Configuring the Identity Manager
  Use the `<identity-manager>` section of a socket provider to configure the identity of the socket. An identity manager requires a private key and a corresponding certificate. These can be provided either in a Java keystore, or individually as separate key and certificate files.
- Configuring a Trust Manager
  When configuring a socket provider, a trust manager requires one or more CA certificates to verify trust. These can be provided either in a Java keystore, or individually as separate certificate files.

Configuring the Identity Manager

Use the `<identity-manager>` section of a socket provider to configure the identity of the socket. An identity manager requires a private key and a corresponding certificate. These can be provided either in a Java keystore, or individually as separate key and certificate files.

> **Note**
>
> Some of the following examples include hard-coded password values. In production environments, avoid using hard-coded passwords which are insecure. Coherence provides several alternative ways to provide passwords. See Specifying Passwords in Socket Provider Configuration.

Using a Java keystore

To use a Java keystore containing the key and certificate pair, configure the `<key-store>` element inside the `<identity-manager>` element.

In Example 3-2, the `<identity-manager>` loads the key and certificate from a keystore file named `server.jks`.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Example 3-2 <identity-manager> configuration

```
<identity-manager>
   <key-store>
       <url>file:server.jks</url>
   </key-store>
</identity-manager>
```

Password Protected keystores

If the keystore is password protected, then specify the password using one of the password configuration options in the <key-store> element.

In Example 3-3, the configuration uses the <password> element in the <key-store> to set the password as foo.

Example 3-3 <identity-manager> configuration whose keystore is password protected

```
<identity-manager>
   <key-store>
       <url>file:server.jks</url>
       <password>foo</password>
   </key-store>
</identity-manager>
```

Using Private Key and Certificate Files

To use key and certificate files directly, configure the <key> and <cert> elements inside the <identity-manager> element.

In Example 3-4, the <identity-manager> loads the private key from a file named server.key and the certificate from a file named server.cert.

Example 3-4 <identity-manager> configuration that loads private key and certificate files

```
<identity-manager>
    <key>server.key</key>
    <cert>server.cert</cert>
</identity-manager>
```

Password Protected Private Keys

If the private key is encrypted and protected with a password, the credentials need to be configured in the <identity-manager> configuration using one of the supported password options.

In Example 3-5, the identity manager is configured to read the private key from a Java keystore named server.jks. The private key is protected with a password foo, which is supplied using the <password> element inside the <identity-manager> element.

Example 3-5 <identity-manager> configuration using password protected private keys

```
<identity-manager>
   <key-store>
       <url>file:server.jks</url>
  </key-store>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
  <password>foo</password>
</identity-manager>
```

> **Note**
>
> When using Java keystores where both the keystore and private key require a password, it is important to make sure that the two passwords are not confused in the configuration. The keystore password is inside the `<key-store>` element. The private key password is at the higher level inside the `<identity-manager>` element.

In Example 3-6, the identity manager is configured to read the private key from a file named `server.key`. The private key is protected with a password `foo`. The password is supplied using the `<password>` element inside the `<identity-manager>` element.

Example 3-6 `<identity-manager>` configuration using a private key

```
<identity-manager>
    <key>server.key</key>
    <cert>server.cert</cert>
    <password>foo</password>
</identity-manager>
```

Configuring a Trust Manager

When configuring a socket provider, a trust manager requires one or more CA certificates to verify trust. These can be provided either in a Java keystore, or individually as separate certificate files.

> **Note**
>
> Some of the following examples use hard coded password values. In production environments, avoid using hard coded passwords, which are insecure. Coherence has several alternative ways to provide passwords. See Specifying Passwords in Socket Provider Configuration.

Using a Java keystore

To use a Java keystore containing the CA certificates, configure the `<key-store>` element inside the `<trust-manager>` element.

In Example 3-7, the `<trust-manager>` element loads the CA certificates from a keystore file named `trust.jks`.

Example 3-7 `<trust-manager>` configuration

```
<identity-manager>
   <key-store>
       <url>file:trust.jks</url>
   </key-store>
</identity-manager>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

Password Protected keystores

If the keystore is password protected, then specify the password using one of the password configuration options in the `<key-store>` element.

In Example 3-8, the configuration uses the `<password>` element in the `<key-store>` to set the password as `foo`.

Example 3-8 `<trust-manager>` configuration whose keystore is password protected

```
<trust-manager>
   <key-store>
       <url>file:server.jks</url>
       <password>foo</password>
   </key-store>
</trust-manager>
```

Using Certificate Files

To use the certificate files directly, configure the `<cert>` elements inside the `<trust-manager>` element. The `<trust-manager>` element can contain multiple `<cert>` elements.

In Example 3-9, the `<trust-manager>` loads the certificates from the files named `ca-one.cert` and `ca-two.cert`.

Example 3-9 `<trust-manager>` configuration using certificate files

```
<trust-manager>
    <cert>ca-one.cert</cert>
    <cert>ca-two.cert</cert>
</trust-manager>
```

Resolving the Socket Provider URL

Some elements in a socket provider configuration are URLs. For example, the `<url>` element within the `<key-store>` element.

The following is an explanation of how the values of these elements are processed to locate the resources they refer to:

1. The value of the XML element is converted to a Java URI.
2. If the value is a valid URI and has a URI scheme, for example `file:` or `http:`, then it is assumed to be a valid URI and Coherence will try to open a stream to this URI to read the data.
3. If the value has no scheme, then Coherence treats it as a file on the file system or on the class path. Coherence will first assume that the value is a file name (either fully qualified or relative to the working directory) and try to locate that file. If this fails, Coherence will try to find the same file as a resource on the class path.

Using a Socket Provider in Configuration

You can configure various places in Coherence configuration files using `<socket-provider>`. You can configure `<socket-provider>` in one of two ways: either a named reference to a

Copyright © 2008, 2026, Oracle and/or its affiliates.

named socket provider in the operational configuration file, or as an in-line socket-provider configuration.

Example 3-10 demonstrates an Extend proxy service in a cache configuration file. The proxy scheme is configured with a `<socket-provider>` element with a value of `mtls` which references the socket provider named `mtls` in the operational configuration file.

Example 3-10 Extend proxy service that references `mtls` socket provider

```
<proxy-scheme>
    <scheme-name>proxy</scheme-name>
    <service-name>Proxy</service-name>
    <acceptor-config>
        <tcp-acceptor>
            <socket-provider>mtls</socket-provider>
        </tcp-acceptor>
    </acceptor-config>
</proxy-scheme>
```

Example 3-11 demonstrates an Extend proxy service in a cache configuration file. The proxy scheme is configured with a `<socket-provider>` element containing the full socket provider configuration.

Example 3-11 Extend proxy service with inline socket provider configuration

```
<proxy-scheme>
    <scheme-name>proxy</scheme-name>
    <service-name>Proxy</service-name>
    <acceptor-config>
        <tcp-acceptor>
            <socket-provider>
              <ssl>
                <identity-manager>
                    <key-store>
                        <url>file:server.jks</url>
                    </key-store>
                </identity-manager>
                <trust-manager>
                    <key-store>
                        <url>file:trust.jks</url>
                    </key-store>
                </trust-manager>
              </ssl>
            </socket-provider>
        </tcp-acceptor>
    </acceptor-config>
</proxy-scheme>
```

- Configure a Socket Provider at Runtime
  When using named socket providers configured in the operational configuration file, you can change the socket provider used in a configuration at runtime based on Java system properties.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Configure a Socket Provider at Runtime

When using named socket providers configured in the operational configuration file, you can change the socket provider used in a configuration at runtime based on Java system properties.

The optional `system-property` attribute of the `<socket-provider>` element specifies the name of the system property used to obtain the socket provider name at runtime. This allows flexibility to choose at runtime what sort of sockets are used. For example, developers can use plain TCP in development testing, without worrying about creating keys and certificates. Then, later in system testing and production, they can specify an SSL socket provider name.

Example 3-12 shows an Extend proxy service in a cache configuration file. The proxy scheme is configured with a `<socket-provider>` element without a value but with a `system-property` attribute set to `proxy.socket.provider`. By default, as the `<socket-provider>` element has no value, no provider will be set and the proxy will use plain TCP sockets.

Example 3-12 Configuration for an Extend proxy service configured to use plain TCP sockets

```
<proxy-scheme>
    <scheme-name>proxy</scheme-name>
    <service-name>Proxy</service-name>
    <acceptor-config>
        <tcp-acceptor>
            <socket-provider system-property="proxy.socket.provider"/>
        </tcp-acceptor>
    </acceptor-config>
</proxy-scheme>
```

If the JVM is started with the system property set, then that property value will be used as the socket provider name. For example, starting Coherence with `-Dproxy.socket.provider=mtls` will use `mtls` as the socket provider name (assuming there is a socket provider named `mtls` configured in the operational configuration file).

Example 3-13 shows an Extend proxy service in a cache configuration file. The proxy scheme is configured with a `<socket-provider>` element with a value of `mtls` and with a `system-property` attribute set to `proxy.socket.provider`. By default, the socket provider named `mtls` from the operational configuration will be used. If the `proxy.socket.provider` system property is set, then the property value will be used as the socket provider name.

Example 3-13 Configuration for an Extend proxy service configured to use a referenced socket provider

```
<proxy-scheme>
    <scheme-name>proxy</scheme-name>
    <service-name>Proxy</service-name>
    <acceptor-config>
        <tcp-acceptor>
            <socket-provider system-property="proxy.socket.provider">
                mtls
            </socket-provider>
        </tcp-acceptor>
    </acceptor-config>
</proxy-scheme>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

Using SSL to Secure Cluster Communication

In a Coherence cluster, all the cluster members communicate with each other over TCP in a peer-to-peer network. Each JVM is both a server that receives connections from other cluster members and a client that connects to other cluster members.

In addition, it is important to realize that TCMP is a peer-to-peer protocol that generally runs in trusted environments where many cluster nodes are expected to remain connected with each other. SSL negotiation is performed once, when the connection is made, and then the connection remains connected for the lifetime of the two cluster member JVMs involved. When configuring SSL, carefully consider the implications on key and certificate administration and on performance.

The socket provider used to control cluster traffic is configured by setting the `<socket-provider>` element inside the `<unicast-listener>` element of the cluster configuration in the operational configuration file.

In Example 3-14, the XML operational configuration file sets the unicast socket provider name to `ssl-config` which is a reference to the socket provider named `ssl-config` in the `<socket-providers>` section.

The actual socket provider configuration will depend on the security requirements of the application.

Example 3-14 Configuration where clusters use a socket provider to configure SSL/TLS

```
<?xml version='1.0'?>

<coherence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://xmlns.oracle.com/coherence/coherence-operational-config"
    xsi:schemaLocation="http://xmlns.oracle.com/coherence/coherence-
operational-config
    coherence-operational-config.xsd">
    <cluster-config>
        <unicast-listener>
            <socket-provider>ssl-config</socket-provider>
        </unicast-listener>

        <socket-providers>
            <socket-provider id="ssl-config">
                <ssl>
                <!-- Actual config omitted for brevity -->
                </ssl>
            </socket-provider>
        </socket-providers>
    </cluster-config>
</coherence>
```

- Cluster Communication Using mTLS
  The most common and recommended configuration for using SSL to secure cluster communication is mTLS (or two-way TLS). To configure an `<ssl>` socket provider for mTLS, both identity and trust must be configured.

Copyright © 2008, 2026, Oracle and/or its affiliates.

- Cluster Communication with One-Way SSL
  You can configure one-way SSL for cluster members so that a cluster member will verify trust of a server certificate that it receives when making a connection to another cluster member. A cluster member will not verify trust for a member that connects to it.

Cluster Communication Using mTLS

The most common and recommended configuration for using SSL to secure cluster communication is mTLS (or two-way TLS). To configure an `<ssl>` socket provider for mTLS, both identity and trust must be configured.

In Example 3-15, the socket provider `mtls-config` is configured with an `<identity-manager>` element containing a keystore named `server.jks` and a `<trust-manager>` element containing a keystore named `trust.jks`. The `<unicast-listener>` `<socket-provider>` element is then set to `mtls-config` to reference the SSL socket provider.

Example 3-15 Configuration for clusters communicating over mTLS

```
<?xml version='1.0'?>

<coherence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://xmlns.oracle.com/coherence/coherence-operational-config"
   xsi:schemaLocation="http://xmlns.oracle.com/coherence/coherence-
operational-config
   coherence-operational-config.xsd">
   <cluster-config>
      <unicast-listener>
         <socket-provider>mtls-config</socket-provider>
     </unicast-listener>

      <socket-providers>
         <socket-provider id="mtls-config">
           <ssl>
              <identity-manager>
                 <key-store>
                     <url>file:server.jks</url>
                 </key-store>
              </identity-manager>
               <trust-manager>
                 <key-store>
                     <url>file:trust.jks</url>
                 </key-store>
               </trust-manager>
           </ssl>
        </socket-provider>
      </socket-providers>
   </cluster-config>
</coherence>
```

By default, when a `<ssl>` socket provider is configured with both identity and trust, Coherence will create a Java SSL context that is configured for mTLS.

Copyright © 2008, 2026, Oracle and/or its affiliates.

> **Note**
>
> When using a certificate with extended usage set in this configuration, the extended usage must include both `serverAuth` and `clientAuth`. The socket provider configuration is used to configure a single Java SSL Context used by both the cluster member's server sockets (used to receive connections from other cluster members) and its client sockets (used to connect to other cluster members).

Cluster Communication with One-Way SSL

You can configure one-way SSL for cluster members so that a cluster member will verify trust of a server certificate that it receives when making a connection to another cluster member. A cluster member will not verify trust for a member that connects to it.

Only a single socket provider can be configured for the unicast listener, therefore the `<ssl>` socket provider must be configured with both identity and trust, the same as with mTLS. To specify one-way SSL, add a `<client-auth>` element and set its value ot `none`.

The `<client-auth>` element configures the corresponding setting in the Java SSL context which determines whether the client must send a certificate. It has three possible values:

- `none` - the client does not send a certificate (even if configured with an identity key and certificate)

  If you want clusters to communicate over one-way SSL, then set `<client-auth>` to `none`.
- `wanted` - the client may send a certificate if it has one
- `required` - the client must send a certificate.

  If you want clusters to communicate over mTLS, then set `<client-auth>` to `required`.

Example 3-16 shows the unicast listener configuration for one-way SSL.

Example 3-16 Unicast Listener Configuration for One-way SSL

```
<?xml version='1.0'?>

<coherence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://xmlns.oracle.com/coherence/coherence-operational-config"
   xsi:schemaLocation="http://xmlns.oracle.com/coherence/coherence-
operational-config
   coherence-operational-config.xsd">
   <cluster-config>
      <unicast-listener>
         <socket-provider>one-way-config</socket-provider>
     </unicast-listener>

      <socket-providers>
         <socket-provider id="one-way-config">
           <ssl>
              <identity-manager>
                 <key-store>
                     <url>file:server.jks</url>
                 </key-store>
              </identity-manager>
               <trust-manager>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
                <key-store>
                    <url>file:trust.jks</url>
                </key-store>
              </trust-manager>
<!-- This element changes the configuration to one-way SSL -->
              <client-auth>none</client-auth>
            </ssl>
          </socket-provider>
      </socket-providers>
   </cluster-config>
</coherence>
```

> **Note**
>
> When using a certificate with extended usage set in this configuration, the extended usage must include `serverAuth`. In one-way SSL, only the server sends a certificate to the client, so the certificate must be valid for use as a server certificate.

Using SSL to Secure Extend and gRPC Client Communication

Oracle Coherence supports SSL to secure communication between Coherence Extend and gRPC clients and a cluster side Extend or gRPC proxy.

When SSL is used to secure communication between clients and proxies, it requires configuration on both the client side and the cluster side. SSL is supported for both Java and .NET Extend clients, but not for C++ Extend clients (without additional configuration as described in Securing the C++ Client with SSL/TLS. SSL is supported for all types of gRPC client.

Both mTLS and one-way SSL can be configured for clients and proxies.

- Configuring a Cluster-Side Extend Proxy SSL Socket Provider
  You can configure SSL in the cluster-side cache configuration file by defining an SSL socket provider for a proxy service.
- Configuring the Cluster-Side gRPC Proxy SSL Socket Provider
  The Coherence gRPC Proxy is configured using an internal proxy cache configuration file.
- Configuring a Java Extend or gRPC Client SSL Socket Provider
  You can configure SSL in the Extend or gRPC client cache configuration file by defining an SSL socket provider for a remote scheme.

Configuring a Cluster-Side Extend Proxy SSL Socket Provider

You can configure SSL in the cluster-side cache configuration file by defining an SSL socket provider for a proxy service.

There are two options for configuring an SSL socket provider depending on the level of granularity that is required.

- Configure the socket provider by proxy service, where each proxy service defines an SSL socket provider configuration or references a predefined configuration that is included in the operational configuration file.

Copyright © 2008, 2026, Oracle and/or its affiliates.

- Configure all proxy services to use the same SSL socket provider configuration by configuring a socket provider in the cache configuration `<defaults>` section.

A proxy service that provides its own configuration overrides the configuration in the `<defaults>` section. The socket provider configuration in the `<defaults>` section can reference a named socket provider configuration that is included in the operational configuration file or be a full in-line socket provider configuration.

> **Note**
>
> When using a certificate with extended usage set in a cluster side proxy socket provider configuration, the extended usage must include `serverAuth`. A proxy opens server sockets to receive client connections so the certificate must be valid for server use.

Configure an SSL Socket Provider per Extend Proxy Service

To configure an SSL socket provider for an Extend proxy service, add a `<socket-provider>` element within the `<tcp-acceptor>` element of each `<proxy-scheme>` definition.

Example 3-17 demonstrates a proxy scheme that configures an SSL socket provider directly in the proxy configuration in the cache configuration file.

Example 3-17 Configuration for an SSL/TLS socket provider per Extend proxy service

```
<proxy-scheme>
   <service-name>ProxyService</service-name>
   <acceptor-config>
      <tcp-acceptor>
         <socket-provider>
            <ssl>
              <identity-manager>
                 <key-store>
                     <url>file:server.jks</url>
                </key-store>
              </identity-manager>
               <trust-manager>
                 <key-store>
                     <url>file:trust.jks</url>
                 </key-store>
               </trust-manager>
           </ssl>
         </socket-provider>
     </tcp-acceptor>
   </acceptor-config>
   <autostart>true</autostart>
</proxy-scheme>
```

Example 3-18 demonstrates configuring the proxy in the cache configuration to reference a named socket provider in the operational configuration file. In this case, the proxy will use the socket provider named `ssl-config`.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Example 3-18 Configuration for a single SSL/TLS socket provider for all Extend proxy services

```
<proxy-scheme>
   <service-name>ProxyService</service-name>
   <acceptor-config>
      <tcp-acceptor>
         <socket-provider>ssl-config</socket-provider>
     </tcp-acceptor>
   </acceptor-config>
   <autostart>true</autostart>
</proxy-scheme>
```

Extend or gRPC Proxy with mTLS

A proxy can be configured for mTLS using a <ssl> socket provider configured with both identity and trust.

Example 3-19 shows a socket provider configured for mTLS. By default, when a socket provider has both identity and trust configured, it will configure the SSL context to use two-way SSL.

> **Note**
>
> If a proxy is configured for mTLS, then the client must also be configured for mTLS.

Example 3-19 Configuration for an Extend or gRPC Proxy using mTLS

```
<socket-provider id="mtls-config">
  <ssl>
     <identity-manager>
        <key-store>
            <url>file:server.jks</url>
        </key-store>
     </identity-manager>
      <trust-manager>
        <key-store>
            <url>file:trust.jks</url>
        </key-store>
      </trust-manager>
  </ssl>
</socket-provider>
```

Extend or gRPC Proxy with One-Way SSL

A proxy can be configured for one-way SSL using a `<ssl>` socket provider configured with only identity. In one-way SSL, the server sends a certificate to the client, which the client verifies against its trust store. When configuring one-way SSL it is important to set the socket provider configuration correctly, that is, the server only has `<identity-manager>` configured and the client has a `<trust-manager>` configured.

Example 3-20 shows a cluster side proxy socket provider configured for one-way SSL.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Example 3-20 Configuration for an Extend or gRPC Proxy using one-way SSL/TLS

```
<socket-provider id="oneway-proxy-config">
  <ssl>
     <identity-manager>
        <key-store>
            <url>file:server.jks</url>
        </key-store>
     </identity-manager>
 </ssl>
</socket-provider>
```

An alternative way to configure a cluster side proxy to use one-way SSL in a socket provider configured with both `<identity-manager>` and `<trust-manager>` is to set the `<client-auth>` element to `none`.

Example 3-21 shows a cluster side socket provider configured with both identity and trust which would normally be mTLS, but with the `<client-auth>` element set to `none` will be one-way and not require the client to send a certificate.

If a proxy is configured for one-way SSL, then the client may be configured with either an mTLS or a one-way configuration. If the client is configured as two-way (that is, it has identity and trust) it will still connect and verify the server certificate, but it will not send its own certificate.

Example 3-21 Configuration for an Extend or gRPC Proxy using one-way SSL/TLS by setting `<client-auth>` to `none`

```
<socket-provider id="one-way-config">
  <ssl>
     <identity-manager>
        <key-store>
            <url>file:server.jks</url>
        </key-store>
     </identity-manager>
      <trust-manager>
        <key-store>
            <url>file:trust.jks</url>
        </key-store>
      </trust-manager>
      <!-- This element changes the configuration to one-way SSL -->
      <client-auth>none</client-auth>
  </ssl>
</socket-provider>
```

Configuring the Cluster-Side gRPC Proxy SSL Socket Provider

The Coherence gRPC Proxy is configured using an internal proxy cache configuration file.

To configure SSL for the gRPC proxy, configure a named socket provider in the operational configuration file, then set the `coherence.grpc.server.socketprovider` system property (or environment variable) to the name of that socket provider.

The Coherence operational configuration contains a special gRPC socket provider configuration named `grpc-insecure`. This configures the default gRPC Java insecure credentials for use by the proxy or client.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Configuring a Java Extend or gRPC Client SSL Socket Provider

You can configure SSL in the Extend or gRPC client cache configuration file by defining an SSL socket provider for a remote scheme.

There are two options for configuring an SSL socket provider, depending on the level of granularity that is required.

- Configure the socket provider per remote scheme, where each remote scheme defines an SSL socket provider configuration or references a predefined configuration that is included in the operational configuration file.
- Configure all remote schemes to use the same SSL socket provider configuration by configuring a socket provider in the cache configuration `<defaults>` section.

A remote service that provides its own configuration overrides the configuration in the `<defaults>` section. The socket provider configuration in the `<defaults>` section can reference a named socket provider configuration that is included in the operational configuration file or be a full in-line socket provider configuration.

> **Note**
>
> - When using certificate with extended usage set in an Extend or gRPC client socket provider configuration, the extended usage must include `clientAuth`.
> - If the cluster side proxy is configured to use mTLS, then the client must also be configured for mTLS. If the cluster side proxy is configured to use one-way SSL, then the client may be configured as either one-way or mTLS. This is because it is the server that determines whether a connection is two-way or one-way (that is, whether the client should send its identity certificate).

Configure an SSL Socket Provider per Remote Service

To configure an SSL socket provider for an Extend remote service, add a `<socket-provider>` element within the `<tcp-initiator>` element of a `<remote-cache-scheme>` definition or a `<remove-invocation-scheme>`.

To configure an SSL socket provider for a gRPC remote service, add a `<socket-provider>` element within the `<grpc-channel>` element of a `<remote-grpc-cache-scheme>` definition.

Example 3-22 demonstrates an Extend remote cache scheme that configures a socket provider that uses SSL. This example configures both an identity keystore (`server.jks`) and a trust keystore (`trust.jks`). This is typical of two-way SSL authentication, in which both the client and proxy must exchange digital certificates and confirm each other's identity.

Example 3-22 Configuration for an SSL/TLS socket provider per remote scheme

```
<?xml version="1.0"?>

<cache-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://xmlns.oracle.com/coherence/coherence-cache-config"
   xsi:schemaLocation="http://xmlns.oracle.com/coherence/coherence-cache-
config
   coherence-cache-config.xsd">
   <caching-scheme-mapping>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
      <cache-mapping>
         <cache-name>*</cache-name>
         <scheme-name>remote-cache</scheme-name>
      </cache-mapping>
   </caching-scheme-mapping>

   <caching-schemes>
      <remote-cache-scheme>
         <scheme-name>remote-cache</scheme-name>
         <service-name>RemoteService</service-name>
         <initiator-config>
            <tcp-initiator>
               <socket-provider>
                  <ssl>
                    <identity-manager>
                       <key-store>
                           <url>file:server.jks</url>
                       </key-store>
                    </identity-manager>
                     <trust-manager>
                       <key-store>
                           <url>file:trust.jks</url>
                       </key-store>
                     </trust-manager>
                  </ssl>
               </socket-provider>
            </tcp-initiator>
         </initiator-config>
      </remote-cache-scheme>
   </caching-schemes>
</cache-config>
```

Example 3-23 demonstrates a gRPC remote cache scheme that configures a socket provider that uses SSL. This example configures both an identity keystore (`server.jks`) and a trust keystore (`trust.jks`). This is typical of two-way SSL authentication, in which both the client and proxy must exchange digital certificates and confirm each other's identity.

Example 3-23 Configuration for a gRPC remote cache scheme that configures a socket provider to use SSL/TLS

```
<?xml version="1.0"?>

<cache-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://xmlns.oracle.com/coherence/coherence-cache-config"
   xsi:schemaLocation="http://xmlns.oracle.com/coherence/coherence-cache-
config
   coherence-cache-config.xsd">
   <caching-scheme-mapping>
      <cache-mapping>
         <cache-name>*</cache-name>
         <scheme-name>remote-cache</scheme-name>
      </cache-mapping>
   </caching-scheme-mapping>

   <caching-schemes>
      <remote-grpc-cache-scheme>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
            <scheme-name>remote-cache</scheme-name>
            <service-name>RemoteService</service-name>
            <grpc-channel>
                 <socket-provider>
                     <ssl>
                       <identity-manager>
                          <key-store>
                              <url>file:server.jks</url>
                          </key-store>
                       </identity-manager>
                        <trust-manager>
                          <key-store>
                              <url>file:trust.jks</url>
                          </key-store>
                        </trust-manager>
                    </ssl>
                 </socket-provider>
            </grpc-channel>
      </remote-grpc-cache-scheme>
   </caching-schemes>
</cache-config>
```

Example 3-24 configures remote schemes that references an SSL socket provider configuration named `ssl-client` that is defined in the `<socket-providers>` element of the operational configuration file.

Example 3-24 Configuration for a remote cache scheme that references a socket provider

```
<?xml version="1.0"?>

<cache-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://xmlns.oracle.com/coherence/coherence-cache-config"
   xsi:schemaLocation="http://xmlns.oracle.com/coherence/coherence-cache-
config
   coherence-cache-config.xsd">
   <caching-scheme-mapping>
      <cache-mapping>
         <cache-name>extend-*</cache-name>
         <scheme-name>remote-cache</scheme-name>
      </cache-mapping>

      <cache-mapping>
         <cache-name>grpc-*</cache-name>
         <scheme-name>grpc-cache</scheme-name>
      </cache-mapping>
   </caching-scheme-mapping>

   <caching-schemes>
      <remote-cache-scheme>
         <scheme-name>remote-cache</scheme-name>
         <service-name>RemoteCache</service-name>
         <initiator-config>
            <tcp-initiator>
               <socket-provider>ssl-client</socket-provider>
            </tcp-initiator>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
        </initiator-config>
     </remote-cache-scheme>

      <remote-grpc-cache-scheme>
         <scheme-name>grpc-cache</scheme-name>
         <service-name>RemoteGrpcCache</service-name>
        <grpc-channel>
            <socket-provider>ssl-client</socket-provider>
         </grpc-channel>
     </remote-grpc-cache-scheme>
   </caching-schemes>
</cache-config>
```

Extend or gRPC Client with mTLS

A client can be configured for mTLS using a `<ssl>` socket provider configured with both identity and trust.

Example 3-25 shows a socket provider configured for mTLS. If the cluster side proxy is configured to use mTLS the client certificate from the identity will be sent to the server. In both two-way and one-way SSL, the CA certificates in the trust store will be used to verify the proxy server identity.

Example 3-25 Configuration for an Extend or gRPC client with mTLS

```
<socket-provider id="mtls-config">
  <ssl>
     <identity-manager>
        <key-store>
            <url>file:server.jks</url>
        </key-store>
     </identity-manager>
      <trust-manager>
        <key-store>
            <url>file:trust.jks</url>
        </key-store>
      </trust-manager>
  </ssl>
</socket-provider>
```

Extend or gRPC Client with One-Way SSL

A client can be configured for one-way SSL using a `<ssl>` socket provider configured with only a trust store. The proxy server must also be configured for one-way SSL.

Example 3-26 shows a socket provider configured for one-way SSL. The CA certificates in the trust store will be used to verify the identity of the server certificate.

Example 3-26 Configuration for an Extend or gRPC client with one-way SSL/TLS

```
<socket-provider id="one-way-config">
  <ssl>
     <trust-manager>
        <key-store>
            <url>file:trust.jks</url>
        </key-store>
      </trust-manager>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
  </ssl>
</socket-provider>
```

Configure a Default Socket Provider for a Cache Configuration File

You can configure a socket provider in the `<defaults>` section of the cache configuration file. This socket provider will then apply to all the schemes in that configuration file that do not specifically configure their own socket provider, such as any remote cache services, remote invocation services, proxy services, gRPC services, and so on.

In Example 3-27, the `<defaults>` section of the cache configuration file has a socket provider that references a provider named `mtls` from the operational configuration file. The cache configuration file contains three remote schemes, none of which have a socket provider configured so they will all use the `mtls` socket provider.

Example 3-27 Configuration for referencing sockets providers in the `<defaults>` section of cache configuration

```
<?xml version="1.0"?>
<cache-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://xmlns.oracle.com/coherence/coherence-cache-config"
        xsi:schemaLocation="http://xmlns.oracle.com/coherence/coherence-cache-
config
            coherence-cache-config.xsd">

    <defaults>
        <socket-provider>mtls</socket-provider>
    </defaults>

    <caching-scheme-mapping>
        <cache-mapping>
            <cache-name>*</cache-name>
            <scheme-name>remote</scheme-name>
        </cache-mapping>

        <cache-mapping>
            <cache-name>grpc-cache</cache-name>
            <scheme-name>remote-grpc</scheme-name>
        </cache-mapping>
    </caching-scheme-mapping>

    <caching-schemes>
        <remote-cache-scheme>
            <scheme-name>remote</scheme-name>
            <service-name>RemoteService</service-name>
        </remote-cache-scheme>

        <remote-invocation-scheme>
            <scheme-name>invocation</scheme-name>
            <service-name>RemoteInvocation</service-name>
        </remote-invocation-scheme>

        <remote-grpc-cache-scheme>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
            <scheme-name>remote-grpc</scheme-name>
            <service-name>RemoteGrpcCache</service-name>
        </remote-grpc-cache-scheme>
    </caching-schemes>
</cache-config>
```

In Example 3-28, the `<defaults>` section of the cache configuration file has a socket provider that references a provider named `mtls` from the operational configuration file. The cache configuration file contains three remote schemes: a remote cache scheme, a remote invocation scheme, and a remote gRPC cache scheme. The remote cache scheme and the remote invocation scheme do not specify a socket provider so they use the `mtls` socket provider. However, the remote gRPC scheme *does* have a socket provider configured and so it references a socket provider named `one-way` in the operational configuration file.

Example 3-28

```
<?xml version="1.0"?>
<cache-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://xmlns.oracle.com/coherence/coherence-cache-config"
    xsi:schemaLocation="http://xmlns.oracle.com/coherence/coherence-cache-
config
        coherence-cache-config.xsd"
    xml-override="{coherence.cacheconfig.override}">

    <defaults>
        <socket-provider>mtls-config</socket-provider>
    </defaults>

    <caching-scheme-mapping>
        <cache-mapping>
            <cache-name>*</cache-name>
            <scheme-name>remote</scheme-name>
        </cache-mapping>

        <cache-mapping>
            <cache-name>grpc-cache</cache-name>
            <scheme-name>remote-grpc</scheme-name>
        </cache-mapping>
    </caching-scheme-mapping>

    <caching-schemes>
        <remote-cache-scheme>
            <scheme-name>remote</scheme-name>
            <service-name>RemoteService</service-name>
        </remote-cache-scheme>

        <remote-invocation-scheme>
            <scheme-name>invocation</scheme-name>
            <service-name>RemoteInvocation</service-name>
        </remote-invocation-scheme>

        <remote-grpc-cache-scheme>
            <scheme-name>remote-grpc</scheme-name>
            <service-name>RemoteGrpcCache</service-name>
            <grpc-channel>
                <socket-provider>one-way</socket-provider>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
            </grpc-channel>
         </remote-grpc-cache-scheme>
      </caching-schemes>
</cache-config>
```

Configuring a .NET Client-Side Stream Provider

Configure SSL in the .NET client-side cache configuration file by defining an SSL stream provider for remote services. The SSL stream provider is defined using the `<stream-provider>` element within the `<tcp-initiator>` element.

> **Note**
>
> Certificates are managed on Window servers at the operating system level using the Certificate Manager. The sample configuration assumes that the Certificate Manager includes the extend proxy's certificate and the trusted CA's certificate that signed the proxy's certificate.

Example 3-29 demonstrates a remote cache scheme that configures an SSL stream provider. Refer to the cache configuration XML schema (*`INSTALL_DIR`*`\config\cache-config.xsd`) for details on the elements that are used to configure an SSL stream provider.

> **Note**
>
> The `<protocol>` element support any allowed `SslProtocols` enumeration values as well as a comma separated list of protocol values. For example:
>
> ```
> <protocol>Tls11,Tls12</protocol>
> ```
>
> Ensure the protocol is specified in both the client-side and server-side configuration.

Example 3-29 Sample .NET Client-Side SSL Configuration

```
<?xml version="1.0"?>

<cache-config xmlns="http://schemas.tangosol.com/cache"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://schemas.tangosol.com/cache
   assembly://Coherence/Tangosol.Config/cache-config.xsd">
   <caching-scheme-mapping>
      <cache-mapping>
         <cache-name>dist-extend</cache-name>
         <scheme-name>extend-dist</scheme-name>
      </cache-mapping>
   </caching-scheme-mapping>

   <caching-schemes>
      <remote-cache-scheme>
         <scheme-name>extend-dist</scheme-name>
         <service-name>ExtendTcpSSLCacheService</service-name>
         <initiator-config>
            <tcp-initiator>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
                  <stream-provider>
                     <ssl>
                        <protocol>Tls12</protocol>
                        <local-certificates>
                           <certificate>
                              <url>C:\</url>
                              <password>password</password>
                              <flags>DefaultKeySet</flags>
                           </certificate>
                        </local-certificates>
                     </ssl>
                  </stream-provider>
                  <remote-addresses>
                     <socket-address>
                        <address>198.168.1.5</address>
                        <port>9099</port>
                     </socket-address>
                  </remote-addresses>
                  <connect-timeout>10s</connect-timeout>
               </tcp-initiator>
               <outgoing-message-handler>
                  <request-timeout>5s</request-timeout>
               </outgoing-message-handler>
            </initiator-config>
         </remote-cache-scheme>
      </caching-schemes>
</cache-config>
```

Securing the C++ Client with SSL/TLS

The Coherence C++ Extend Client does not officially support SSL/TLS. However, you can use one of the following options to work around this limitation to run C++ extend clients securely against an SSL/TLS enabled Coherence proxy server.

Secure the C++ client using a Load Balancer

You can configure a load balancer such as F5 to perform encryption on behalf of your C++ client and communicate with the SSL/TLS proxy servers behind the load balancer. Refer to the documentation for the load balancer service for information on how to configure it to provide data protection.

Secure C++ Client using SSH Tunneling

When SSH tunneling is enabled, the C++ client connects to a port on the local host that the SSH client listens on. The SSH client then forwards the requests over its encrypted tunnel to the server. The server connects to the SSL/TLS enabled Coherence proxy server - usually on the same machine or in the same data center as the SSH server. You can easily find examples on how to configure an SSH tunnel. Coherence proxy servers are behind the SSH server.

Secure C++ Client in a Cloud

If you are in a cloud environment, such as Oracle Cloud Infrastructure Container Engine for Kubernetes (OKE), you can configure an NGINX container on the same pod as the C++ client to serve as an SSL/TLS proxy to communicate with the Coherence SSL/TLS proxy server. You can also use an Istio sidecar proxy or egress gateway to perform encryption on behalf of the C++ client. Refer to the corresponding documentation for your cloud environment for instructions on how to secure data to the upstream servers.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Using SSL to Secure Federation Communication

Oracle Coherence supports using SSL to secure communication between cluster participants in a federated cluster. Communication is secured between federated service members and requires SSL to be configured on each cluster participant.

To use SSL to secure federation communication, you can configure the `<socket-provider>` element in the `<federated-scheme>`.

The socket provider for federation is similar to the one used for the clusters unicast sockets, that is, the socket provider configures both server and client sockets. This restricts the types of configuration that are supported.

Example 3-30 shows a `<federated-scheme>` with a `<socket-provider>` configured within the scheme definition.

Example 3-30 Configuration for using an inline `<socket-provider>` to configure SSL/TLS between cluster participants in a federated cluster

```
<federated-scheme>
   <scheme-name>federated</scheme-name>
   <service-name>federated</service-name>
   <backing-map-scheme>
      <local-scheme />
   </backing-map-scheme>
   <autostart>true</autostart>
   <socket-provider>
     <ssl>
         <identity-manager>
            <key-store>
                <url>file:server.jks</url>
            </key-store>
         </identity-manager>
          <trust-manager>
            <key-store>
                <url>file:trust.jks</url>
            </key-store>
          </trust-manager>
      </ssl>
   </socket-provider>
  <topologies>
      <topology>
         <name>MyTopology</name>
      </topology>
   </topologies>
</federated-scheme>
```

Example 3-31 shows a `<federated-scheme>` with a `<socket-provider>` that references a socket provider named `ssl-federation` (which has been configured in the `<socket-providers>` section of the operational configuration file).

Copyright © 2008, 2026, Oracle and/or its affiliates.

Example 3-31 Configuration for using a referenced `<socket-provider>` to configure SSL/TLS between cluster participants in a federated cluster

```
<federated-scheme>
   <scheme-name>federated</scheme-name>
   <service-name>federated</service-name>
   <backing-map-scheme>
      <local-scheme />
   </backing-map-scheme>
   <autostart>true</autostart>
   <socket-provider>ssl-federation</socket-provider>
  <topologies>
      <topology>
         <name>MyTopology</name>
      </topology>
   </topologies>
</federated-scheme>
```

- Federation with mTLS
  The most common and recommended configuration for using SSL to secure federation is mTLS. To configure an `<ssl>` socket provider for mTLS, both identity and trust must be configured.
- Federation with One-Way SSL
  You can configure one-way SSL for federation participants so that a participant will verify trust of a server certificate that it receives when making a connection to another participants. A federation participant will not verify trust for a member that connects to it.

Federation with mTLS

The most common and recommended configuration for using SSL to secure federation is mTLS. To configure an `<ssl>` socket provider for mTLS, both identity and trust must be configured.

In Example 3-32, the socket provider is configured with an `<identity-manager>` element that contains a keystore named `server.jks`, and a `<trust-manager>` element that contains a keystore named `trust.jks`. By default, when a `<ssl>` socket provider is configured with both identity and trust, Coherence will create a Java SSL context that is configured for mTLS.

> **Note**
>
> When using certificate with extended usage set in this configuration, the extended usage must include both `serverAuth` and `clientAuth`. The socket provider configuration is used to configure a single Java SSL Context used by both the federated scheme server sockets (used to receive connections from other federations participants) and its client sockets (used to connect to other federations participants).

Example 3-32 Configuration for securing federated cluster communication over mTLS

```
<socket-provider>
   <ssl>
      <identity-manager>
         <key-store>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
            <url>file:server.jks</url>
          </key-store>
        </identity-manager>
         <trust-manager>
          <key-store>
            <url>file:trust.jks</url>
          </key-store>
         </trust-manager>
   </ssl>
</socket-provider>
```

Federation with One-Way SSL

You can configure one-way SSL for federation participants so that a participant will verify trust of a server certificate that it receives when making a connection to another participants. A federation participant will not verify trust for a member that connects to it.

Only a single socket provider can be configured for the federated scheme, therefore, the `<ssl>` socket provider must be configured with both identity and trust, the same as with the mTLS example. To specify one-way SSL, add a `<client-auth>`element and set its value to `none`.

The `<client-auth>` element is used to configure the corresponding setting in the Java SSL context which determines whether the client must send a certificate. When set to none the client does not send a certificate (even if configured with an identity key and certificate).

Example 3-33 shows the federated scheme socket provider configuration for one-way SSL.

> **Note**
>
> When using certificate with extended usage set in this configuration, the extended usage must include `serverAuth`. In one-way SSL, only the server sends a certificate to the client, so the certificate must be valid for use as a server certificate.

Example 3-33 Configuration for securing federated cluster communication over one-way SSL/TLS

```
<socket-provider>
   <ssl>
      <identity-manager>
         <key-store>
            <url>file:server.jks</url>
         </key-store>
      </identity-manager>
       <trust-manager>
         <key-store>
            <url>file:trust.jks</url>
         </key-store>
       </trust-manager>
<!-- This element changes the configuration to one-way SSL -->
       <client-auth>none</client-auth>
   </ssl>
 </socket-provider>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

Coherence PeerX509 Algorithm

Oracle Coherence includes a proprietary peer trust algorithm, PeerX509, which works by assuming trust (and only trust) of the certificates that are in the trust manager keystore. It also leverages the peer-to-peer protocol features of TCMP. Specifically, for the SSL negotiation to succeed, the certificate received must be the same as one of the certificates held by the trust manager .

You can configure PeerX509 by setting the `<algorithm>` element in the `<trust-manager>` element.

In Example 3-34, the trust manager uses the PeerX509 algorithm. Both the identity manager and the trust manager are configured to use the same keystore. This is a common approach for PeerX509 when used for to secure cluster membership, because all cluster members use the same configuration, and the certificate sent by the client or server socket is guaranteed to be in the trust store.

> **Note**
>
> - PeerX509 is a Coherence proprietary algorithm and is not compliant with standards such as Federal Information Processing Standards (FIPS). It may not be usable in highly restricted environments.
> - Trust is verified if the certificate received matches one of those in the trust store. There is no checking of additional certificate data such as extended usages or if the certificate is signed.

Example 3-34 Configuration where the trust manager uses the PeerX509 algorithm

```
<socket-provider>
    <ssl>
        <identity-manager>
            <key-store>
                <url>file:server.jks</url>
            </key-store>
        </identity-manager>
        <trust-manager>
            <algorithm>PeerX509</algorithm>
            <key-store>
                <url>file:server.jks</url>
            </key-store>
        </trust-manager>
    </ssl>
</socket-provider>
```

Specifying a Global Socket Provider

You can configure a global socket provider in the Coherence operational configuration file. When set, every server or client socket that Coherence creates will use this configuration unless it has been overridden with a specific socket provider of its own.

Copyright © 2008, 2026, Oracle and/or its affiliates.

In the Coherence operational configuration file, within the `<cluster-config>` element, specify a `<global-socket-provider>` element.

In Example 3-35, the operational configuration file configures a socket provider named `mtls-config` that is configured for mTLS. The `<global-socket-provider>` element is then set to `mtls-config` so that this socket provider will be used everywhere.

Example 3-35 Configuration for specifying a global socket provider

```
<?xml version='1.0'?>

<coherence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://xmlns.oracle.com/coherence/coherence-operational-config"
   xsi:schemaLocation="http://xmlns.oracle.com/coherence/coherence-
operational-config
   coherence-operational-config.xsd">
   <cluster-config>

      <global-socket-provider>mtls-config</global-socket-provider>

      <socket-providers>
         <socket-provider id="mtls-config">
           <ssl>
              <identity-manager>
                 <key-store>
                     <url>file:server.jks</url>
                 </key-store>
              </identity-manager>
               <trust-manager>
                 <key-store>
                     <url>file:trust.jks</url>
                 </key-store>
               </trust-manager>
           </ssl>
         </socket-provider>
      </socket-providers>
   </cluster-config>
</coherence>
```

The default operational configuration file allows the global socket provider to be set using the system property `coherence.global.socketprovider` (or the environment variable `COHERENCE_GLOBAL_SOCKET_PROVIDER`).

In Example 3-36, the operational configuration configures a socket provider named `mtls-config` for two-way SSL, but it does not set the global socket provider element.

Example 3-36 Configuration for a socket provider *without* a global socket provider

```
<?xml version='1.0'?>

<coherence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://xmlns.oracle.com/coherence/coherence-operational-config"
   xsi:schemaLocation="http://xmlns.oracle.com/coherence/coherence-
operational-config
   coherence-operational-config.xsd">
   <cluster-config>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
      <socket-providers>
         <socket-provider id="mtls-config">
           <ssl>
              <identity-manager>
                 <key-store>
                     <url>file:server.jks</url>
                 </key-store>
              </identity-manager>
               <trust-manager>
                 <key-store>
                     <url>file:trust.jks</url>
                 </key-store>
               </trust-manager>
           </ssl>
         </socket-provider>
      </socket-providers>
   </cluster-config>
</coherence>
```

If Coherence is now started with the system property `-Dcoherence.global.socketprovider=mtls-config`, then the `mtls-config` socket provider will be used as the global socket provider.

> **Note**
>
> The global socket provider will be used for *all* server and client sockets. Therefore, the configured socket provider must be capable of being used for both server and client sockets. Typically, this means you should configure the global socket provider for mTLS, or for one-way SSL as described in Cluster Communication with One-Way SSL.

Specifying Passwords in Socket Provider Configuration

Java keystores and private keys can be secured with credentials, typically a password. The socket provider configuration provides several ways to specify a password. It is up to the application developer to choose the most suitable approach based on the required level of security versus simplicity of configuration.

- Specify Plain Text Passwords
- Passwords From Java System Properties
  You can use the Coherence configuration system property replacement feature to specify a password using a system property. The `<password>` element has an optional `system-property` attribute that specifies which Java system property to use to obtain the value for the XML element.
- Reading Passwords From a URL
  You can load a password from a URL, such as a file on the file system using the `<password-url` element.
- Custom Password Providers

Copyright © 2008, 2026, Oracle and/or its affiliates.

Specify Plain Text Passwords

Recognize the literal password configuration form and prefer controlled password-provider sources.
The `<password>` element is a literal password source in supported SSL/keystore configuration. It can be useful when recognizing an existing configuration, but embedding production secrets in XML or JVM command lines exposes them to configuration readers, process inspection, backups, and accidental logs.

Use Configuring TLS with Protected Password Sources for a deployable provider-based example. Limit filesystem and credential-service access, keep store/key password locations distinct, and coordinate secret rotation with the consuming loader or channel lifecycle.

If a deployment deliberately retains literal-password compatibility, protect the entire configuration and its operational handling as a secret. Do not treat obfuscation or a property substitution as encryption or access control.

Passwords From Java System Properties

You can use the Coherence configuration system property replacement feature to specify a password using a system property. The `<password>` element has an optional `system-property` attribute that specifies which Java system property to use to obtain the value for the XML element.

In Example 3-37, the `<password>` element is configured to read an encrypted private key. Its `system-property` attribute is set to `key.credentials`, so at runtime, when the XML configuration is parsed, the value of the `<password>` system property will be used as the password.

Using system properties is more flexible than plain text, hardcoded passwords, but it is not particularly secure. The passwords will be injected into the XML after it is loaded and stored in the Coherence configuration classes in memory in plain text.

Example 3-37 Configuration for specifying a password using Java system properties

```
<identity-manager>
   <key>server.key</key>
   <cert>server.cert</cert>
   <password system-property="key.credentials"/>
</identity-manager>
```

Reading Passwords From a URL

You can load a password from a URL, such as a file on the file system using the `<password-url` element.

Example 3-38 shows an SSL socket provider configuration that reads the keystore and private key passwords from files on the file system.

- The identity manager's keystore password is read from `/coherence/security/server-pass.txtfile`.
- The private key used by the identity manager is read from `/coherence/security/key-pass.txtfile`.

Copyright © 2008, 2026, Oracle and/or its affiliates.

- The keystore password used by the trust manager is read from the `/coherence/security/trust-pass.txtfile`.

Although Example 3-38 uses files, you can use any valid URL that is readable, for example, a simple HTTP URL to get the password from a web server.

Example 3-38 Configuration for retrieving passwords from a URL

```
<socket-provider>
   <ssl>
      <identity-manager>
         <key-store>
            <url>file:server.jks</url>
            <password-url>
              file:/coherence/security/server-pass.txt
            </password-url>
         </key-store>
         <password-url>
           file:/coherence/security/key-pass.txt
         </password-url>
      </identity-manager>
      <trust-manager>
         <key-store>
            <url>file:trust.jks</url>
            <password-url>
              file:/coherence/security/trust-pass.txt
            </password-url>
         </key-store>
      </trust-manager>
   </ssl>
</socket-provider>
```

Custom Password Providers

Obtain passwords from a controlled source and make ownership, refresh, and failure behavior explicit.
A password provider implements `com.tangosol.net.PasswordProvider` and returns a password as `char[]`. Use the built-in `URLPasswordProvider` for a protected file source, or a reviewed adapter for the approved credential system. Configuring TLS with Protected Password Sources provides complete named-provider configuration.

Custom provider contract

Return an owned character array suitable for the caller's use and cleanup. Do not share a mutable singleton array between concurrent operations. Define what missing, expired, or unavailable material means; a source failure should not silently yield a hardcoded fallback password. Bound remote retrieval and avoid logging the password or credential response.

Coherence supports constructor/factory configuration through the normal instance syntax. Named references use `<password-provider><name>provider-name</name></password-provider>`. Put reusable definitions in the operational `<cluster-config><password-providers>` section.

When application code consumes a provider directly, clear its owned copy in a `finally` block using `PasswordProvider.reset(...)`. Immutable strings, encoded authorization headers,

Copyright © 2008, 2026, Oracle and/or its affiliates.

external library buffers, and provider-side caches have separate lifetimes; clearing one array is not a guarantee that every copy has been erased.

Coordinate credential rotation with refreshable TLS loaders or channel/application lifecycle. Document provider-specific behavior and test failed refresh and restoration. Keep secret permissions and credential-service identity restrictions in place even when the provider encrypts data at rest.

Controlling Cipher Suite and Protocol Version Usage

An SSL socket provider can be configured to control the use of potentially weak ciphers or specific protocol versions. To control cipher suite and protocol version usage, edit the SSL socket provider definition and include the `<cipher-suites>` element and the `<protocol-versions>` elements, respectively, and enter a list of cipher suites and protocol versions using the `name` element. Include the `usage` attribute to specify whether the cipher suites and protocol versions are allowed (value of `white-list`) or disallowed (value of `black-list`). The default value for the `usage` attribute if no value is specified is `white-list`.
For example:

```
<socket-provider>
   <ssl>
   ...
      <cipher-suites usage="black-list">
         <name>TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256</name>
      </cipher-suites>
      <protocol-versions usage="black-list">
         <name>SSLv3</name>
      </protocol-versions>
   ...
   </ssl>
</socket-provider>
```

Using Host Name Verification

Learn how to configure host name verification in Oracle Coherence. A host name verifier ensures that the host name in the URL to which the client connects matches the host name in the digital certificate that the server sends back as part of the SSL connection.
For standalone Coherence, verifier defaults depend on the security profile and socket-provider configuration. Explicit default verification rejects a mismatch in either profile; permissive compatibility behavior can warn where hardened policy rejects. WebLogic secured production mode has its own integration. See Verifying and Rotating TLS Identities.

A host name verifier is useful when an SSL client (for example, Coherence acting as an SSL client) connects to a cache server on a remote host. It helps to prevent man-in-the-middle attacks.

Coherence includes a default host name verifier, and provides the ability to create and use a custom host name verifier.

This section includes the following topics:

- Using the Default Coherence Host Name Verifier
  If you are using the default Coherence host name verifier, the host name verification passes if the host name in the certificate matches the host name to which the client tries to connect.

Copyright © 2008, 2026, Oracle and/or its affiliates.

- Using a Custom Host Name Verifier
  When using a custom host name verifier, the class that implements the custom host name verifier must be specified in the CLASSPATH of Coherence (when acting as an SSL client) or a standalone SSL client.

Using the Default Coherence Host Name Verifier

If you are using the default Coherence host name verifier, the host name verification passes if the host name in the certificate matches the host name to which the client tries to connect.

The default host name verifier verifies host name in two phases:

- Verification with wildcarding.
- Verification without wildcarding, if verification with wildcarding fails.

For standalone Coherence, verifier defaults depend on the security profile and socket-provider configuration. Explicit default verification rejects a mismatch in either profile; permissive compatibility behavior can warn where hardened policy rejects. WebLogic secured production mode has its own integration. See Verifying and Rotating TLS Identities.

Verification with Wildcarding

If the the host name in the server certificate of the SSL session supports wildcarding, the `CommonName` attribute must meet the following criteria:

- Have at least two dot ('.') characters.
- Must start with "*."
- Have only one "*" character.

In addition, the non-wildcarded portion of the CommonName attribute must equal the domain portion of the `urlhostname` parameter in a case-sensitive string comparison. The domain portion of `urlhostname` string is the `urlhostname` substring that remains after the `hostname` substring is removed. The `hostname` portion of `urlhostname` is the substring up to and excluding the first '.' (dot) of the `urlhostname` parameter string.

For example:

`urlhostname`: `mymachine.oracle.com`

`CommonName`: `*.oracle.com`

`.oracle.com` will compare successfully with `.oracle.com`.

`urlhostname`: `mymachine.uk.oracle.com`

`CommonName`: `*.oracle.com`

`.uk.oracle.com` will not compare successfully with `.oracle.com`

`DNSNames` obtained from the server certificate's SubjectAlternativeNames extension may be wildcarded.

Verification without Wildcarding

If wildcarded host name verification fails, the default host name verifier performs non-wildcarded verification. It verifies the `CommonName` attribute of the server certificate's `SubjectDN` or the `DNSNames` of the server certificate's `SubjectAlternativeNames` extension against the host name in the client URL (`urlhostname`). The `certificate` attribute must

Copyright © 2008, 2026, Oracle and/or its affiliates.

match the `urlhostname` (not case sensitive) parameter. The `SubjectDN CommonName` attribute is verified first, and if successful, the `SubjectAlternativeNames` attributes are not verified.

If the server certificate does not have a `SubjectDN`, or the `SubjectDN` does not have a `CommonName` attribute, then the `SubjectAlternativeName` attributes of type `DNSNames` are compared to the `urlhostname` parameter. The verification passes upon the first successful comparison to a `DNSName`. For a successful verification, the `urlhostname` must be equal to the `certificate` attribute being compared.

If `urlhostname` is `localhost`, you can set the `coherence.security.ssl.allowLocalhost` system property to `true` to enable 127.0.0.1, or the default IP address of the local machine to pass.

Using a Custom Host Name Verifier

When using a custom host name verifier, the class that implements the custom host name verifier must be specified in the CLASSPATH of Coherence (when acting as an SSL client) or a standalone SSL client.

For more information about using a custom host name verifier, see the description for the `<hostname-verifier>` element in ssl.

Configuring Client Authentication

You can use the `<client-auth>` element to specify whether a SSL/TLS socket provider should use one-way or two-way SSL/TLS authentication.

To apply `<client-auth>`, you must configure a socket provider with both an `<identity-manager>` and a `<trust-manager>` element in its XML configuration. If no `<trust-manager>` is configured, then only one-way authentication can be used. When a `<trust-manager>` is configured, Coherence will default to using two-way authentication.

`<client-auth>` is an enumeration with three valid values.

Value	Description
`none`	The socket provider does not request an authentication certificate from the client.
`wanted`	The socket provider requests an authentication certificate from the client, but the client is not required to send one. Corresponds to the `want client auth` setting in the Java SSL/TLS engine created by Coherence to manage SSL/TLS sockets.
`required`	The socket provider requires that the client send an authentication certificate. Corresponds to the `need client auth` setting in the Java SSL/TLS engine created by Coherence to manage SSL/TLS sockets.

Example 3-39 Sample One-Way SSL/TLS Authentication

The `<client-auth>` element is set to `none` so Coherence uses one-way SSL/TLS authentication, even though a trust manager has been configured.

Copyright © 2008, 2026, Oracle and/or its affiliates.

In this case, on the SSL/TLS Engine, both `want client auth` and `need client auth` would be set to `false` .

```
...
<cluster-config>
   <socket-providers>
      <socket-provider id="mySSLConfig">
         <ssl>
            <protocol>TLS</protocol>
            <identity-manager>
               <key-store>
                  <url>file:server.jks</url>
                  <password>password</password>
               </key-store>
               <password>password</password>
            </identity-manager>
            <trust-manager>
               <key-store>
                  <url>file:trust.jks</url>
                  <password>password</password>
               </key-store>
            </trust-manager>
            <client-auth>none</client-auth>
         </ssl>
      </socket-provider>
   </socket-providers>
</cluster-config>
```

Example 3-40 Sample Optional Client Auth

The `<client-auth>` element is set to `wanted` so the client may send a certificate but is not required to.

In this case, on the SSL/TLS Engine, `want client auth` would be set to `true` and `need client auth` would be set to `false` .

```
...
<cluster-config>
   <socket-providers>
      <socket-provider id="mySSLConfig">
         <ssl>
            <protocol>TLS</protocol>
            <identity-manager>
               <key-store>
                  <url>file:server.jks</url>
                  <password>password</password>
               </key-store>
               <password>password</password>
            </identity-manager>
            <trust-manager>
               <key-store>
                  <url>file:trust.jks</url>
                  <password>password</password>
               </key-store>
            </trust-manager>
            <client-auth>wanted</client-auth>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
        </ssl>
      </socket-provider>
    </socket-providers>
</cluster-config>
```

Using Private Key and Certificate Files

Coherence also supports using private key and certificate files directly, instead of loading them into a keystore. The examples in Specifying Passwords in Socket Provider Configuration used Java keystore files to store the private key and certificates used to establish trust and identity in Coherence SSL.

> **Note**
>
> Out of the box, Coherence only supports file formats supported by the JDK. These are private key files in PEM format (that is, a file with a header of `-----BEGIN RSA PRIVATE KEY-----` or `-----BEGIN ENCRYPTED PRIVATE KEY-----`) and X509 certificate files (that is, a file with a header of `-----BEGIN CERTIFICATE-----`).

- Configuring an Identity Manager
- Configuring a Trust Manager

Configuring an Identity Manager

When configuring an `<identity-manager>` element of a socket provider, instead of the `<keystore>` element, the `<key>` and `<cert>` elements can be used to supply the private key a certificate file locations. The value for both the `<key>` and `<cert>` element is a URL from which to load the key or certificate data.

Example 3-41 shows an `<identity-manager>` configuration that uses a private key loaded from the `/coherence/security/client.pem` file and a certificate loaded from the `/coherence/security/client.cert` file.

Example 3-41 Sample Identity Manager Using a Private Key and a Certificate File

```
<socket-provider>
  <ssl>
    <identity-manager>
      <key>file:/coherence/security/client.pem</key>
      <cert>file:/coherence/security/client.cert</cert>
    </identity-manager>
  </ssl>
</socket-provider>
```

When configuring an `<identity-manager>` element, the `<keystore>` element, and the `<key>` and `<cert>` elements are mutually exclusive; either configure a keystore, or a key and certificate. The Coherence operational configuration XSD validation does not allow both.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Configuring a Trust Manager

When configuring a `<trust-manager>` element of a socket provider, instead of the `<keystore>` element, one or more `<cert>` elements can be used to supply the certificate file locations. The value for the `<cert>` element is a URL from which to load the certificate data.

Example 3-42 shows a `<trust-manager>` configuration that uses a certificate loaded from the `/coherence/security/server-ca.cert` file.

Example 3-42 Sample Trust Manager Using a Certificate File

```
<socket-provider>
  <ssl>
    <trust-manager>
      <cert>file:/coherence/security/server-ca.cert</cert>
    </trust-manager>
  </ssl>
</socket-provider>
```

When configuring a `<trust-manager>` element, the `<keystore>` element and the `<cert>` elements are mutually exclusive; either configure a keystore, or one or more certificates. The Coherence operational configuration XSD validation does not allow both.

Using Custom Keystore, Private Key, and Certificate Loaders

To support loading keystores, private keys, and certificates from sources other than simple URLS or files, and to read different data formats, Coherence provides a way to configure custom loaders to read the required data from whatever external source is required. For example, in the cloud, keys and certificates can be stored in a secrets service and loaded directly from secrets, instead of loading from files. The Coherence OCI project on GitHub includes custom keystore, key, and certificate loaders that can read data from secrets in the Oracle Cloud (OCI) Secrets Service. See Coherence OCI.
This section includes the following topics:

- Using the Custom KeyStore Loader
- Using the Custom PrivateKey Loader
- Using a Custom Certificate Loader

Using the Custom KeyStore Loader

If using Java Keystores, you can implement an instance of `com.tangosol.net.ssl.KeyStoreLoader` in application code and configure it in the `<key-store-loader>` element, which is a child of the `<key-store>` element. This class can load the contents of a Java KeyStore from any desired location.

Example 3-43 shows a custom implementation of the `KeyStoreLoader` interface.

Example 3-43 A Custom KeyStore Loader Class

```
package com.acme.coherence;

import com.tangosol.net.ssl.KeyStoreLoader;
import java.io.IOException;
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
import java.security.GeneralSecurityException;
import java.security.KeyStore;

public class CustomKeyStoreLoader
        implements KeyStoreLoader
    {
    @Override
    public KeyStore load(String sType, PasswordProvider password)
            throws GeneralSecurityException, IOException
        {
        // return a KeyStore of the required type
        }
    }
```

Example 3-44 shows how you can use the `CustomKeyStoreLoader` class in a `<trust-manager>` configuration.

Example 3-44 Configure an Identity Manager with a Custom KeyStore Loader Class

```
<socket-provider>
  <ssl>
    <identity-manager>
      <key-store>
        <key-store-loader>
          <class-name>com.acme.coherence.CustomKeyStoreLoader</class-name>
        </key-store-loader>
      </key-store>
    </identity-manager>
  </ssl>
</socket-provider>
```

Example 3-45 shows how you can use the `CustomKeyStoreLoader` class in a `<trust-manager>` configuration.

Example 3-45 Configure a Trust Manager with a Custom KeyStore Loader Class

```
<socket-provider>
  <ssl>
    <trust-manager>
      <key-store>
        <key-store-loader>
          <class-name>com.acme.coherence.CustomKeyStoreLoader</class-name>
        </key-store-loader>
      </key-store>
    </trust-manager>
  </ssl>
</socket-provider>
```

As with other extension points in Coherence, the `<key-store-loader>` is an "instance" configuration that takes a `class-name` or a `class-factory-name` and `method-name` parameter. Optionally, the configuration can also use `<init-params>` to pass parameters to the class constructor or the factory method.

Example 3-46 shows how you can refactor the `CustomKeyStoreLoader` class to add constructor arguments.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Example 3-46 A Custom KeyStore Loader with Parameters

```
package com.acme.coherence;

import com.tangosol.net.ssl.KeyStoreLoader;
import java.io.IOException;
import java.security.GeneralSecurityException;
import java.security.KeyStore;

public class CustomKeyStoreLoader
        implements KeyStoreLoader
    {
    private final String param1;

    private final String param2;

    public CustomKeyStoreLoader(String param1, String param2)
        {
        this.param1 = param1;
        this.param2 = param2;
        }

    @Override
    public KeyStore load(String sType, PasswordProvider password)
            throws GeneralSecurityException, IOException
        {
        // return a KeyStore of the required type
        }
    }
```

Example 3-47 shows how you can configure the parameterized `CustomKeyStoreLoader` class. With the example configuration, the `CustomKeyStoreLoader` constructor is called with the parameters `foo` and `bar`.

Example 3-47 Configure a Custom KeyStore Loader with Parameters

```
<socket-provider>
  <ssl>
    <identity-manager>
      <key-store>
        <key-store-loader>
          <class-name>com.acme.coherence.CustomKeyStoreLoader</class-name>
          <init-params>
            <init-param>
              <param-type>string</param-type>
              <param-value>foo</param-value>
            </init-param>
            <init-param>
              <param-type>string</param-type>
              <param-value>bar</param-value>
            </init-param>
          </init-params>
        </key-store-loader>
      </key-store>
    </identity-manager>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
  </ssl>
</socket-provider>
```

At runtime, the `CustomKeyStoreLoader` class's `load` method is called to load the keystore. In the configurations above, the `type` parameter passed to the `load` method is the default keystore type ("JKS"). The `PasswordProvider` passed to the `load` method is the default null implementation that returns an empty password.

Example 3-48 shows how you can configure the keystore type and password, which are passed as parameters to the custom `KeyStoreLoader.load`. The example shows using the `<password>` element, but you can also use the `<password-url>` or the `<password-provider>` elements to supply the password to the loader.

Example 3-48 Passing the Keystore Type and Password to a Custom KeyStore Loader

```
<socket-provider>
  <ssl>
    <identity-manager>
      <key-store>
        <key-store-loader>
          <class-name>com.acme.coherence.CustomKeyStoreLoader</class-name>
        </key-store-loader>
        <password>secret</password>
        <type>PKCS12</type>
      </key-store>
    </identity-manager>
  </ssl>
</socket-provider>
```

Using the Custom PrivateKey Loader

If using private keys instead of keystores, you can implement an instance of `com.tangosol.net.ssl.PrivateKeyLoader` in application code and configure it in the `<key-loader>` element. The custom loader can then load a `PrivateKey` from any desired location in any required format.

As with other extension points in Coherence, the `<key-loader>` is an "instance" configuration that takes a `class-name` or a `class-factory-name` and a `method-name` parameter. Optionally, the configuration can also use `<init-params>` to pass parameters to the class constructor or factory method.

Example 3-49 shows a custom PrivateKeyLoader class.

Example 3-49 A Custom Private Key Loader

```
package com.acme.coherence;

import com.tangosol.net.PasswordProvider;
import com.tangosol.net.ssl.PrivateKeyLoader;
import java.io.IOException;
import java.security.GeneralSecurityException;
import java.security.KeyStore;

public class CustomPrivateKeyLoader
        implements PrivateKeyLoader
    {
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
    @Override
    public PrivateKey load(PasswordProvider password)
            throws GeneralSecurityException, IOException
        {
        // return a PrivateKey (optionally encrypted with a password)
        }
    }
```

Example 3-50 shows how you can configure the `CustomPrivateKeyLoader` class in the `<identity-manager>` element.

Example 3-50 Configure a Custom Private Key Loader

```
<socket-provider>
  <ssl>
    <identity-manager>
        <key-loader>
           <class-name>com.acme.coherence.CustomPrivateKeyLoader</class-name>
        </key-loader>
    </identity-manager>
  </ssl>
</socket-provider>
```

At runtime, the `CustomPrivateKeyLoader` class's `load` method is called to create the `PrivateKey` instance. In the example above, there is no password configured for the key, so the `PasswordProvider` that is passed to the `load` method returns an empty password (`new char[0]`). You can add a password by using one of the password elements allowed in the `<identity-manager>` elements.

Example 3-51 shows a configuration with a password. In this example, the `PasswordProvider` returns the contents fetched from the URL `file: /coherence/security/key-pass.txt` as the key password.

Example 3-51 Configure a Password for a Custom Private Key Loader

```
<socket-provider>
  <ssl>
    <identity-manager>
      <key-loader>
        <class-name>com.acme.coherence.CustomPrivateKeyLoader</class-name>
      </key-loader>
      <password-url>file:/coherence/security/key-pass.txt</password-url>
    </identity-manager>
  </ssl>
</socket-provider>
```

Using a Custom Certificate Loader

If using certificate files in the identity manager or trust manager, you can implement an instance of `com.tangosol.net.ssl.CertificateLoader` in application code and configure it in the `<cert-loader>` element. This class can load an array of `Certificate` instances from any desired location in the required format.

As with other extension points in Coherence, the `<cert-loader>` is an "instance" configuration that takes a `class-name` or a `class-factory-name` and a `method-name` parameter. Optionally,

Copyright © 2008, 2026, Oracle and/or its affiliates.

the configuration can also use `<init-params>` to pass parameters to the `class` constructor or the `factory` method.

Example 3-52 shows an example of a custom `CertificateLoader` class. The `load` method is called to load the certificates.

Example 3-52 A Custom Certificate Loader

```
package com.acme.coherence;

public class CustomCertificateLoader
        implements CertificateLoader
    {
    @Override
    public Certificate[] load()
            throws GeneralSecurityException, IOException
        {
        // return a Certificate array
        }
    }
```

Example 3-53 shows how you can configure the `CustomCertificateLoader` class in the identity manager.

Example 3-53 Configure a Custom Certificate Loader in an Identity Manager

```
<socket-provider>
  <ssl>
    <identity-manager>
      <key>server.pem</key>
      <cert-loader>
        <class-name>com.acme.coherence.CustomCertificateLoader</class-name>
      </cert-loader>
    </identity-manager>
  </ssl>
</socket-provider>
```

Example 3-54 shows how you can configure the `CustomCertificateLoader` class in the trust manager.

Example 3-54 Configure a Custom Certificate Loader in an Trust Manager

```
<socket-provider>
  <ssl>
    <trust-manager>
      <cert-loader>
        <class-name>com.acme.coherence.CustomCertificateLoader</class-name>
      </cert-loader>
    </trust-manager>
  </ssl>
</socket-provider>
```

The `load()` method of the `CertificateLoader` class returns an array of certificates; so it can load multiple certificates. You can also configure multiple `<cert-loader>` elements to use

Copyright © 2008, 2026, Oracle and/or its affiliates.

multiple custom loaders. All the certificates provided by all the `<cert>` or `<cert-loader>` elements are combined into a single set of certificates for the SSL context to use.

Example 3-55 shows how you can configure multiple `<cert>` and custom loaders in a trust manager.

Example 3-55 Configure Multiple Certificates and Loaders in a Trust Manager

```
<socket-provider>
  <ssl>
    <trust-manager>
      <cert>server-ca.cert</cert>
      <cert-loader>
        <class-name>com.acme.coherence.CustomCertificateLoader</class-name>
        <init-params>
          <init-param>
            <param-type>string</param-type>
            <param-value>foo</param-value>
          </init-param>
        </init-params>
      </cert-loader>
      <cert-loader>
        <class-name>com.acme.coherence.CustomCertificateLoader</class-name>
        <init-params>
          <init-param>
            <param-type>string</param-type>
            <param-value>bar</param-value>
          </init-param>
        </init-params>
      </cert-loader>
    </trust-manager>
  </ssl>
</socket-provider>
```

Using Refreshable KeyStores, Private Keys, and Certificates

In some environments, keys, and certs used for TLS are created with relatively short lifetimes. This means that a Coherence application should be able to renew the keys and certs, ideally without having to restart the JVM. In Coherence releases prior to 14.1.1.2206, this feature was not available because a keystore was loaded once when the socket provider was instantiated. From Coherence release 14.1.1.2206 onward, it is possible to specify a refresh period, which then schedules a refresh of the SSL context to reload any configured keystores, private keys, and certificates.
The `<refresh-period>` element is used to configure the refresh time. This is a child element of the `ssl` element, meaning that the setting applies to both the identity manager and trust manager.

Example 3-56 configures a `<refresh-period>` element with a value of `24h` to refresh the keys and certs every 24 hours.

Example 3-56 Configure a Refresh Period

```
<socket-provider>
  <ssl>
    <identity-manager>
      <key>server.pem</key>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
      <cert>server.cert</cert>
    </identity-manager>
    <refresh-period>24h</refresh-period>
  </ssl>
</socket-provider>
```

Refreshable keystores, keys, and certs can easily be combined with custom keystore loaders, private key loaders, and certificate loaders, so that the new versions of the required SSL artifacts can be pulled from an external source.

- Configuring a Refresh Policy

Configuring a Refresh Policy

When using refreshable keys and certs, it may sometimes be useful to have an additional check to determine whether a refresh should occur. You can perform this check by configuring a `<refresh-policy>` as well as a `<refresh-period>`.

The `<refresh-policy>` element is a standard Coherence instance configuration and should resolve to an instance of `com.tangosol.net.ssl.RefreshPolicy`. When a scheduled refresh time is reached, the policy is checked first (by calling the `RefreshPolicy.shouldRefresh()` method) to determine whether the refresh should go ahead.

Example 3-57 shows a custom `RefreshPolicy` implementation.

Example 3-57 A Custom Refresh Policy Class

```
package com.acme.coherence;

public class CustomRefreshPolicy
        implements RefreshPolicy
    {
    @Override
    public boolean shouldRefresh(Dependencies deps, ManagerDependencies
depsIdMgr, ManagerDependencies depsTrustMgr)
        {
        // perform some custom logic to determine whether it is time to
refresh
        return true;
        }
    }
```

Example 3-58 shows how you can configure the custom refresh policy as part of the `<ssl>` element alongside the `<refresh-period>`.

Example 3-58 Configure a Custom Refresh Policy

```
<socket-provider>
  <ssl>
    <identity-manager>
      <key>server.pem</key>
      <cert>server.cert</cert>
    </identity-manager>
    <refresh-period>24h</refresh-period>
    <refresh-policy>
      <class-name>com.acme.coherence.CustomRefreshPolicy</class-name>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
    </refresh-policy>
  </ssl>
</socket-provider>
```

For some policies, it is useful to know what keystores, keys, or certs are currently in use to determine whether they need to be refreshed. There are a number of default methods on `RefreshPolicy` that can be overridden for this purpose.

Example 3-59 shows how you can capture the certificates used by a trust store configuration, and then use the certificates to verify whether they are close to expiry. In this example, the `trustStoreLoaded` method is called when the trust store is created to notify the policy of the certificates used by the trust store. In the `shouldRefresh` method, the certificates can then be checked to determine whether they will still be valid at the next refresh interval.

Example 3-59 A Detailed Custom Certificate Refresh Policy

```
import com.oracle.coherence.common.net.SSLSocketProvider.Dependencies;
import com.oracle.coherence.common.util.Duration;
import
com.tangosol.coherence.config.builder.SSLSocketProviderDependenciesBuilder.Man
agerDependencies;
import com.tangosol.coherence.config.unit.Seconds;
import com.tangosol.net.ssl.RefreshPolicy;

import java.security.cert.Certificate;
import java.security.cert.CertificateExpiredException;
import java.security.cert.CertificateNotYetValidException;
import java.security.cert.X509Certificate;
import java.util.Date;

public class CustomRefreshPolicy
        implements RefreshPolicy
    {
    private Certificate[] certs;

    @Override
    public void trustStoreLoaded(Certificate[] certs)
        {
        this.certs = certs;
        }

    @Override
    public boolean shouldRefresh(Dependencies deps, ManagerDependencies
depsIdMgr, ManagerDependencies depsTrustMgr)
        {
        if (certs == null)
            {
            return true;
            }

        // get the refresh period from the dependencies
        Seconds secs = deps.getRefreshPeriod();
        // calculate the next refresh time as a Date
        Date nextRefresh = new Date(System.currentTimeMillis() +
secs.as(Duration.Magnitude.MILLI));
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
        for (Certificate certificate : certs)
            {
            try
                {
                // The certs are all X509 certs, so check their validity on
the next refresh date
                ((X509Certificate) certificate).checkValidity(nextRefresh);
                }
            catch (CertificateExpiredException |
CertificateNotYetValidException e)
                {
                // a cert will have expired, so we need to update now
                return true;
                }
            }

        // no certs should have expired at the next refresh check
        return false;
        }
    }
```

Example 3-60 shows how you can configure the `CustomRefreshPolicy` class in the `<ssl>` configuration.

Example 3-60 Configure the Custom Certificate Refresh Policy

```
<socket-provider>
  <ssl>
    <trust-manager>
      <ca-cert>server-ca.cert</ca-cert>
      <ca-cert>client-ca.cert</ca-cert>
    </trust-manager>
    <refresh-period>24h</refresh-period>
    <refresh-policy>
      <class-name>com.acme.coherence.CustomRefreshPolicy</class-name>
    </refresh-policy>
  </ssl>
</socket-provider>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

4

Using an Access Controller

Authenticate subjects and apply Coherence service permissions with an access controller appropriate to the deployment.
The Coherence security framework can authenticate a member/user subject and apply configured permissions to clustered services and caches. It is an opt-in control, separate from the security profile, TLS, and per-operation storage authorization. Retain the distinction between service access and permission to read or mutate individual cache data.

Use a suitable JAAS login module and configure the default or custom access controller. The default controller needs the expected principals and private/public signing material in the subject. An application-specific identity module must satisfy that contract if the default controller is used.

Authenticating Credentials with JAAS supplies a credential-assertion example and Using Authenticated Subjects with Modern Java shows subject context with current Java and Coherence APIs. The retained framework topics describe keystores, permissions, default/custom controllers, and audit logging.

Signing policy

In the hardened profile, which is the default starting with Coherence 26.1.0.0, the default controller selects `SHA256withRSA` and rejects weak signing algorithms. Explicit compatibility mode retains the historical `SHA1withDSA` default with a bounded warning. Migrate signing keys and explicit algorithm configuration together; an RSA algorithm cannot use incompatible DSA keys.

Signer/principal binding is enforced according to the current profile. Hardened authorization binds to the verified signer and rejects unsafe additional/unbound principals. Fresh per-operation signature state and rejection of an unverified certificate/principal mismatch apply in both profiles. Keep permission subjects consistent with the actual verified certificate identity.

Subject validation cache

`coherence.security.subject.validation.ttl` controls the Standard implementation's successful-validation cache. The default is `10s`; unitless values are milliseconds. Accepted durations range from one through `Integer.MAX_VALUE` milliseconds. Invalid, nonpositive, overflowing, or out-of-range values use the default with a warning. This setting is independent of both runtime mode and security profile and is read during initialization.

A longer TTL reduces repeated validation work but retains a successful decision longer. Select it with credential/key retirement and workload cost in mind. It does not make an unvalidated wire-created subject trustworthy or establish a revocation system by itself.

- Authenticating Credentials with JAAS
- Using Authenticated Subjects with Modern Java
- Overview of Using an Access Controller
  Coherence uses an access controller to secure access to cluster resources and operations. It uses a JAAS LoginModule to authenticate a caller, and an access controller on one or more cluster nodes verifies the access rights of the caller’s Subject.
- Proof of Identity

Copyright © 2008, 2026, Oracle and/or its affiliates.

- Configuring the JAAS LoginModule
  The LoginModule used by JAAS is configured in a JAAS configuration file. If using the Coherence security APIs to perform the JAAS login, Coherence uses a LoginModule configured with the name “Coherence”.
- Configuring a JAAS Callback Handler
  If Coherence needs to authorize permissions but the calling thread is not running inside the context of a Subject, then Coherence performs a JAAS login call. Depending on the specific LoginModule JAAS is configured to use, it may require an instance of a `javax.security.auth.callback.CallbackHandler` interface that can provide various parameters to the configured LoginModule.
- Using a Custom JAAS LoginModule
  Any custom JAAS LoginModule can be used that produces a Subject containing a principal and private/public keys. For example, you can use Java’s built-in `com.sun.security.auth.module.KeyStoreLoginModule` class.
- Enable the Access Controller
  Before enabling an access controller, you must configure the controller in the Coherence operational configuration file. The default configuration uses Coherence’s default access controller, but you can override it to use a custom controller implementation if desired.
- Using the Default Access Controller Implementation
  Coherence includes a default access controller implementation that uses a Java keystore for authentication. The implementation class is the `com.tangosol.net.security.DefaultController` class. The access controller is configured within the `<security-config>` element in the operational configuration file.
- Using a Custom Access Controller Implementation

Authenticating Credentials with JAAS

Convert validated username/password tokens into authenticated subjects using a configured JAAS login module.
Use a JAAS module that actually validates the presented credentials and returns the identity required by downstream authorization. The module may integrate an enterprise identity system through its supported implementation; configuring an extension point does not itself provide LDAP, Active Directory, or cloud IAM validation.

For the ordinary credential path, prefer the provided `com.tangosol.net.security.JAASIdentityAsserter`. Configure its constructor with the JAAS entry name. The custom adapter below is an optional extension example for applications needing additional subject/error validation.

```
<identity-asserter>
  <class-name>com.tangosol.net.security.JAASIdentityAsserter</class-name>
  <init-params>
    <init-param><param-type>java.lang.String</param-type><param-
value>Coherence</param-value></init-param>
  </init-params>
</identity-asserter>
```

The complete Java 21+ asserter below accepts `UsernameAndPassword`, used by explicit gRPC Basic authentication and by an appropriately configured Extend transformer. It does not accept

Copyright © 2008, 2026, Oracle and/or its affiliates.

a raw subject as proof. Configure a `Coherence` JAAS entry and include the login module on the runtime classpath.

```
package com.example.security;

import com.tangosol.net.Service;
import com.tangosol.net.security.IdentityAsserter;
import com.tangosol.net.security.SimpleHandler;
import com.tangosol.net.security.UsernameAndPassword;
import java.util.Arrays;
import javax.security.auth.Subject;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;

public final class JaasIdentityAsserter implements IdentityAsserter {
    @Override
    public Subject assertIdentity(Object token, Service service) {
        if (!(token instanceof UsernameAndPassword credentials)
                || credentials.getUsername() == null
                || credentials.getUsername().isBlank()
                || credentials.getPassword() == null
                || credentials.getPassword().length == 0) {
            throw new SecurityException("Authentication failed");
        }
        char[] password = credentials.getPassword().clone();
        try {
            var login = new LoginContext("Coherence",
                    new SimpleHandler(credentials.getUsername(), password,
false));
            login.login();
            var subject = login.getSubject();
            if (subject == null || subject.getPrincipals().isEmpty()) {
                throw new SecurityException("Authentication failed");
            }
            return subject;
        } catch (LoginException failure) {
            throw new SecurityException("Authentication failed");
        } finally {
            Arrays.fill(password, '\0');
        }
    }
}
```

The returned subject is used by the caller, so this minimal asserter does not immediately log it out and invalidate its credentials. A production module that owns refreshable credentials or session resources needs an appropriate lifecycle integration. Avoid caching unbounded subjects inside the asserter. Log bounded failure metadata without credentials or an externally visible detailed login cause.

For a keystore-based identity, the existing `coherence-login` module provides `com.tangosol.security.KeystoreLogin`. A JAAS entry can select an explicitly provisioned PKCS12 store:

```
Coherence {
  com.tangosol.security.KeystoreLogin required
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
    keyStorePath="/run/coherence/users.p12"
    keyStoreType="PKCS12";
};
```

Set `java.security.auth.login.config` to the protected JAAS configuration file. Provision aliases, certificate principals, and passwords according to the login module's callback contract. This is a keystore identity example, not a general role-directory implementation.

Place the following in the server operational override; merge it with the TLS and application settings, preserving schema order:

```
<security-config>
  <identity-asserter>
    <class-name>com.example.security.JaasIdentityAsserter</class-name>
  </identity-asserter>
  <subject-scope>true</subject-scope>
</security-config>
```

Enable the Coherence security framework separately when its access-controller permissions are required. For HTTP acceptors, configure their JAAS entry names as documented for that endpoint rather than assuming the gRPC/Extend asserter configures HTTP login. Verify a real successful login, missing/invalid credentials, the exact resulting principal, and a denied operation with valid credentials.

Using Authenticated Subjects with Modern Java

Bind a verified subject to application work using current Java subject-context and Coherence lifecycle APIs.
JAAS authentication remains useful on current JDKs. The Java Security Manager's former policy-based authorization model does not: the Security Manager is permanently disabled beginning with JDK 24. Coherence's service/storage access checks remain separate application controls.

The example below uses Java 21 or later, `Subject.callAs`, the Coherence bootstrap API, a typed `NamedMap`, and try-with-resources. It illustrates a member application using a configured login module and Coherence access-controller deployment. Start with an isolated configured cluster and the intended service identity; the example does not provision users, keys, roles, or permissions.

```
package com.example.security;

import com.tangosol.net.Coherence;
import com.tangosol.net.NamedMap;
import com.tangosol.net.security.SimpleHandler;
import java.util.Arrays;
import javax.security.auth.Subject;
import javax.security.auth.login.LoginContext;

public final class JaasSession {
    public static void main(String[] args) throws Exception {
        var console = System.console();
        if (console == null) {
            throw new IllegalStateException("An interactive console is
required");
        }
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
        var username = console.readLine("User: ");
        char[] password = console.readPassword("Password: ");
        if (password == null) {
            throw new IllegalStateException("No password was supplied");
        }
        var login = new LoginContext("Coherence",
                new SimpleHandler(username, password, false));
        boolean authenticated = false;
        try {
            login.login();
            authenticated = true;
            var subject = login.getSubject();
            Subject.callAs(subject, () -> {
                try (var coherence =
Coherence.clusterMember().start().join()) {
                    var session = coherence.getSession();
                    NamedMap<String, String> accounts =
session.getMap("accounts");
                    System.out.println(accounts.get("account-1"));
                }
                return null;
            });
        } finally {
            Arrays.fill(password, '\0');
            if (authenticated) {
                login.logout();
            }
        }
    }
}
```

Obtain and use subject-scoped handles within the intended subject context. Do not share a handle carrying one identity across unrelated users. Do not assume asynchronous callbacks or newly created execution contexts inherit an identity in the same way; use the supported context propagation and test the actual route.

An Extend client also needs a transformer producing the token accepted by its proxy. A locally authenticated subject passed through the default transformer is still a raw subject on the wire and is rejected by the hardened default asserter. Configure the explicit trusted token chain described in Securing Extend Client Connections.

Overview of Using an Access Controller

Coherence uses an access controller to secure access to cluster resources and operations. It uses a JAAS LoginModule to authenticate a caller, and an access controller on one or more cluster nodes verifies the access rights of the caller’s Subject.

See LoginModule in *Java Authentication and Authorization Service (JAAS) Reference Guide*.

An access controller:

- Verifies proof of the caller’s identity.
- Grants or denies access to a protected clustered resource based on the caller's permissions.
- Encrypts outgoing communications based on the caller's private credentials.

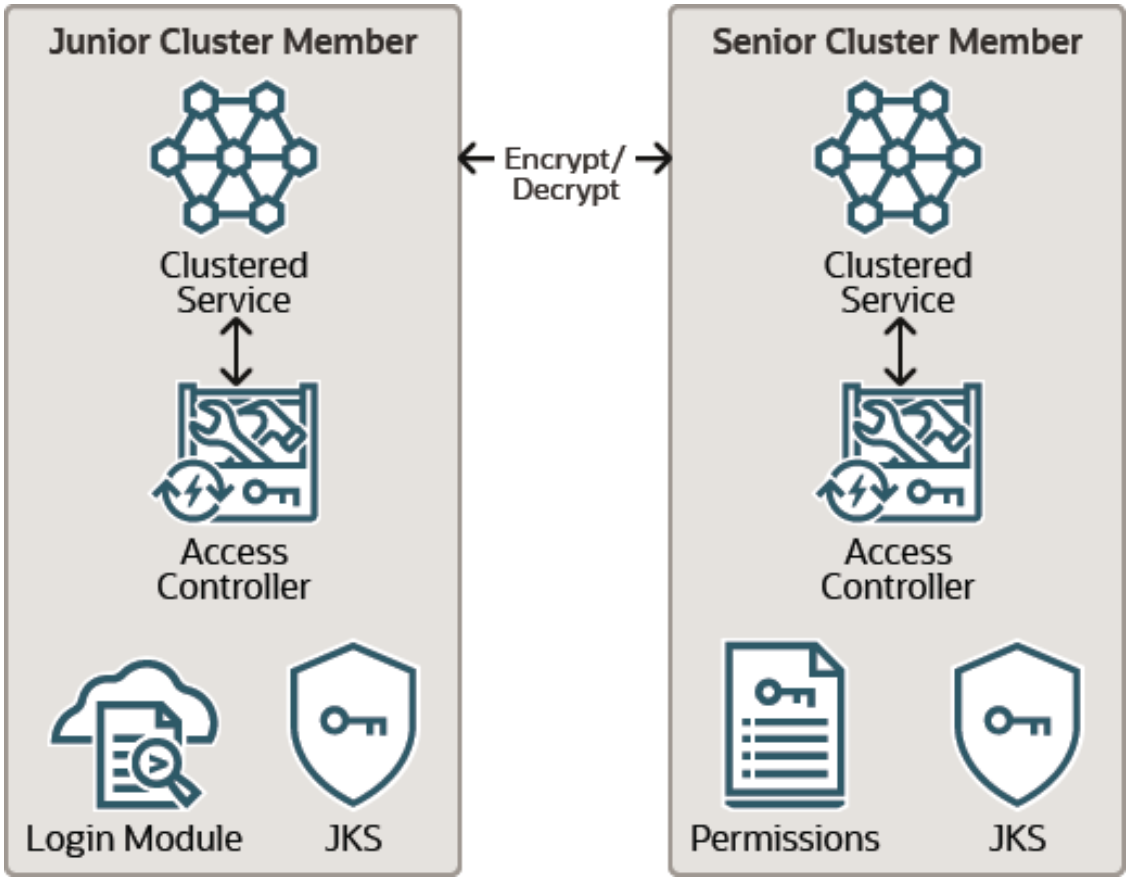
Copyright © 2008, 2026, Oracle and/or its affiliates.

- Decrypts incoming communications based on the caller's public credentials.

Coherence provides a default access controller implementation. The default access controller implementation is based on the key management infrastructure that ships as a standard part of the JDK. See Using the Default Access Controller Implementation.

Figure 4-1 shows a conceptual view of securing two cluster members using access controllers.

Figure 4-1 Conceptual View of Access Controller Security



Understanding the Security Context

Each clustered service maintains the concept of a senior service member that serves as a controlling agent for a particular service. The senior member does not consult with other members when accessing a clustered resource. However, junior member that want to join a service must request and receive a confirmation from the senior member. The senior member notifies all other cluster members about the joining member.

Any cluster member may be the senior member, or may become the senior member when the current senior member dies, so it is important that all cluster members are configured to be able to properly authenticate and authorize all other members.

The security subsystem is designed to operate in a partially hostile environment because data is distributed among cluster members. Every member is considered to be a malicious member. That is, members are assumed to lack sufficient credentials to join a clustered service or obtain access to a clustered resource.

Copyright © 2008, 2026, Oracle and/or its affiliates.

File system mechanisms and standard Java security policies guarantee the trustworthiness of a single node. However, there are two scenarios to consider with member communication:

- A malicious node surpasses the local access check and attempts to join a clustered service or to gain access to a clustered resource that a trusted node controls.
- A malicious node creates a clustered service or clustered resource and becomes its controller.

The security subsystem uses a two-way encryption algorithm to prevent either of these two scenarios from occurring. All client requests must establish proof of identity, and all service responses must establish proof of trustworthiness.

Proof of Identity

Use authenticated subjects and verified signing material for Coherence access-controller decisions.
The Coherence security framework obtains a subject through the configured JAAS login module. The default controller expects the required principals and matching private/public signing material, then verifies the signer before applying permissions. A subject constructed from wire-supplied principal names is not that proof.

Use Using Authenticated Subjects with Modern Java for a current bootstrap/session example and Using an Access Controller for signing and validation-cache behavior. For Extend, use the explicitly configured transformer/asserter chain; the default raw-subject compatibility path is not an authentication mechanism for a hardened deployment.

Preserve trusted key, principal, and permission mappings across cluster members. Additional principals must meet the controller's current binding rules. For stronger propagated subject authority at cluster peer boundaries, use the supported proof configuration and readiness prerequisites in Securing Cluster Peers and Federation.

Configuring the JAAS LoginModule

The LoginModule used by JAAS is configured in a JAAS configuration file. If using the Coherence security APIs to perform the JAAS login, Coherence uses a LoginModule configured with the name “Coherence”.

For example, the JAAS login configuration file below uses the `com.sun.security.auth.module.KeyStoreLoginModule`.

```
Coherence {
    com.sun.security.auth.module.KeyStoreLoginModule required
      keyStoreURL="keystore.jks";
};
```

You can use any suitable LoginModule that produces a Subject with the content required by the access controller. When using the default access controller, the Subject must have at least one Principal, a private key and public certificate pair.

Although not commonly required, you can change the name of the LoginModule used by Coherence by using the `<login-module-name>` element within the `<security-config>` element in the operational configuration file.

```
<security-config>
   <enabled system-property="coherence.security">true</enabled>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
   <login-module-name>MyLoginModule</login-module-name>
</security-config>
```

Then, update the JAAS configuration to include a LoginModule configured with that name:

```
MyLoginModule {
    com.sun.security.auth.module.KeyStoreLoginModule required
      keyStoreURL="keystore.jks";
};
```

The Coherence KeystoreLogin Class

Coherence includes `coherence-login.jar` which contains a LoginModule class named `com.tangosol.security.KeystoreLogin`. This LoginModule can use a Java keystore to obtain a Subject with a principal and a private/public key pair.

You can configure the `KeystoreLogin` LoginModule in the JAAS login configuration file.

```
Coherence {
    com.tangosol.security.KeystoreLogin required
      keyStorePath="keystore.jks";
};
```

A LoginModule configuration contains the LoginModule class name and optionally, a number of key/value pairs that are passed to the LoginModule when it is initialized. At a minimum, the `KeystoreLogin` configuration should include the fully qualified path of the keystore containing the keys and certificates.

The `KeystoreLogin` class supports the following configuration keys and values:

Key	Value
`keyStorePath`	The path to the keystore to use to load the Subject principal, keys, and certificates. The `keyStorePath` value may include property names that are expanded at runtime. For example, if you specify `keyStorePath="${coherence.security.jaas.keystore}"`, the configuration pulls the location of the keystore from the `coherence.security.jaas.keystore` system property.
`keyStoreType`	The keystore type. For backward compatibility, the default is JKS, but PKCS12 is also supported.
`keyStorePass`	The password to use to access the keystore

Configuring a JAAS Callback Handler

If Coherence needs to authorize permissions but the calling thread is not running inside the context of a Subject, then Coherence performs a JAAS login call. Depending on the specific LoginModule JAAS is configured to use, it may require an instance of a `javax.security.auth.callback.CallbackHandler` interface that can provide various parameters to the configured LoginModule.

Copyright © 2008, 2026, Oracle and/or its affiliates.

An access controller will attempt a JAAS login when all other authentication methods have been unsuccessful.

A custom callback handler is configured within the `<security-config>` element in the operational configuration override file. Add a `<callback-handler>` element that includes the fully qualified name of the implementation class. The following example configures a callback handler named `package.MyCallbackHandler`.

```
<security-config>
   <callback-handler>
      <class-name>package.MyCallbackHandler</class-name>
   </callback-handler>
</security-config>
```

Like other class instance XML configurations in Coherence, the `<callback-handler>` element can also be configured with `<init-params>` which will be passed to the class constructor.

```
<security-config>
   <callback-handler>
      <class-name>package.MyCallbackHandler</class-name>
      <init-params>
        <init-param>
          <param-type>String</param-type>
          <param-value>foo</param-value>
        </init-param>
      </init-params>
   </callback-handler>
</security-config>
```

As the callback handler supplies properties required by the JAAS LoginModule, these two classes are typically written and configured to work together.

The Coherence `KeystoreLogin` class passes a `NameCallback` and a `PasswordCallback` to the callback handler. The callback handler should set the `NameCallback` value to be the alias to use to retrieve the private key from the keystore. If the private key and/or keystore are password protected, the callback handler should set the password of the keystore and private key into the `PasswordCallback` value. The restriction here is that both the keystore and private key must use the same password.

Using a Custom JAAS LoginModule

Any custom JAAS LoginModule can be used that produces a Subject containing a principal and private/public keys. For example, you can use Java's built-in `com.sun.security.auth.module.KeyStoreLoginModule` class.

The class should be configured in the JAAS login configuration file:

```
// LoginModule Configuration for Oracle Coherence
Coherence {
    com.sun.security.auth.module.KeyStoreLoginModule required
      keyStoreURL="file://keystore.jks"
      keyStoreAlias="admin"
      keyStorePasswordURL="file://keystore-pass.txt";
};
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

If the LoginModule will require a callback handler, then a suitable callback handler should be configured as described in Configuring a JAAS Callback Handler.

Enable the Access Controller

Before enabling an access controller, you must configure the controller in the Coherence operational configuration file. The default configuration uses Coherence's default access controller, but you can override it to use a custom controller implementation if desired.

Enable the use of an access controller within the <security-config> element of the Coherence operational configuration. Add an `<enabled>` element set to `true`. For example:

```
<security-config>
   <enabled system-property="coherence.security">true</enabled>
</security-config>
```

The `coherence.security` system property or `COHERENCE_SECURITY` environment variable can also be used to enable the access controller. For example:

```
-Dcoherence.security=true
```

> **Note**
>
> When access controller security is enabled, every call to the `CacheFactory.getCache()` or `ConfigurableCacheFactory.ensureCache()` API causes a security check. This negatively affects an application's performance if these calls are made frequently. The best practice is for the application to hold on to the cache reference and reuse it so that the security check is performed only on the initial call. With this approach, ensure that your application only uses the references in an authorized way.

Using the Default Access Controller Implementation

Coherence includes a default access controller implementation that uses a Java keystore for authentication. The implementation class is the `com.tangosol.net.security.DefaultController` class. The access controller is configured within the `<security-config>` element in the operational configuration file.

The default access controller uses private and public keys to both authenticate and authorize a caller and to perform encryption. The default controller makes some assumptions about the sort of JAAS Subject that is being used and how it has been configured.

The default access controller encrypts outgoing permission requests using the private key from the caller's subject. This means that the subject must contain both a principal and a private key. The JAAS LoginModule used must produce a subject containing a principal and corresponding private key.

When authenticating a caller, the default controller expects to find a public key in its keystore using the caller's principal name as the alias. This means that the keystore used by the default controller must contain the correctly aliased public keys for all the possible callers. If each cluster member uses a different principal and private/public key, then the default controller's keystore must contain all of the matching public keys correctly aliased. As any cluster member

Copyright © 2008, 2026, Oracle and/or its affiliates.

could become the senior member, all cluster members must have access to all the public keys in their keystore.

This section includes the following topics:

- The Key Store Type and Encryption Algorithm
- The Default Access Controller Keystore
  The default access controller requires a keystore used to obtain the caller's public keys. This could be the same keystore as was used by the JAAS LoginModule, or it could be a different keystore specifically for access control.
- Create a Permissions File
  The default access controller requires a permissions file that declares access rights for principals. See the *COHERENCE_HOME*`/lib/security/permissions.xsd` schema for the syntax of the permissions file.
- Enable Security Audit Logs
  Security audit logs are used to track the cluster operations that are being performed by each user.
- Configure the Default Access Controller
  The default access controller is configured within the `<security-config>` element in the operational configuration file.

The Key Store Type and Encryption Algorithm

Choose compatible key material and a supported signing algorithm for the access controller. The default access controller's signing algorithm is selected independently of a TLS protocol/cipher. The hardened profile is the default starting with Coherence 26.1.0.0; it selects `SHA256withRSA` and rejects weak algorithms. Explicit compatibility mode retains historical `SHA1withDSA` with a warning. Algorithm configuration must match the actual signing keys and supported provider.

Use provisioned modern key/certificate material and the intended keystore type, such as PKCS12 where supported by the selected login/controller configuration. Do not describe JKS/DSA as the universal requirement for current deployments. Preserve a legacy store only for the supported compatibility path that actually needs it.

Coordinate key, certificate principal, permissions, JAAS, and controller configuration before rollout. Verify signer/principal binding and a denied permission. A new algorithm name with an old incompatible key is not a successful migration. See Using an Access Controller and Verifying and Rotating TLS Identities.

The Default Access Controller Keystore

The default access controller requires a keystore used to obtain the caller's public keys. This could be the same keystore as was used by the JAAS LoginModule, or it could be a different keystore specifically for access control.

When a cluster member uses the default access controller to authorize a request, it tries to load the public key from the keystore using the caller’s principal name as the alias. If different cluster members are using different principals, then the access controller key store must contain the public keys for every cluster member. The keystore does not need to contain the private keys as it is only used for decryption to verify a caller.

The keystore may be password protected, in which case the default access controller must be configured with the password.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Create a Permissions File

The default access controller requires a permissions file that declares access rights for principals. See the *COHERENCE_HOME*/lib/security/permissions.xsd schema for the syntax of the permissions file.

The following example assigns all rights to the Manager principal, only join rights to the Worker principal for caches that have names prefixed by common, and all rights to the Worker principal for the invocation service named invocation.

```
<?xml version='1.0'?>
<permissions>
   <grant>
      <principal>
         <class>javax.security.auth.x500.X500Principal</class>
         <name>CN=Manager,OU=MyUnit</name>
      </principal>
      <permission>
         <target>*</target>
         <action>all</action>
      </permission>
   </grant>
   <grant>
      <principal>
         <class>javax.security.auth.x500.X500Principal</class>
         <name>CN=Worker,OU=MyUnit</name>
      </principal>
      <permission>
         <target>cache=common*</target>
         <action>join</action>
      </permission>
      <permission>
         <target>service=invocation</target>
         <action>all</action>
      </permission>
   </grant>
</permissions>
```

Enable Security Audit Logs

Security audit logs are used to track the cluster operations that are being performed by each user.

Each operation results in a log message being emitted. The messages will be written using the configured Coherence logger. For example:

```
"Destroy" action for cache "Accounts" has been permitted for the user "CN=Bob,
OU=Accounting".
```

Security audit logs are disabled by default but can be enabled in the default access controller configuration.

Configure the Default Access Controller

The default access controller is configured within the <security-config> element in the operational configuration file.

See security-config in *Developing Applications with Oracle Coherence*.

Copyright © 2008, 2026, Oracle and/or its affiliates.

The default operational configuration file already contains all the required configuration for the default access controller. This means that the default controller can be configured by just setting various system properties or environment variables.

The default configuration looks like this:

```
<access-controller>
  <class-name>com.tangosol.net.security.DefaultController</class-name>
  <init-params>
    <init-param id="1">
      <param-type>java.io.File</param-type>
      <param-value system-property="coherence.security.keystore"/>
    </init-param>
    <init-param id="2">
      <param-type>java.io.File</param-type>
      <param-value system-property="coherence.security.permissions"/>
    </init-param>
    <init-param id="3">
      <param-type>boolean</param-type>
      <param-value system-property="coherence.security.log">false</param-
value>
    </init-param>
    <init-param id="4">
      <param-type>String</param-type>
      <param-value system-property="coherence.security.keystore.password"/>
    </init-param>
  </init-params>
</access-controller>
```

The `<class-name>` element is used to set the access controller class name to be the default access controller. This is followed by an `<init-params>` element containing the four `<init-param>` elements that configure the access controller. Each `<init-param>` corresponds to a constructor parameter in the `DefaultController` class constructor.

Table 4-1 `init-param` elements for configuring the access controller

Init Param	System Property or Environment Variable	Description
1	`coherence.security.keystore`	The fully qualified file name of the keystore that the default controller should use. See The Default Access Controller Keystore.
2	`coherence.security.permissions`	The fully qualified file name of the permissions XML file. See Create a Permissions File.
3	`coherence.security.log`	A boolean value that is true to enable audit logging or false to disable audit logging. See Enable Security Audit Logs.
4	`coherence.security.keystore.password`	An optional password for the keystore.

Copyright © 2008, 2026, Oracle and/or its affiliates.

> **Note**
>
> The Coherence default operational configuration file re-uses some system property names in multiple places. For example, both the default access controller and the default SSL socket provider use the same property names for the keystore and credentials. When using both TLS and an access controller, this will work if the application is supposed to use the same keystore for both TLS and for the access controller, but when this is not the case the only way to override these names it to use a custom operational configuration override file.

Configuring a Password Protected Keystore

When using a password protected keystore the password must be provided to the default access controller. This can be done as shown above either in the XML configuration or using a system property or environment variable. As passing credentials using a system property or environment variable is not always the most secure method, it is possible to configure a custom `com.tangosol.net.PasswordProvider` implementation that will be called at runtime to supply the password.

A password provider can be configured for the access controller by adding a `<password-provider>` element to the `<access-controller>` element of the security configuration. The `<password-provider>` element comes after the `<init-prams>` element.

For example, the Coherence `com.tangosol.net.URLPasswordProvider` password provider class, which reads a password from a URL, can be configured in an override XML file like this:

```
<access-controller> <password-provider id="trust-pass"> <class-
name>com.tangosol.net.URLPasswordProvider</class-name> <init-params> <init-param>
<param-type>String</param-type> <param-value>file://store-pass.txt</param-value>
</init-param> </init-params> </password-provider> </access-controller>
```

In the example above, the password would be read from the file at the URL `file://store-pass.txt`.

Using a Custom Access Controller Implementation

You can create a custom access controller implementation if you have specific security requirements that are not addressed by the default implementation. Custom access controllers must implement the `com.tangosol.net.security.AccessController` interface.
To configure a custom access controller within the `<security-config>` element, add an `<access-controller>` element that includes the fully qualified name of the implementation class. The following example configures a custom access controller called `MyAccessController`.

```
<security-config>
   <enabled system-property="coherence.security">true</enabled>
   <access-controller>
      <class-name>package.MyAccessController</class-name>
   </access-controller>
</security-config>
```

Specify any required initialization parameters by using the `<init-params>` element. The following example includes parameters to pass the `MyAccessController` class a keystore and a permissions file.

```
<security-config>
   <enabled system-property="coherence.security">true</enabled>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
   <access-controller>
      <class-name>package.MyAccessController</class-name>
      <init-params>
         <init-param>
            <param-type>java.io.File</param-type>
            <param-value>./keystore.jks</param-value>
         </init-param>
         <init-param>
            <param-type>java.io.File</param-type>
            <param-value>./permissions.xml</param-value>
         </init-param>
      </init-params>
   </access-controller>
</security-config>
```

Writing a custom access controller is not a trivial task as the controller must work how Coherence security internals expect it to. An incorrectly written access controller could inadvertently allow access to that should be blocked.

An access controller has to implement three methods from the `com.tangosol.net.security.AccessController` interface.

```
void checkPermission(ClusterPermission permission, Subject subject);

SignedObject encrypt(Object o, Subject subject)
        throws IOException, GeneralSecurityException;

Object decrypt(SignedObject so, Subject subjectRemote, Subject subjectThis)
        throws ClassNotFoundException, IOException, GeneralSecurityException;
```

- • The `checkPermission` method is used to verify that the passed in `Subject` has the specified permission.
- • The `encrypt` method encrypts a token for the specified `Subject`. This encrypted token will be used to authenticate and authorize the `Subject` by Coherence. Depending on the Coherence APIs being called, this authentication and authorization may happen on locally in the same access controller, or the token may be serialized and sent to a remote cluster member where its access controller will use it to authenticate and authorize the Subject.
- • The `decrypt` method decrypts the specified token using the specified Subjects. Whilst not obvious from the name, this method also verifies trust for the `subjectRemote` Subject, which is the calling Subject that will be requesting permissions. The default controller does this by obtaining a public key from its keystore using the principal name from the `subjectRemote` and using that key to decrypt the token. If no key is present in the local keystore for the `subjectRemote` principal the permissions will be rejected. Any custom access controller must also perform authentication and authorization in the decrypt method to verify trust in the `subjectRemote` Subject. It should then return the decrypted token.

The `encrypt` and `decrypt` methods should obviously work to properly encrypt or decrypt the values. For example, an encrypted token should decrypt to the same value that was encrypted. Coherence typically passes permission instances to the encrypt method, so the decrypt method must return the same value or Coherence will not work.

See Oracle Coherence Community Edition for an example of how to write a custom access controller.

Copyright © 2008, 2026, Oracle and/or its affiliates.

5

Authorizing Access to Server-Side Operations

Attach an operation policy to protected cache storage and distinguish auditing from authorization.
`StorageAccessAuthorizer` checks entry-level and map-level read/write access when it is configured for a partitioned cache. It is opt-in: absent authorization remains absent in either security profile. Configure it on every protected scheme and deploy the implementation and identity mapping consistently to the storage members.

The built-in `auditing` implementation is non-strict and can allow a null subject while logging. `strict-auditing` rejects missing subjects but does not by itself implement business roles or prove that a received subject was authenticated. Use a trusted identity path and a real application policy where role/resource decisions are required.

Use Implementing an Account Storage Authorizer for a complete example. The retained declaration/wiring topics show operational registration and backing-map scheme references. Proxy wrapper authorization can still be useful for its integration scope; storage-side checks cover the configured storage boundary rather than relying only on a particular proxy wrapper.

Operation coverage and cost

The authorizer receives the operation reason as well as the subject and resource context. Test reads, writes, invoke, aggregate, index add/remove, lock/unlock, listeners, and any application-specific path that is exposed. Single-key invoke uses write authorization; lock/unlock checks occur before lock state mutation; configured index removal checks must also handle an absent subject.

Do not use a null subject as an automatic internal-system bypass. Distinguish legitimate internal routes using the supported contract and a reviewed application policy. Authorization code executes on a hot path: local decisions over verified identity and resource data are usually preferable to a remote directory/policy request on every microsecond-scale cache operation. Bound caches and include policy-change/credential-retirement semantics.

Continuous Aggregation uses aggregate authorization for registration, removal, and full-result queries and entry-processor authorization for projections. No separate permission is implied merely because a maintained result is used. Check the feature's release availability and classify custom executable definitions.

- Implementing an Account Storage Authorizer
- Overview of Access Control Authorization
- Creating Access Control Authorization Implementations
- Declaring Access Control Authorization Implementations
- Enabling Access Control Authorization on a Partitioned Cache

Implementing an Account Storage Authorizer

Map verified certificate principals to simple reader/writer permissions and wire the policy to the accounts cache.
This complete Java 21+ example permits reads to a configured reader or writer certificate principal and writes only to the writer. It denies unknown caches, null subjects, and unmatched

Copyright © 2008, 2026, Oracle and/or its affiliates.

identities. It assumes the subject's certificate identity was established through the deployment's trusted authentication path. Adapt the identity-to-role mapping to the approved directory/token integration rather than trusting a role supplied by an unverified client.

```
package com.example.security;

import com.tangosol.net.BackingMapContext;
import com.tangosol.net.security.StorageAccessAuthorizer;
import com.tangosol.util.BinaryEntry;
import javax.security.auth.Subject;
import javax.security.auth.x500.X500Principal;

public final class AccountAuthorizer implements StorageAccessAuthorizer {
    private final X500Principal reader = new X500Principal(System.getProperty(
            "example.accounts.reader", "CN=reader,OU=Accounts,O=Example"));
    private final X500Principal writer = new X500Principal(System.getProperty(
            "example.accounts.writer", "CN=writer,OU=Accounts,O=Example"));

    private void authorize(BackingMapContext context, Subject subject,
boolean write) {
        if (context == null || !"accounts".equals(context.getCacheName())
                || subject == null) {
            throw new SecurityException("Access denied");
        }
        var identities = subject.getPrincipals(X500Principal.class);
        if (!identities.contains(writer) && (write || !
identities.contains(reader))) {
            throw new SecurityException("Access denied");
        }
    }

    @Override
    public void checkRead(BinaryEntry entry, Subject subject, int reason) {
        authorize(entry.getBackingMapContext(), subject, false);
    }
    @Override
    public void checkWrite(BinaryEntry entry, Subject subject, int reason) {
        authorize(entry.getBackingMapContext(), subject, true);
    }
    @Override
    public void checkReadAny(BackingMapContext context, Subject subject, int
reason) {
        authorize(context, subject, false);
    }
    @Override
    public void checkWriteAny(BackingMapContext context, Subject subject, int
reason) {
        authorize(context, subject, true);
    }
}
```

Register it under `<cluster-config><storage-authorizers>` in the operational override:

```
<storage-authorizers>
  <storage-authorizer id="accounts-policy">
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
    <class-name>com.example.security.AccountAuthorizer</class-name>
  </storage-authorizer>
</storage-authorizers>
```

Reference it from the protected distributed scheme's backing-map configuration:

```
<backing-map-scheme>
  <storage-authorizer>accounts-policy</storage-authorizer>
  <local-scheme/>
</backing-map-scheme>
```

Map only the intended cache names to that scheme, deploy the authorizer class, and set the reader/writer principal configuration consistently. Unlike an executable sent by a remote client, a server-configured authorizer is trusted deployment code. Keep its artifact under the same change control as member configuration.

Verification

1. Verify the reader can read and cannot write, invoke a write processor, change indexes, or mutate locks where those routes require write policy.
2. Verify the writer's permitted reads/writes and an unknown principal's denial.
3. Verify a missing subject fails through the actual client and any applicable internal route.
4. Exercise map-wide and single-entry APIs, the actual proxy protocol, failover, and representative load. Check that a proxy-only success does not hide a storage configuration omission.

Overview of Access Control Authorization

Access control authorization allows applications to define their own authorization logic to limit access to cluster operations. Authorization is based on identities that are represented as a `Principal` within a `Subject`.Applications are responsible for ensuring that the `Subject` is present for caller threads. If the `Subject` is missing or cannot be retrieved, then the operation fails with a `SecurityException` error.
Applications implement the `StorageAccessAuthorizer` interface to provide authorization logic. The implementations are declared in the operational override configuration file and must also be enabled on a partitioned cache by configuring the backing map of a distributed scheme in a cache configuration file. Access control authorization is only available for partitioned caches.

The `StorageAccessAuthorizer` interface provides methods that are used to perform read, write, read any, and write any authorization checks. Coherence assumes that there is a logical consistency between authorization decisions made by `StorageAccessAuthorizer` implementations. That is, for a given `Subject`, the write authorization implies the read authorization for a given entry; the read any authorization implies read authorization for all entries; and, the write any authorization implies write and read authorization for all entries.

Table 5-1 lists which authorization checks are caused by `NamedCache` API and `BinaryEntry` API methods.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Table 5-1 Authorization Checks for Common Methods

Authorization Check	NamedCache API Methods	BinaryEntry API Methods
None	• containsKey • containsValue • isEmpty • size • lock • unlock	
Read	• get • getAll	• getValue • getBinaryValue • extract • getOriginalValue • getOriginalBinaryValue
Write	• invoke • put • putAll • remove • removeAll	• setValue • update • updtaeBinaryValue • remove • expire
Read Any	• addMapListener[1] • aggregate • entrySet • keySet • removeMapListener[1]	
Write Any	• addIndex • clear • invokeAll • removeIndex • values	

[1] If a listener is a `MapTriggerListener`, then a Write Any authorization check is performed instead.

Creating Access Control Authorization Implementations

Implement all storage-authorizer callbacks and validate both resource and authenticated identity.
Implement `StorageAccessAuthorizer` to decide entry-level and map-level read/write access. The callbacks receive a subject and reason plus the entry or backing-map context. Use those inputs to define a deliberate policy for the protected cache and operation.

Callback	Resource scope
`checkRead`	An entry read.
`checkWrite`	An entry mutation or write-authorized operation.
`checkReadAny`	Map-level read scope.
`checkWriteAny`	Map-level mutation scope.

Implementing an Account Storage Authorizer provides a complete current example and wiring. Handle null subjects explicitly and consider reason-specific restrictions where the policy

Copyright © 2008, 2026, Oracle and/or its affiliates.

distinguishes invoke, aggregation, index, or lock operations. Do not assume every caller is a client or that an internal route always has a subject.

Audit-only implementations can be useful during migration, but logging an operation is not enforcing a role. Use structured/bounded operation and resource metadata and avoid logging cache keys, values, secrets, or an entire credential-bearing subject. Measure hot-path cost with the real policy before deployment.

Declaring Access Control Authorization Implementations

Access control authorization implementations must be declared so that the class is loaded when a cluster starts. Multiple authorization implementations can be created and are referenced using a unique identification.
To declare access control authorizer implementations, edit the operational override file and include a `<storage-authorizers>` element, within the `<cluster-config>` element, and declare each authorization implementation using a `<storage-authorizer>` element. See storage-authorizer in *Developing Applications with Oracle Coherence*. Each declaration must include a unique `id` attribute that is used by a partitioned cache to select an implementation. For example:

```
<cluster-config>
   <storage-authorizers>
      <storage-authorizer id="LogAuthorizer">
         <class-name>package.MyLogAuthorizer</class-name>
      </storage-authorizer>
   </storage-authorizers>
</cluster-config>
```

As an alternative, the `<storage-authorizer>` element supports the use of a `<class-factory-name>` element to use a factory class that is responsible for creating instances and a `<method-name>` element to specify the static factory method on the factory class that performs object instantiation. For example:

```
<cluster-config>
   <storage-authorizers>
      <storage-authorizer id="LogAuthorizer">
         <class-factory-name>package.MyAuthorizerFactory</class-factory-name>
         <method-name>getAuthorizer</method-name>
      </storage-authorizer>
   </storage-authorizers>
</cluster-config>
```

Any initialization parameters that are required for an implementation can be specified using the `<init-params>` element. For example:

```
<cluster-config>
   <storage-authorizers>
      <storage-authorizer id="LogAuthorizer">
         <class-name>package.MyLogAuthorizer</class-name>
         <init-params>
            <init-param>
               <param-name>f_fStrict</param-name>
               <param-value>true</param-value>
            </init-param>
         </init-params>
      </storage-authorizer>
   </storage-authorizers>
</cluster-config>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

Enabling Access Control Authorization on a Partitioned Cache

A partition cache service must be configured to use an access control authorization implementation. The implementation is enabled in the cache definition and is reference by name.

To enable access control authorization on a partitioned cache, edit the cache configuration file and add a `<storage-authorizer>` element, within the `<backing-map-scheme>` element of a distributed scheme, whose value is the `id` attribute value of an authorization implementation that is declared in the operational override file. For example:

```
<distributed-scheme>
   ...
   <backing-map-scheme>
      <storage-authorizer>LogAuthorizer</storage-authorizer>
      <local-scheme/>
   </backing-map-scheme>
   <autostart>true</autostart>
</distributed-scheme>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

6

Securing Extend Client Connections

Configure transport, explicit identity assertion, operation authorization, and executable policy for Extend services.
Define an SSL socket provider on the proxy TCP acceptor and the matching client initiator. Verify CA trust and the server name, and supply a client identity when mTLS is required. The retained TLS topics show placement; Configuring TLS with Protected Password Sources supplies the reusable provider recipe.

Configure the proxy identity asserter and client identity transformer as a deliberate pair. The token must be safely materialized and then authenticated. Receiving a raw subject is not proof that its principals were authenticated. Overview of Using Identity Tokens describes the default-profile contract and the explicit chain.

Apply storage/service authorization to protected operations and classify application processors, aggregators, filters, extractors, comparators, triggers, and invocables. The same payload can pass decoding and still be rejected by execution or caller policy; use Classifying Remote Executable Objects.

Remote invocation exposure

The shipped hardened posture disables the Extend InvocationService proxy unless explicitly enabled. Set `coherence.invocation.enabled=true` or the operational proxy setting only when remote invocation is intended. Explicit property configuration takes precedence as documented. Enabling the proxy does not bypass executable classification or caller authorization.

The Concurrent proxy has its own autostart control, `coherence.concurrent.extend.enabled`. Inventory it separately. Cache service access, remote invocation, and distributed task execution need their own allowed-operation tests.

Verification

Test trusted and untrusted certificates, missing/invalid tokens, a valid identity with denied permissions, and an unclassified executable. Establish the subject on the actual remote channel, including reconnection and failover. Keep clients with different identities from accidentally sharing subject-scoped handles or credentials.

- Using Identity Tokens to Restrict Client Connections
- Associating Identities with Extend Services
- Implementing Extend Client Authorization

Using Identity Tokens to Restrict Client Connections

Identity tokens are used to control which clients can access a cluster. The token is sent between extend clients and extend proxies whenever a connection is attempted. Only extend clients that pass a valid identity token are allowed to access the cluster.
This section includes the following topics:

- Overview of Using Identity Tokens
- Creating a Custom Identity Transformer

Copyright © 2008, 2026, Oracle and/or its affiliates.

- Enabling a Custom Identity Transformer
- Creating a Custom Identity Asserter
- Enabling a Custom Identity Asserter
- Using Custom Security Types
- Understanding Custom Identity Token Interoperability

Overview of Using Identity Tokens

Pair a client token transformer with a proxy asserter that verifies identity before authorizing operations.
An `IdentityTransformer` produces the token associated with a client subject/service. An `IdentityAsserter` validates the received token and returns the subject that downstream policy can use. Token decoding and authentication are distinct steps; a safe data object may still contain forged identity claims.

The default transformer returns the subject. Only an explicitly selected `coherence.security.mode=compatibility` retains the default asserter's raw-Subject pass-through. The hardened profile, which is the default starting with Coherence 26.1.0.0 when the property is unset, rejects non-null tokens, including a raw subject. Null-token behavior remains available for deliberately unauthenticated clients; require an explicit trusted chain when anonymous access is inappropriate.

Use the provided JAAS credential asserter or a reviewed custom identity integration. Validate token type, issuer/credential, intended service/audience, validity, and required identity properties according to the selected token system. Do not accept a subject merely because it contains a role name.

Transformers and asserters run at the documented connection/channel creation points. Deploy the transformer on the client and asserter on the proxy, and configure authentication on any different protocol independently. Propagate the verified subject to storage authorization through the supported routes. Cluster members themselves remain a separate trust boundary; stronger subject proof is described in Securing Cluster Peers and Federation.

Creating a Custom Identity Transformer

Transform a deliberately established local credential into the token contract accepted by the proxy.
Choose the transformer to match the proxy's asserter. For a JAAS username/password chain, an application may place a short-lived `UsernameAndPassword` in the local subject's private credentials and use the following Java 21+ transformer. The subject must be created by trusted application login/credential code and used only for the intended remote session.

```
package com.example.security;

import com.tangosol.net.Service;
import com.tangosol.net.security.IdentityTransformer;
import com.tangosol.net.security.UsernameAndPassword;
import javax.security.auth.Subject;

public final class CredentialIdentityTransformer implements
IdentityTransformer {
    @Override
    public Object transformIdentity(Subject subject, Service service) {
        if (subject == null) {
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
            throw new SecurityException("Credentials required");
        }
        var credentials =
subject.getPrivateCredentials(UsernameAndPassword.class);
        if (credentials.size() != 1) {
            throw new SecurityException("Exactly one credential is required");
        }
        return credentials.iterator().next();
    }
}
```

Configure this class under the client operational `<security-config><identity-transformer>`. The caller owns the source password and subject lifetime: keep credentials available through required channel creation/reconnection, remove them and clear owned arrays after the session is closed, and avoid sharing the subject across unrelated users. Never clear an array immediately on returning a token while the transport still needs to serialize it.

This example defines a username/password service credential flow. For a signed token or external identity provider, use that provider's verified contract and refresh/audience rules rather than inventing claims in the transformer. The proxy must independently validate the token through its configured asserter.

Enabling a Custom Identity Transformer

To enable a custom identity transformer implementation, edit the client-side `tangosol-coherence-override.xml` file and add an `<identity-transformer>` element within the `<security-config>` node. The element must include the full name of the implementation class. For example:

```
<?xml version='1.0'?>

<coherence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://xmlns.oracle.com/coherence/coherence-operational-config"
   xsi:schemaLocation="http://xmlns.oracle.com/coherence/coherence-operational-config
   coherence-operational-config.xsd">
   <security-config>
      <identity-transformer>
        <class-name>com.my.PasswordIdentityTransformer</class-name>
      </identity-transformer>
   </security-config>
</coherence>
```

Creating a Custom Identity Asserter

Prefer the provided JAAS adapter where suitable and customize assertion only for a defined trusted token contract.
For username/password tokens, `com.tangosol.net.security.JAASIdentityAsserter` adapts the token to a named JAAS login entry. Configure its constructor with the intended entry name. Authenticating Credentials with JAAS includes both that configuration path and an example of a custom adapter when additional subject validation is needed.

A custom `IdentityAsserter` must reject unsupported token types, failed verification, and unusable identities. Return the verified subject needed by downstream authorization; returning null after a shared-password comparison does not establish a user or business role. Use transport encryption and bounded credential failure handling.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Do not use a password system property as an identity directory, accept an unverified raw subject as proof, or expose secrets in messages/logs. Define credential rotation, token expiry, audience/service binding, and the lifecycle of any retained subject resources. Package custom application token types with the required serialization policy before they can reach the asserter.

Enabling a Custom Identity Asserter

To enable a custom identity asserter implementation, edit the cluster-side `tangosol-coherence-override.xml` file and add an `<identity-asserter>` element within the `<security-config>` node. The element must include the full name of the implementation class. For example:

```
<?xml version='1.0'?>

<coherence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://xmlns.oracle.com/coherence/coherence-operational-config"
   xsi:schemaLocation="http://xmlns.oracle.com/coherence/coherence-operational-config
   coherence-operational-config.xsd">
   <security-config>
      <identity-asserter>
        <class-name>com.my.PasswordIdentityAsserter</class-name>
      </identity-asserter>
   </security-config>
</coherence>
```

Using Custom Security Types

Security objects are automatically serialized and deserialized using Portable Object Format (POF) when they are passed between extend clients and extend proxies. Security objects that are predefined in POF require no configuration or programming changes. However, security objects that are not predefined in POF (for example, when an application uses Kerberos authentication) cause an error. For custom security types, an application must convert the custom type or define the type in POF. There are two approaches for using unsupported types.

Converting the Type

The custom identity transformer implementation converts a custom security object type to a type that is predefined for POF, such as a character array or string, before returning it as an object token. On the proxy server, the custom identity asserter implementation converts the object back (after validation) to a `Subject`.

For example, a subject may contain credentials that are not serialized. The identity transformer implementation extracts the credential and converts it to a character array, returning that array as the token. On the proxy server, the identity asserter converts the character array to the proper credential type, validates it, and then constructs a `Subject` to return.

Defining the Custom Type in POF

You can define the custom security types in both the client's and the proxy's POF configuration file. For detailed information about using POF with Java, see Using Portable Object Format in *Developing Applications with Oracle Coherence*. For more information about using POF with C++ and C#, see Building Integration Objects (C++) and Building Integration Objects (.NET), respectively in *Developing Remote Clients for Oracle Coherence*.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Understanding Custom Identity Token Interoperability

Solutions that use a custom identity token must always consider what tokens may be sent by an extend client and what tokens may be received by an extend proxy. This is particularly important during rolling upgrades and when a new custom identity token solution is implemented.

Oracle Coherence Upgrades

Interoperability issues may occur during the process of upgrading. In this scenario, different client versions may interoperate with different proxy server versions. Ensure that a custom identity asserter can handle identity tokens sent by an extend client. Conversely, ensure that a custom identity transformer sends a token that the extend proxy can handle.

Custom Identity Token Rollout

Interoperability issues may occur between extend clients and extend proxies during the roll out a custom identity token solution. In this scenario, as extend proxies are migrated to use a custom identity asserter, some proxies continue to use the default asserter until the rollout operation is completed. Likewise, as extend clients are migrated to use a custom identity transformer, clients continue to use the default transformer until the rollout operation is completed. In both cases, the extend clients and extend proxies must be able to handle the default token type until the rollout operation is complete.

One strategy for such a scenario is to have a custom identity asserter that accepts the default token types temporarily as clients are updated. The identity asserter checks an external source for a policy that indicates whether those tokens are accepted. After all clients have been updated to use a custom token, change the policy to accept the custom tokens.

Associating Identities with Extend Services

Create and use subject-scoped remote handles under the intended authenticated context. An Extend connection/channel can associate an identity token with the service context. Configure the client transformer and proxy asserter to agree on a verified token contract, and configure subject scoping as required by the application. A locally authenticated subject still needs that explicit remote assertion path.

Use current subject-context APIs and acquire the session/map handles within the intended identity context. See Using Authenticated Subjects with Modern Java and Creating a Custom Identity Transformer. Keep the Coherence instance/session lifetime deliberate and avoid sharing one identity's handles across unrelated users.

Verify identity propagation with a real protected operation, reconnection, and failover. Do not assume all asynchronous callbacks or arbitrary executors inherit context. Use the protocol's supported propagation and test the specific path rather than relying on a thread-local assumption.

Implementing Extend Client Authorization

Oracle Coherence*Extend authorization controls which operations can be performed on a cluster based on an extend client's access rights. Authorization logic is implementation-specific and is enabled on a cluster proxy. The code samples in this section are based on the Java authorization example, which is included in the examples that are delivered as part of the distribution. The example demonstrates a basic authorization implementation that uses the

Copyright © 2008, 2026, Oracle and/or its affiliates.

Principal obtained from a client request and a role-based policy to determine whether to allow operations on the requested service. Download the examples for the complete implementation. This section includes the following topics:

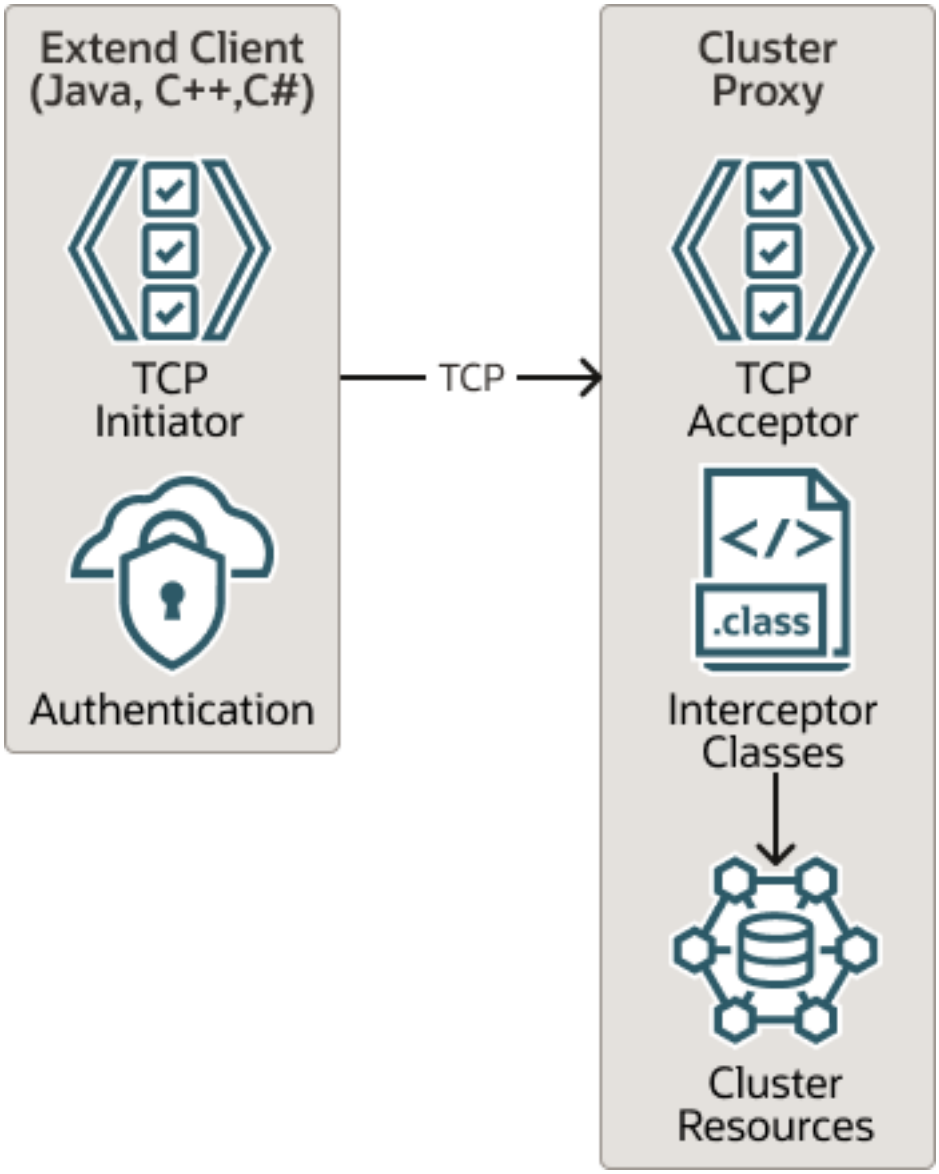
- Overview of Extend Client Authorization
- Create Authorization Interceptor Classes
- Enable Authorization Interceptor Classes

Overview of Extend Client Authorization

Interceptor classes provide the ability to implement client authorization. An extend proxy calls the interceptor classes before a client accesses a proxied resource (cache, cache service, or invocation service). Interceptor classes are implementation-specific. They must provide the necessary authorization logic before passing the request to the proxied resources.

Figure 6-1 shows a conceptual view of extend client authorization.

Figure 6-1 Conceptual View of Extend Client Authorization



Copyright © 2008, 2026, Oracle and/or its affiliates.

Create Authorization Interceptor Classes

To create interceptor classes for both a proxied cache service and a proxied invocation service, implement the `CacheService` and `InvocationService` interfaces, respectively. Or, as is more common, extend a set of wrapper classes: `com.tangosol.net.WrapperCacheService` (with `com.tangosol.net.cache.WrapperNamedCache`) and `com.tangosol.net.WrapperInvocationService`. The wrapper classes delegate to their respective interfaces and provide a convenient way to create interceptor classes that apply access control to the wrapped interface methods.

Example 6-1 is taken from the Oracle Coherence examples. The example demonstrates creating an authorization interceptor class for a proxied invocation service by extending `WrapperInvocationService`. It wraps all `InvocationService` methods on the proxy and applies access controls based on the `Subject` passed from an extend client. The implementation allows only a `Principal` with a specified role name to access the `InvocationService` methods.

Example 6-1 Extending the WrapperCacheService Class for Authorization

```
public class EntitledCacheService
   extends WrapperCacheService
   {
   public EntitledCacheService(CacheService service)
       {
       super(service);
       }

   public NamedCache ensureCache(String sName, ClassLoader loader)
      {
      SecurityExampleHelper.checkAccess(SecurityExampleHelper.ROLE_READER);
      return new EntitledNamedCache(super.ensureCache(sName, loader));
      }

   public void releaseCache(NamedCache map)
      {
      if (map instanceof EntitledNamedCache)
         {
         EntitledNamedCache cache =  (EntitledNamedCache) map;
         SecurityExampleHelper.checkAccess(SecurityExampleHelper.ROLE_READER);
         map = cache.getNamedCache();
         }
      super.releaseCache(map);
      }

   public void destroyCache(NamedCache map)
      {
      if (map instanceof EntitledNamedCache)
         {
         EntitledNamedCache cache =  (EntitledNamedCache) map;
         SecurityExampleHelper.checkAccess(SecurityExampleHelper.ROLE_ADMIN);
         map = cache.getNamedCache();
         }
      super.destroyCache(map);
      }
}
```

Notice that the `EntitledCacheService` class requires a named cache implementation. The `WrapperNamedCache` class is extended and wraps each method of the `NamedCache` instance. This allows access controls to be applied to different cache operations.

Copyright © 2008, 2026, Oracle and/or its affiliates.

> **Note**
>
> Much of the functionality that is provided by the `WrapperNamedCache` class is also covered by the `StorageAccessAuthorizer` interface, which provides a better and simplified way to authorize cluster operations. See Authorizing Access to Server-Side Operations.

Example 6-2 is a code excerpt taken from the Oracle Coherence examples. The example demonstrates overriding the `NamedCache` methods and applying access checks before allowing the method to be executed. See the examples for the complete class.

Example 6-2 Extending the WrapperNamedCache Class for Authorization

```
public class EntitledNamedCache
   extends WrapperNamedCache
   {
   public EntitledNamedCache(NamedCache cache)
      {
       super(cache, cache.getCacheName());
      }

   public Object put(Object oKey, Object oValue, long cMillis)
      {
      SecurityExampleHelper.checkAccess(SecurityExampleHelper.ROLE_WRITER);
      return super.put(oKey, oValue, cMillis);
      }

   public Object get(Object oKey)
      {
      SecurityExampleHelper.checkAccess(SecurityExampleHelper.ROLE_READER);
      return super.get(oKey);
      }

   public void destroy()
      {
      SecurityExampleHelper.checkAccess(SecurityExampleHelper.ROLE_ADMIN);
      super.destroy();
      }
...
```

Example 6-3 is taken from the Oracle Coherence examples. The example demonstrates creating an authorization interceptor class for a proxied cache service by extending the `WrapperCacheService` class. It wraps all `CacheService` methods on the proxy and applies access controls based on the `Subject` passed from an extend client. The implementation allows only a `Principal` with the specified role to access the `CacheService` methods

Example 6-3 Extending the WrapperInvocationService Class for Authorization

```
public class EntitledInvocationService
   extends WrapperInvocationService
   {
   public EntitledInvocationService(InvocationService service)
      {
      super(service);
      }

   public void execute(Invocable task, Set setMembers, InvocationObserver
      observer)
      {
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
      SecurityExampleHelper.checkAccess(SecurityExampleHelper.ROLE_WRITER)
      super.execute(task, setMembers, observer);
      }

   public Map query(Invocable task, Set setMembers)
      {
      SecurityExampleHelper.checkAccess(SecurityExampleHelper.ROLE_WRITER)
      return super.query(task, setMembers);
      }
}
```

When a client attempts to use a remote invocation service, the proxy calls the `query()` method on the `EntitledInvocationService` class, rather than on the proxied `InvocationService` instance. The `EntitledInvocationService` class decides to allow or deny the call. If the call is allowed, the proxy then calls the `query()` method on the proxied `InvocationService` instance.

Enable Authorization Interceptor Classes

To enable interceptor classes for a proxied cache service and a proxied invocation service, edit a proxy scheme definition and add a `<cache-service-proxy>` element and `<invocation-service-proxy>` element, respectively. Use the `<class-name>` element to enter the fully qualified name of the interceptor class. Specify initialization parameters using the `<init-params>` element. See cache-service-proxy and invocation-service-proxy in *Developing Applications with Oracle Coherence* for detailed information about using these elements.

The following example demonstrates enabling interceptor classes for both a proxied cache service and a proxied invocation service. The example uses the interceptor classes from Example 6-1 and Example 6-3.

```
<proxy-scheme>
   ...
   <proxy-config>
      <cache-service-proxy>
         <class-name>
            com.tangosol.examples.security.EntitledCacheService
         </class-name>
         <init-params>
            <init-param>
               <param-type>com.tangosol.net.CacheService</param-type>
               <param-value>{service}</param-value>
            </init-param>
         </init-params>
      </cache-service-proxy>
      <invocation-service-proxy>
         <class-name>
            com.tangosol.examples.security.EntitledInvocationService
         </class-name>
         <init-params>
            <init-param>
               <param-type>com.tangosol.net.InvocationService</param-type>
               <param-value>{service}</param-value>
            </init-param>
         </init-params>
      </invocation-service-proxy>
</proxy-config>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

7

Securing Serialization and Remote Execution

Control which application types may be decoded and which objects may execute remotely, then stage enforcement using application artifacts and workload verification.
A distributed operation carries more than data. Entry processors, aggregators, filters, extractors, comparators, invocables, and tasks can cause code to run on another process. Coherence applies checks during decoding and at executable installation or invocation. User authorization is a separate policy applied to the authenticated caller and requested resource.

1. Define the application types that may cross each service boundary and package their security configuration.
2. Classify remote executable classes and functional-interface targets; classify nested state as well as the outer object.
3. Configure serializer and protocol-specific policies, including POF registration, JSON aliases, and gRPC serializer names.
4. Exercise the actual client/server workflows under hardened policy before enabling it across the deployment.

Start with Generating Application Security Configuration, then use Classifying Remote Executable Objects and Choosing a Lambda Serialization Policy. Diagnose failures with Security Errors and Diagnostic Signals.

Use Developing Classified Application Types for complete record and lite/POF processor examples with explicit type/executable configuration.

Layered Input Validation

Layered validation allows Coherence to fail closed for common deserialization entry points. It adds bytecode deny-list checks, JEP-290 filter enforcement, strict POF type handling, dynamic class-definition validation, generated application allowlists, security-mode-aware defaults, and per-boundary telemetry.

The compatibility model has two independent axes:

- `coherence.mode` remains the released runtime/license mode switch. It is independent of the security profile.
- `coherence.security.mode` controls the compatibility-sensitive serialization and remote-execution gates. Starting with Coherence 26.1.0.0, an unset property selects `hardened`. Set `-Dcoherence.security.mode=compatibility` explicitly when a migration needs the older compatibility-sensitive behavior and shadow telemetry.

Why not JEP-290 alone?

`java.io.ObjectInputFilter` (JEP-290) is a transport-level filter wired to `ObjectInputStream`. Coherence's dominant wire formats (POF, `ExternalizableLite`, `PortableObject`) bypass `ObjectInputStream` entirely, so a JEP-290 filter alone would silently miss the majority of deserialization traffic. serialization hardening also needs the same allowlist to drive three distinct gates -- class allowlist on deserialization, lambda-bytecode acceptance, and remote-execution policy -- and to compose across many modules at build time with annotation-driven generation. JEP-290's single allow/reject bit and flat filter string cannot express that. Coherence therefore uses `security-config.xml` as the source of truth, has every wire-format

Copyright © 2008, 2026, Oracle and/or its affiliates.

reader consult it, and registers a JEP-290 filter that delegates back to the same allowlist for the paths that do run through `ObjectInputStream`. JEP-290 is a hook, not a substitute, and customer-supplied `jdk.serialFilter` settings still compose normally for non-Coherence streams.

Compatibility and Existing Payloads

`coherence.security.mode` controls how serialization hardening gates are applied while applications migrate to generated `security-config.xml` and explicit remote-executable classification.

Security mode	Allowlist gate: `isAllowlistEnforced()`	DYNAMIC remote default: `isDynamicRemoteDefaultDeny()`	Remote executable gate: `isRemoteExecutableEnforced()`	Use
Explicit `compatibility`	`false`	`false`	`false`	Explicit migration profile. Set `coherence.security.mode=compatibility` before startup when the upgrade plan needs older compatibility-sensitive behavior. Selected gates emit `result=would_reject` shadow telemetry in either runtime mode; omission of the property does not select this profile starting with Coherence 26.1.0.0.
`hardened` or unset (starting with 26.1.0.0)	`true`	`false`	`true`	Enforced security profile. Generated allowlists and named remote-executable classification are enforced. The current integration default permits unauthenticated dynamic remote code; set `coherence.remote.dynamic.unauthenticated=deny` explicitly to refuse those paths.

Trust Application Code Deliberately

An allowlisted class is trusted application code. Its constructors, read methods, resolution hooks, and invoked methods can use resources available to the member process. Review

Copyright © 2008, 2026, Oracle and/or its affiliates.

dependencies and code changes with the same care as member-side code. Class allowlisting is not an isolation sandbox, and it does not replace filesystem, process, or network restrictions.

Use exact application classes where practical. Broad package entries make every matching future class eligible. Built-in deny rules are evaluated independently and cannot be bypassed merely by adding an allow entry. Preserve those boundaries when migrating an existing payload.

- Developing Classified Application Types
- Generating Application Security Configuration
- Classifying Remote Executable Objects
- Choosing a Lambda Serialization Policy

Developing Classified Application Types

Use modern data and processor examples with explicit serialization, POF registration, and executable classification.

An immutable application value

This Java 21+ record uses ordinary Java serialization and an explicit application allowlist annotation. Compile the value module and run security-config generation for that module. The annotation classifies the type; it does not select a serializer or define a POF type ID.

```
package com.example.security;

import com.tangosol.util.function.Remote;
import java.io.Serializable;
import java.math.BigDecimal;
import java.util.Objects;

@Remote.Allowed
public record Invoice(String id, BigDecimal amount) implements Serializable {
    public Invoice {
        Objects.requireNonNull(id, "id");
        Objects.requireNonNull(amount, "amount");
    }
}
```

Select a compatible serializer on the actual receiving path. If the application uses POF, provide its supported record/model mapping and registration instead of assuming `Serializable` alone supplies a POF codec. For gRPC, the separately approved client serializer format still applies.

A purpose-built remote processor

The following complete processor supports the Coherence lite and POF codecs explicitly and has a public no-argument constructor for realization. It operates on non-null integer values, creates a missing value from zero, and rejects arithmetic overflow through `Math.addExact`. It requires Java 21+ and the application security-generation step.

```
package com.example.security;

import com.tangosol.io.ExternalizableLite;
import com.tangosol.io.pof.PofReader;
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
import com.tangosol.io.pof.PofWriter;
import com.tangosol.io.pof.PortableObject;
import com.tangosol.util.InvocableMap;
import com.tangosol.util.function.Remote;
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

@Remote.Executable
public final class IncrementBalance
        implements InvocableMap.EntryProcessor<String, Integer, Integer>,
                   ExternalizableLite, PortableObject {
    private int delta;

    public IncrementBalance() {
    }

    public IncrementBalance(int delta) {
        this.delta = delta;
    }

    @Override
    public Integer process(InvocableMap.Entry<String, Integer> entry) {
        int current = entry.isPresent() ? entry.getValue() : 0;
        int updated = Math.addExact(current, delta);
        entry.setValue(updated);
        return updated;
    }

    @Override
    public void readExternal(DataInput in) throws IOException {
        delta = in.readInt();
    }

    @Override
    public void writeExternal(DataOutput out) throws IOException {
        out.writeInt(delta);
    }

    @Override
    public void readExternal(PofReader in) throws IOException {
        delta = in.readInt(0);
    }

    @Override
    public void writeExternal(PofWriter out) throws IOException {
        out.writeInt(0, delta);
    }
}
```

For a POF deployment, save the following as an application file such as `billing-pof-config.xml`, use a type ID allocated uniquely within the application's registry, and select this

Copyright © 2008, 2026, Oracle and/or its affiliates.

configuration on every participating process. The example ID is illustrative; it is not an Oracle-assigned application ID.

```
<pof-config xmlns="http://xmlns.oracle.com/coherence/coherence-pof-config">
  <user-type-list>
    <include>coherence-pof-config.xml</include>
    <user-type>
      <type-id>10201</type-id>
      <class-name>com.example.security.IncrementBalance</class-name>
    </user-type>
  </user-type-list>
</pof-config>
```

Deploy the processor class and generated `META-INF/coherence/security-config.xml` on the receiver. Its generated entry has `executable="true"`; POF registration does not replace that classification. With a typed map obtained from the intended authenticated session, an invocation is:

```
NamedMap<String, Integer> balances = session.getMap("balances");
int updated = balances.invoke("account-1", new IncrementBalance(5));
```

Use an authorization policy and cache mapping appropriate to `balances`; the accounts-only policy elsewhere in this guide deliberately denies other cache names. Verify allowed execution and a denied caller, then verify that an unclassified test processor is rejected in an isolated hardened deployment. No global dynamic/reflection allowance is needed merely to run this purpose-built processor.

Generating Application Security Configuration

Generate and package security configuration for every application artifact that contributes serialized payload types or remote executables.
The build-time generator scans compiled classes in the current artifact. It does not scan dependency jars. Each jar that contributes wire payload types should generate and package its own `META-INF/coherence/security-config.xml`.

Maven

The `security-config-maven-plugin` goal is named `generate` and is bound to `process-classes` by default.

```
<plugin>
  <groupId>com.oracle.coherence</groupId>
  <artifactId>security-config-maven-plugin</artifactId>
  <version>${coherence.version}</version>
  <executions>
    <execution>
      <id>generate-security-config</id>
      <phase>process-classes</phase>
      <goals>
        <goal>generate</goal>
      </goals>
    </execution>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
  </executions>
</plugin>
```

Set `-Dcoherence.security-config.skip=true` to skip generation for a build.

Gradle

The Coherence Gradle plugin registers the `securityConfig` task when the Java plugin is applied.

```
plugins {
    id 'java'
    id 'com.oracle.coherence.gradle.plugin' version coherenceVersion
}

securityConfig {
    processTestClasses = false
}
```

Set `processTestClasses = true` to also generate `securityConfigTest` output for test classes.

Annotations

Use `com.tangosol.util.function.Remote.Allowed` for application classes that may legitimately cross a wire-deserialization boundary as polymorphic payloads.

At type level, `@Remote.Allowed` includes the annotated class and, by default, its declared nested types. Set `recursive = false` to include only the annotated class:

```
// Generates allowlist entries for MyType AND MyType.Nested.
@Remote.Allowed
public class MyType
    {
    static class Nested {}
    }

// Generates an allowlist entry for MyType only; Nested is excluded.
@Remote.Allowed(recursive = false)
public class MyType
    {
    static class Nested {}
    }
```

At package level, add `@Remote.Allowed` to `package-info.java`. The default includes classes in that package and subpackages; `recursive = false` limits generation to the package itself.

`com.tangosol.io.pof.schema.annotation.PortableType` also generates allowlist entries. A class does not need both annotations unless your source code uses both concepts for clarity.

The generated artifact is `META-INF/coherence/security-config.xml`. At member start, Coherence loads every such resource visible to the thread context class loader, validates each resource against `coherence-security-config.xsd`, and merges the allowed class names into a single runtime allowlist. A malformed or invalid resource fails member startup and names the offending URL. The schema is shipped as `coherence-security-config.xsd` in the Coherence jar.

Copyright © 2008, 2026, Oracle and/or its affiliates.

A generated file looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<security-config xmlns="http://xmlns.oracle.com/coherence/coherence-security-
config" version="1.0">
  <allowed-classes>
    <class name="com.acme.billing.InvoiceFilter"
source="@Remote.Allowed"/>
    <class name="com.acme.billing.InvoiceProcessor"
source="@Remote.Allowed"/>
    <class name="com.acme.billing.model.Invoice"
source="@PortableType"/>
    <class name="com.acme.billing.util.MoneyExtractor"
source="@Remote.Allowed(package)"/>
  </allowed-classes>
</security-config>
```

Each jar on the runtime classpath contributes its own `META-INF/coherence/security-config.xml`; the runtime unions all of them into a single allowlist. This means multi-module applications do not need a single curated file - each module that sends wire payload classes ships its own generated resource. A class that is not listed in any generated file and is not added via `coherence.serialization.allowed` is rejected at the gate.

"Wire payload classes" means more than the obvious entry processors, filters, aggregators, and POF data types. Any class whose state crosses a Coherence wire boundary is subject to the allowlist gate, including classes captured by the closure of a serialized lambda. If your code writes `cache.invokeAll(filter, entry -> doSomething(entry, helper))` and `helper` is an instance of `com.acme.PricingHelper`, then `PricingHelper` is part of the lambda's serialized state and must be allowlisted on the receiver. The typical fix is to annotate the helper class with `@Remote.Allowed` or apply a package-level annotation. After upgrade, the most common `security-config-missing` rejections come from such capturer classes rather than from obvious payload types.

Verify the Packaged Artifact

```
jar tf billing-model.jar
jar tf billing-processors.jar
```

Each contributing JAR must contain `META-INF/coherence/security-config.xml`. Inspect the resource itself and confirm the expected fully qualified class names and executable or lambda-target attributes. Check the packaged JAR used by the receiving process, rather than only the build output directory.

Generate configuration as part of a normal repeatable build. Apply generation separately to each dependency module that owns payload classes; the application generator does not scan dependency JARs. A manually authored security resource must be merged into the artifact deliberately so resource copying does not overwrite generated entries.

Build and deploy the producing and receiving artifacts, restart the relevant processes, and exercise a legitimate operation. In an isolated test deployment, send an unclassified application type and confirm rejection before trusting the configured boundary. Existing compatibility traffic alone does not prove that hardened decoding will succeed.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Classifying Remote Executable Objects

Declare trusted remote behavior explicitly and verify the operation-specific installation policy on the receiving service.
Remote executable objects are application classes whose methods Coherence may invoke on behalf of a remote request.

What Counts As An Executable

- Processors — `InvocableMap.EntryProcessor` implementations that run against cache entries.
- Aggregators — `InvocableMap.EntryAggregator` implementations that compute server-side aggregate results.
- Invocables — `Invocable` implementations submitted to the `InvocationService`.
- Filters — `Filter` implementations evaluated against cache, query, topic, REST, or gRPC data.
- Value extractors — `ValueExtractor` implementations that read values during server-side filtering, sorting, indexing, or projection.
- Comparators — `Comparator` implementations used for server-side ordering.
- Triggers — `MapTrigger` implementations installed to run during cache mutation.
- Server-side event interceptors — `EventInterceptor` implementations that run in the cluster when matching events fire.
- `coherence-concurrent` tasks / callables / runnables — executable work submitted through the Coherence concurrent APIs.

Client-installed `MapListeners` are routed, not invoked, on the server; they do not require `@Remote.Executable`. A listener that additionally implements `MapTrigger` is governed by the `MapTrigger` rule.

The Annotation

```
@Remote.Executable
public class PriceAdjustmentProcessor
        implements InvocableMap.EntryProcessor<String, Order, Void> { ... }
```

`@Remote.Executable` has CLASS retention and is class-level only. It implies `@Remote.Allowed`: a class that is executable is also a class the receiver may deserialize. The build-time generator emits a single entry in `security-config.xml` with `source="@Remote.Executable"` and `executable="true"`. At runtime, the security-config reader adds that class to `executableFqns()` as well as `allowedFqns()`.

XML Alternative

You can also author the same classification by hand in `META-INF/coherence/security-config.xml`:

```
<class name="com.example.PriceAdjustmentProcessor"
       source="@Remote.Executable"
       executable="true"/>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

Runtime merge semantics are the same as generated allowlist entries: Coherence unions every visible classpath source into one view. See Generated `security-config.xml` for the full merge story.

InvocationService

When security hardening disables the Coherence*Extend `InvocationService` proxy by default, operators who intentionally expose remote invocation can set `coherence.invocation.enabled=true` or configure `<proxy-config><invocation-service-proxy><enabled>true</enabled></invocation-service-proxy></proxy-config>`. The system property overrides operational configuration when both are present.

When the proxy is enabled, every remote `Invocable` submitted through Extend is checked by `RemoteExecutablePolicy` before `InvocationService.query(...)` executes it and before any delegated `PriorityTask` scheduling, timeout, or cancellation callback can run. The invocable class must carry `@Remote.Executable` or appear with `executable="true"` in merged `security-config.xml`. Cluster-internal `Invocable` dispatch is not gated by the Extend proxy flag.

If the proxy is disabled, the server log records `InvocationService proxy is DISABLED in <mode> mode by policy` once at proxy startup, while the client receives the same generic service-unavailable shape used for an unbound Extend receiver. The wire response intentionally does not name the policy decision. See the Coherence security guide section "Securing Extend Client Connections" for Extend authentication setup.

MethodInvocationProcessor / ScriptProcessor

`MethodInvocationProcessor` and `ScriptProcessor` are built-in `EntryProcessor` implementations that execute behavior described by their serialized state. Both classes are `final`, annotated with `@Remote.Executable`, and checked by `RemoteExecutablePolicy` before their dynamic-remote gates run.

The shared `coherence.remote.dynamic.unauthenticated` policy controls unauthenticated DYNAMIC lambda bytecode, reflective method invocation through `MethodInvocationProcessor`, and script evaluation through `ScriptProcessor`. The current integration default is `allow` in both profiles. Set `deny` explicitly when those paths are not approved. The hardened profile alone does not refuse them.

`MethodInvocationProcessor` also applies the lambda bytecode deny-list before reflection. The supplier class, the runtime target class, and the `targetClass#methodName` pair are each checked against `META-INF/coherence/lambda-bytecode-denylist.txt` and extension resources such as `META-INF/coherence/lambda-bytecode-denylist.d/*.txt`. Entries such as `java.lang.Runtime` and `java.lang.System#exit` are hard floors in every security mode. When a processor targets an absent cache entry, the supplier class is checked first, but the supplier is not executed and the entry is not populated until the dynamic-remote mode gate allows MIP.

`ScriptProcessor` runs scripts in the singleton Graal `Context` with explicit host access. `allowAllAccess(true)` is not enabled. Host class lookup rejects deny-listed classes, and loadable deny-listed classes are also blocked through Graal `HostAccess.denyAccess(...)`. These HostAccess checks are hard floors in every mode.

MapTrigger Remote Install and Removal

Remote `MapTrigger` installation and removal through Coherence*Extend are checked before the listener is registered with or removed from the storage tier. Named trigger classes must carry `@Remote.Executable` or appear with `executable="true"` in merged `META-INF/`

Copyright © 2008, 2026, Oracle and/or its affiliates.

`coherence/security-config.xml`. Missing classification rejects named classes in hardened mode and records compatibility shadow telemetry.

Dynamic trigger classes use the shared dynamic-install gate. The current implementation allows a dynamic class through that gate even when the preceding executable-class check lacks a named-class entry. An explicit `coherence.remote.dynamic.unauthenticated=deny` rejects the install in either profile. With the current `allow` default, missing executable classification alone is not a refusal for this dynamic route; serialization, lambda-target and hard bytecode checks still apply.

Classes declared in cache configuration are trusted server configuration. Coherence logs a WARN for each unannotated declared MapTrigger or EventInterceptor class, deduplicated by class name and kind, so operators can classify those classes before allowing equivalent remote installs. Event-interceptor coverage described here is a startup advisory; it does not imply a remote interceptor installation API.

Topic Subscriber Filter / Extractor Install

Remote topic subscriber group and subscriber creation through Coherence*Extend and gRPC are checked before metadata is installed on the storage tier. Subscriber filters use `OperationReason.EVALUATE_FILTER`; extractors and converters use `OperationReason.EXTRACT`; both publish `role=TOPICS` in telemetry. Named filter and extractor classes require `@Remote.Executable` or `executable="true"`; missing classification rejects in hardened mode and records compatibility shadow telemetry.

Coherence also walks shipped topic subscriber containers that can hide nested executable state before install and again during persisted replay. This includes filter wrappers such as `ArrayFilter`, `ExtractorFilter`, `LimitFilter`, and `FilterWrapper`; extractor wrappers such as `ConditionalExtractor`, `CollectionExtractor`, `KeyExtractor`, fragment extractors, and transaction wrappers; comparator wrappers such as `ExtractorComparator`, `ChainedComparator`, and `SafeComparator`; and script-backed subscriber filters and extractors. Each nested class is checked with topic subscriber telemetry and replay-drift handling. Customer-defined composites with opaque fields are not inspected reflectively; annotate or allowlist the composite class and any nested classes exposed through the built-in containers.

Dynamic or lambda-shaped subscriber filter and extractor classes use the same dynamic-install exception and shared policy as remote dynamic triggers. The current integration default permits them in either profile; set `coherence.remote.dynamic.unauthenticated=deny` explicitly to refuse their remote installation. Persisted replay has a separate class-policy drift check and does not inherit this install-time exception.

Persisted subscriber metadata is rechecked when topic subscription metadata is activated from persistence. The `coherence.topics.persisted.policy-drift` property controls how Coherence handles metadata whose class was valid at install time but is no longer executable at replay time:

- `reject` refuses the replayed subscription and emits `sub_reason=replay_drift`.
- `warn-allow` emits a deduped WARN for the drifted class and allows replay.

The default is `reject` in hardened security mode and `warn-allow` in compatibility security mode. An explicit `reject` value fails closed in both security modes. Operators should use `warn-allow` only as a migration bridge while annotating the affected filter/extractor classes, regenerating `security-config.xml`, and redeploying the owning artifacts.

Cache-config-declared topic subscribers do not emit the same startup advisory. Verify their declared filters and extractors directly; remote installs and persisted replay rechecks still apply as described above.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Cache Data-Plane Processor / Aggregator / Filter / Extractor / Comparator Install

Remote `NamedCache` data-plane operations through Coherence*Extend, REST, and gRPC are checked after the wire payload or registry alias resolves to a concrete executable object and before dispatch to the storage tier. Entry processors use `OperationReason.PROCESS_ENTRY`; aggregators use `AGGREGATE`; filters use `EVALUATE_FILTER`; value extractors use `EXTRACT`; comparators use `COMPARE`. Telemetry publishes `role=EXTEND_PROXY`, `role=REST`, or `role=GRPC` depending on the transport.

Named processor, aggregator, filter, extractor, and comparator classes require `@Remote.Executable` or `executable="true"` in merged security configuration. Missing classification rejects named classes in hardened mode and records compatibility shadow telemetry. Dynamic and lambda-shaped installs use the dynamic-install exception described above. Script-backed executables also use the shared dynamic gate; `deny` explicitly refuses these paths in either profile. The script gate covers `ScriptProcessor`, `ScriptFilter`, `ScriptValueExtractor`, and `ScriptAggregator`, at remote install where exposed and directly before execution or delegate creation.

Coherence also walks the built-in composite containers that can hide nested executables. This includes processor wrappers such as `CompositeProcessor`, `ConditionalProcessor`, asynchronous / priority processor wrappers, `ExtractorProcessor`, `PropertyProcessor`, and `UpdaterProcessor`; aggregator wrappers such as `CompositeAggregator`, `GroupAggregator`, `ReducerAggregator`, asynchronous / priority aggregators, and `TopNAggregator`; filter wrappers such as `ArrayFilter`, `ExtractorFilter`, `LimitFilter`, `MapEventTransformerFilter`, `WrapperQueryRecorderFilter`, and transaction `FilterWrapper`; extractor wrappers such as `ConditionalExtractor`, `CollectionExtractor`, `DeserializationAccelerator`, transaction `ExtractorWrapper`, and the fragment extractors; and comparator wrappers such as `ExtractorComparator`, `ChainedComparator`, and `SafeComparator`. Comparator payloads that are also `ValueExtractor` instances, such as `KeyExtractor`, `ChainedExtractor`, `ConditionalExtractor`, `CollectionExtractor`, and other `AbstractExtractor` subclasses, are routed through the extractor cascade as well as the comparator gate; this also applies when they are wrapped by `SafeComparator`, `InverseComparator`, `EntryComparator`, or transaction `ComparatorWrapper`. `ValueManipulator` and `ValueUpdater` instances nested inside property and updater processors are treated as executable mutation state under the processor install check. Each cascaded class records its own policy telemetry event. Customer-defined composites with opaque fields are not inspected reflectively; annotate or allowlist the composite class and any nested classes that are exposed through the built-in containers above.

REST default processor and aggregator factories preflight the configured class before invoking its constructor. REST resources still check the produced processor or aggregator instance before dispatch so nested cascade checks are not lost. Custom REST `ProcessorFactory` and `AggregatorFactory` implementations remain a trusted compatibility boundary: Coherence cannot preflight arbitrary factory side effects before the custom factory chooses to construct or return an executable object.

To migrate an application, identify each rejected or would-reject class, annotate the supported remote executable type with `@Remote.Executable` or add an `executable="true"` entry to generated security config, redeploy the owning jar, and rerun the workload with `-Dcoherence.security.mode=hardened`. For REST registry entries, classify the registry factory output. For CohQL queries and gRPC bytes-based requests, classify the concrete filter, comparator, extractor, aggregator, or processor class that is sent by the client.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Coherence Concurrent Executor

Coherence Concurrent executor submissions are checked before task execution, scheduling, executor selection, result collection, completion callbacks, and subscriber callbacks. Named `Task`, `Callable`, `Runnable`, `ExecutionStrategy`, `Task.Collector`, completion predicate, completion runnable, and `Task.Subscriber` classes must carry `@Remote.Executable` or appear with `executable="true"` in merged `META-INF/coherence/security-config.xml`. Dynamic concurrent task and callback classes use the shared dynamic-install exception and policy described above. Enforcement telemetry uses `OperationReason.CONCURRENT_TASK` and `role=CONCURRENT`.

The concurrent gate also walks shipped executor wrappers that can carry nested callback state. `CallableTask`, `RunnableTask`, `RunnableWithResultTask`, scheduled task wrappers, and `CronTask` gate their nested callable, runnable, or task state after the task-manager subject is available. `CronTask` gates both the original and current task. `StandardExecutionStrategy` gates its candidate predicate, and `ConditionalCollector` gates its predicate and nested collector. Concurrent cache-processor wrappers such as `LocalOnlyProcessor` and `ClusteredTaskManager.ChainedProcessor` gate their nested processors before they can invoke them. Built-in predicate wrappers such as `Predicates.not(...)` and typed composites such as `Predicates.and(...)` are unwrapped and their nested predicates are checked before use. Value-style predicate wrappers gate nested payloads that could otherwise dispatch application code through `equals(...)` or `toString()`: unsafe `Predicates.equalTo(...)` values, non-built-in `Predicates.has(...)` registration options, and non-JDK `Predicates.onException(...)` throwables are checked before the predicate can run. Common scalar values, built-in registration options, exact product `Member` registration metadata, and bootstrap `java.*` throwables remain data-only. Crafted or non-product nested members inside `Predicates.has(Member.of(...))` are checked before comparison. `ValueTask` is gated as a task, but its value is result data and is not treated as an executable callback merely because it implements a callback interface. Likewise, `RunnableWithResultTask` gates its nested runnable, but its configured result is returned as data and is not executed by remote execution policy. Deserialization safety for either result value belongs to the serialization policy. Task properties, task options, execution plans, retained results, and tracing context are data-only for this gate.

During a rolling upgrade or partition transfer, a member running with `coherence.security.mode=hardened` can reject a replicated concurrent task entry that was installed before that member joined if its executable classes would not pass the member's current policy. Before enabling hardened security mode on members that may own concurrent task partitions, deploy consistent classification for tasks and every nested strategy, collector, predicate, and completion callback across the cluster. Keep security mode in compatibility posture during that preparation. The `coherence.mode` runtime/license setting does not select this gate.

Local member submissions capture the current caller `Subject` as advisory gate context and serialize it with `ClusteredTaskManager` at POF index `20` and as a final appended ExternalizableLite field. Remote-client submissions through remote cache services, including `RemoteExecutor` client paths, serialize a `null` subject; client JVM subjects are not authorization-bearing. The subject is not used with `Subject.doAs(...)` and does not grant per-subject execution permission.

The `ConcurrentProxy` Extend service uses the existing `coherence.concurrent.extend.enabled` property with a security-mode-aware default: enabled in compatibility security mode and disabled in hardened security mode. A disabled proxy logs the resolved runtime mode, security mode, property/config source, and enabled state at startup, plus a startup WARN in hardened mode naming the override property. Clients still see

Copyright © 2008, 2026, Oracle and/or its affiliates.

a generic service-unavailable shape rather than policy-specific wire text. Cluster-internal executor use is independent of this proxy switch.

Functional-Interface Lambda Targets

A lambda target is the functional interface named by a serialized lambda. For example, a lambda sent as a `Remote.Function` declares `com.tangosol.util.function.Remote$Function` as its functional-interface target; the generated lambda class is a separate implementation detail.

Coherence pre-checks that functional interface before it runs the lambda bytecode scanner. The target check answers "may this receiver materialize a lambda for this API surface?" with a constant-time lookup in merged `security-config.xml`. The bytecode scanner still answers the separate question "does this wire-supplied bytecode contain denied references or unsupported shape?" Both checks apply; a target allowlist entry does not disable bytecode scanning.

To allow a custom serializable functional interface as a remote lambda target, annotate the interface with `@Remote.Executable` and ensure it also carries `@FunctionalInterface`:

```
@FunctionalInterface
@Remote.Executable
public interface PriceFunction
        extends java.io.Serializable
    {
    BigDecimal apply(Order order);
    }
```

The build-time scanner emits:

```
<class name="com.example.PriceFunction"
       source="@Remote.Executable"
       lambda-target="true"/>
```

You can also author that XML entry by hand. In that case the entry must name the functional interface, not the generated lambda implementation class.

Unannotated functional interfaces are denied in hardened security mode before lambda bytecode is materialized. In compatibility security mode, allowlist and lambda-target enforcement are disabled; use `-Dcoherence.security.mode=hardened` in pre-production to validate lambda-target readiness before migration. The supported migration is to annotate the functional interface or add a manual `lambda-target="true"` entry to the artifact that owns it.

When the target pre-check rejects a lambda, the reason is `REASON_LAMBDA_TARGET_NOT_ALLOWED` and the operator-visible reason string is `lambda-target-not-allowed`. Pre-GA adopters who previously relied on custom serializable functional interfaces should rebuild those artifacts with `@Remote.Executable`, verify the generated `security-config.xml`, and redeploy receivers before sending lambdas that target the new interface.

Migration

`@Remote.Executable` covers two distinct classifications:

- Functional interfaces annotated with `@Remote.Executable` are enforced today when security mode is hardened. Lambdas that target an unannotated functional interface are rejected with `lambda-target-not-allowed` before bytecode is materialized. Migrate by

Copyright © 2008, 2026, Oracle and/or its affiliates.

annotating every custom serializable functional interface used as a remote lambda target and rebuilding the artifact that owns it. See Functional-Interface Lambda Targets.

- Concrete executable classes are classified by the build. Runtime enforcement covers the remote sinks described above, with the dynamic-install exception and startup-only interceptor advisory already noted. Regenerate `security-config.xml`, review executable shadow telemetry, and test the actual named and dynamic paths under the intended profile and explicit dynamic policy.

In both cases the build pipeline is the same. Rebuild every artifact that owns an executable class or functional interface; the `security-config-maven-plugin` and the Gradle `securityConfig` task pick up the annotations automatically.

Why `@Remote.Executable` And Not Just `@Remote.Allowed`?

`@Remote.Allowed` answers "can this class be deserialised on the server?"; `@Remote.Executable` answers "may the server invoke its methods on behalf of a remote request?" The latter is a strict subset and deserves an explicit opt-in so that adding a class to the deserialisation allowlist does not silently grant it execution privileges.

Classification and Caller Authorization

Executable classification applies to the code being installed or invoked. A storage authorizer applies the caller policy to the resource and operation. Configure both when the application needs both guarantees. A successful class-policy check does not assign a caller a role.

Continuous Aggregation registration, removal, and full-result queries use aggregate authorization and the corresponding remote-install policy. Partition projections use the entry-processor path. Where that feature is available, classify custom filters, extractors, aggregators, and projection functions just as for their on-demand equivalents.

Choosing a Lambda Serialization Policy

Choose how remote lambdas are represented and account for captured state, executable targets, and mixed-version deployment.
Coherence supports dynamic and static lambda serialization. Dynamic serialization is the shipped default and carries generated bytecode to the receiver, avoiding a requirement to deploy the same compiled lambda implementation everywhere first. Static serialization resolves a compiled lambda representation already available to the receiver. Both forms remain subject to class, executable-target, and captured-state validation.

Policy	Benefit	Deployment requirement
Dynamic, compatibility profile	Retains the shipped dynamic representation and compatible defaults.	Observe available diagnostics and test under hardened policy before rollout.
Dynamic, hardened profile	Retains dynamic deployment flexibility. The current integration default is `allow`; use an explicit `deny` when dynamic remote code is outside the deployment policy.	The default unauthenticated dynamic policy is deny; an explicit allow is a deliberate expansion of executable input.
Static	Avoids delivering a generated lambda class definition over the wire.	Make matching compiled application lambda classes available to all senders and receivers before their traffic crosses versions.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Dynamic Remote Policy

The current implementation defaults `coherence.remote.dynamic.unauthenticated` to `allow` in both profiles during hardened-default integration. Set `deny` explicitly to refuse unauthenticated dynamic lambdas, reflective `MethodInvocationProcessor`, and script execution through the shared dynamic gate. An explicit `allow` permits those paths together; it does not bypass their independent validation. Do not infer dynamic default-deny from the hardened profile.

```
-Dcoherence.security.mode=hardened
-Dcoherence.remote.dynamic.unauthenticated=allow
```

The setting does not disable bytecode structure checks, forbidden class/method references, functional-interface target validation, script host-access restrictions, or storage authorization. It does not add a subject-aware permission system to dynamic code. Limit who can connect and submit requests before allowing this behavior.

Static Lambda Deployment

```
-Dcoherence.lambdas=static
```

Deploy compatible compiled application classes to every participating sender and receiver. Test rolling upgrades with the exact application versions and captured types involved. Changing representation is a deployment change; keep existing persisted subscriber metadata and older clients in the test matrix.

Captured State and Functional Targets

A captured helper, value object, or nested executable is part of the serialized object graph. Register the relevant application classes, and classify custom functional-interface targets with `lambda-target="true"` where required by the generated/manual security configuration. Avoid capturing an application service, credential source, or large object graph simply for convenience.

A client-side listener that is delivered events does not become a remote server executable merely because it is a listener. Server-side filters, extractors, triggers, and task callbacks have their own installation rules. Use the route descriptions in Classifying Remote Executable Objects to identify the boundary actually being exercised.

Copyright © 2008, 2026, Oracle and/or its affiliates.

8

Limiting Resource Use and Protecting Persisted Data

Bound supported serialization containers and preserve explicit trust boundaries for transactions, persistence metadata, stored values, and native indexes.
Resource limits and type validation protect different aspects of input handling. A well-typed container can still be too large, and a small object can still be an unexpected type. Apply both policies at their documented consumers. Limit storage-member heap and admission capacity using the deployment and service controls appropriate to the workload.

- Use Configuring Serialization Container Limits for generic serializer policy and POF consumers.
- Use Transaction Bridges and Persistence Metadata before upgrading transaction or persistence workflows.
- Use Protecting Memcached Values and Native Indexes for optional protocol and native-library boundaries.

Persisted Data Is Still Input

Restrict writes to persistence, snapshot, archive, and index directories to trusted processes. Restrict import and recovery operations. Keep backups of both application data and the configuration/classes required to interpret it. Restore in an isolated, appropriately patched environment before accepting production traffic.

Structural validation, class filters, and checksums are not keyed authentication of a snapshot or journal. Cryptographic snapshot/journal integrity and key lifecycle require a separately supported format. Do not infer tamper-proof persistence from the metadata parsing controls described here.

- Configuring Serialization Container Limits
- Transaction Bridges and Persistence Metadata
- Protecting Memcached Values and Native Indexes

Configuring Serialization Container Limits

Set serializer-owned byte, element, and map-entry limits and verify the policy on the service that decodes the payload.
Serialization container limits are owned by the serializer. POF consumes this policy; do not assume that every third-party serializer or every object-graph path enforces all three limits. Structural checks reject negative, impossible, truncated, and overflow-prone input independently of the security profile.

Limit	Hardened default	Property
Container bytes	64 MiB, or 67108864 bytes	`coherence.serializer.limit.maxContainerBytes`
Array, collection, and object elements	262144	`coherence.serializer.limit.maxElements`

Copyright © 2008, 2026, Oracle and/or its affiliates.

Limit	Hardened default	Property
Map entries	131072	`coherence.serializer.limit.maxMapEntries`

Defaults and explicit policy

The finite hardened defaults above apply starting with Coherence 26.1.0.0 when `coherence.security.mode` is unset. An explicitly selected compatibility profile permits default-unbounded behavior and warns when applicable shadow thresholds are exceeded. Hardened mode requires finite limits and rejects `unlimited`. A finite limit explicitly configured in compatibility mode still constrains its consumer.

Specify bytes using a supported size suffix, such as `64m`, or a count of bytes. Element and map-entry values are counts. Choose limits from measured legitimate payloads. A per-container limit does not bound the total object graph, concurrent requests, or member heap.

A service-local named serializer override

Place the following serializer element in the cache service scheme that owns the decode path. The `id` refers to an operational serializer template. Each service receives its configured serializer policy; changing one service should not mutate the template used by other services.

```
<serializer id="pof">
  <limits>
    <max-container-bytes system-
property="billing.serialization.maxBytes">64m</max-container-bytes>
    <max-elements system-property="billing.serialization.maxElements">262144</
max-elements>
    <max-map-entries system-
property="billing.serialization.maxMapEntries">131072</max-map-entries>
  </limits>
</serializer>
```

A limit element with `system-property` uses that property's value when set and its element body as the fallback. Missing values inherit. Global property-backed defaults form the base; named/global serializer configuration, service-local overrides, and explicitly configured serializer instances refine it. The existing text reference `<serializer>pof</serializer>` remains valid when no local override is required.

Operational serializer declarations support `<limits>` after their class/instance configuration. Cache-config inline instances use `<serializer><instance>...</instance><limits>...</limits></serializer>`. Preserve the instance syntax used by the relevant schema. Custom serializers must expose and consume the appropriate policy; configuration alone does not establish that a custom parser honors every bound.

Verification and rollout

1. Test the largest legitimate array, collection, map, and sparse structure on the actual serializer and service.
2. In an isolated deployment, submit a payload above each bound. Verify controlled rejection before excessive allocation or partial installation.
3. Repeat with property and service-local overrides to establish the effective configuration.

Copyright © 2008, 2026, Oracle and/or its affiliates.

4. Coordinate producer/consumer limits, including replay and recovery readers. Increase a legitimate limit to a measured finite value rather than disabling policy.

Transaction Bridges and Persistence Metadata

Account for typed metadata, bridge-local type restrictions, and legacy payload behavior when transaction and persistence data cross an upgrade boundary.
Coherence distinguishes narrow product-owned Java serialization bridges from application data carried by the configured Coherence serializer. There is no OIS-specific escape-hatch property. When a supported application class is missing, update generated `security-config.xml` or `coherence.serialization.allowed` after security review. When a narrow bridge rejects a legitimate product shape, the remedy is a reviewed product allowlist or bridge-filter update, not a customer property that disables the bridge check.

Transaction result and XID bridges

The retained Java serialization byte fields in transaction `Results`, JCache entry-processor exceptions, and POF `XidWrapper` compose a field-specific expected-type filter with the central serialization hardening filter and any JVM filter before the nested object stream materializes a class. The `Results` `ExternalizableLite` exception field supports only exact classes in the runtime-exception bridge policy throughout the complete root, cause, and suppressed graph. Before emitting the exception tag, its ordinary writer uses a cycle-safe identity walk to validate every throwable, then emits null or a fixed Java-serialization tag. Its reader validates that tag before dispatch and filters the nested stream, including the narrow JDK list/array infrastructure required by ordinary suppressed exceptions. Every array admitted by the transaction and JCache exception bridges has a maximum length of 16,384. The reader rejects a larger exact `Object[]`, stack-trace array, allowed-exception array, or primitive array before allocation. The `Results` writer applies the same maximum to each throwable's suppressed-exception and stack-trace arrays before emitting the exception tag. Applications with larger exception state must trim it before transmission. Unsupported custom exception classes anywhere in the graph, including `ExternalizableLite` or `Externalizable` runtime exceptions, fail on write before serialization callbacks instead of producing bytes the reader cannot consume. These explicit bridge checks apply in both security profiles, including when the profile property is unset. Applications that previously propagated a custom runtime exception through a transaction result must translate it to a supported JDK or Coherence transaction exception before transmission. An application serialization allowlist does not widen this exact product bridge.

The shipped transaction POF configuration serializes `Results` through its direct `PortableObject` implementation and retains POF fields 1 through 5. Configuring `ExternalizableLitePofSerializer` for `Results` is unsupported: that adapter exposes the exception property only through `PofInputStream.readObject`, which cannot enforce the exact bridge policy before POF type construction and callbacks. `Results` rejects that adapter on read and write before any property dispatch. Use the shipped direct `PortableObject` registration; there is no property that bypasses this check.

The XidWrapper ExternalizableLite/tagged DataInput field retains a broader generic serialization boundary than the tightened POF bridge. Its compatible format can carry ExternalizableXid or a custom serializable Xid. Compatibility does not automatically attach the central filter to an otherwise unfiltered BufferInput; hardened policy attaches it before tagged class dispatch. This path does not provide the exact XID-only field policy of the POF bridge. Keep the general application allowlist and any configured input filter appropriately constrained.

New POF XidWrapper writers canonicalize the retained field to ExternalizableXid, and tightened POF readers reject arbitrary legacy custom serializable Xid implementations. Before a mixed-version XA upgrade can deliver such legacy POF bytes to a new reader, drain or

Copyright © 2008, 2026, Oracle and/or its affiliates.

complete in-flight and prepared XA work that depends on custom serializable XID implementations. Apply this drain guidance to the tightened POF bridge; the tagged DataInput field retains the distinct boundary described above.

Transaction bindings during mixed-version operation

New transaction `Message` writers emit both the validated legacy POF binding map at field 2 and typed binding fields 4 through 9. New readers use only the typed fields and do not materialize field 2. Consequently, a legacy-only sender that supplies only POF field 2 is accepted by a new receiver with an empty binding set. Coordinate binding-dependent transactional traffic during rolling upgrade; do not assume a legacy-only binding map will be interpreted by a new reader. The tagged `DataInput` binding path uses the typed scalar codec directly.

Results application maps

The three Results maps carry application data: cache keys and values, query results, entry-processor results, aggregator results, and application natural keys used by session mutation metadata. They retain those application semantics during mixed-version operation. Configure their serializer and application type policy rather than expecting the narrow product-exception bridge to restrict ordinary cache data.

For tagged `DataInput`, nested Java serialization values retain the serialization hardening object-stream filter in both security profiles. Hardened security mode also attaches the central filter before general Coherence tagged class dispatch; compatibility mode preserves the prior generic application materialization behavior unless the input already has an operator/JVM filter. POF map values remain governed by the configured POF context and serializer policy. These maps are application serializer data, not narrow OIS bridge fields.

Memcached-origin stored values

Memcached binary pass-through writes and legacy `DECO_MEMCACHED` values are validated before storage or ordinary Coherence materialization. Dangerous terminal formats, including Java serialization, XML serialization, `ExternalizableLite`, unknown, and XML-bean shapes, reject before dispatch, including through supported binary, identity, or optional wrappers. `FMT_EXT` interop remains available only when materialization uses a proven non-default configured serializer. This stored-value guard is a hard boundary in both security profiles; ordinary Memcached scalar/binary behavior and supported POF interop remain available.

`coherence.mode` remains the runtime/license setting and does not select any of these object-stream bridge hardening behaviors. Use `coherence.security.mode=compatibility` or `hardened` only for the central profile differences described above.

Persistence Metadata and Application Data

Product metadata has a narrower type universe than application cache data. Partition counts and persistence versions use fixed parsing; quorum compatibility data retains a guarded reader, and snapshot/store names are validated before use. Arbitrary application keys, values, and result maps continue to use the selected application serializer and its configured class/size policy.

Do not replace the shipped direct `PortableObject` registration for transaction results with `ExternalizableLitePofSerializer`. That adapter is unsupported for the tightened result field. Do not assume an application allowlist widens a bridge-local throwable/XID type universe or raises its fixed array bound.

Before rollout, drain or explicitly test XA work using custom serializable XIDs and transaction bindings produced by older writers. Retain a recoverable copy of old persisted data. A rollback of security profile does not undo mode-independent bridge changes.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Protecting Memcached Values and Native Indexes

Treat optional Memcached listeners and HNSW index files as separate trust and resource boundaries.

Memcached-origin stored values

Memcached binary pass-through and legacy `DECO_MEMCACHED` values are validated before storage or ordinary Coherence materialization. Dangerous terminal Java, XML, ExternalizableLite, unknown, and XML-bean shapes reject, including through supported wrappers. Scalar/binary and supported POF interoperability remain available; `FMT_EXT` requires a proven non-default configured serializer.

These checks apply in both security profiles. They do not add authentication or encryption to a Memcached listener. Inventory that listener separately, restrict its network reachability, and expose it only where the protocol security capabilities meet the deployment's requirements. Extend identity configuration does not secure another protocol automatically.

HNSW input and file validation

The Java/JNI boundary validates dimensions, vector lengths, query counts, graph parameters, and signed-to-unsigned conversions before native work. Native index loading checks sizes, allocation arithmetic, link metadata, and neighbor identities. A failed load must not be treated as a usable newly loaded index.

Use the native library distributed with the corresponding patched Java module on every target platform. Replacing a Java JAR does not replace an older separately installed native binary. Keep index files in a protected directory and test load/rebuild procedures with the supported binary.

Invalid dimensions, oversized counts, and corrupt files should produce controlled errors. Preserve the original index before retrying or rebuilding. These checks do not cryptographically authenticate index files and do not establish bounds fixes in separate C++ or .NET clients.

Copyright © 2008, 2026, Oracle and/or its affiliates.

9

Securing gRPC Client Connections

Require the intended transport, authentication, serializer, and diagnostic policies on the gRPC proxy and its clients.
A gRPC connection has independent transport, identity, and operation-policy choices. Anonymous compatibility remains available unless authentication is configured. A global identity asserter alone does not make a gRPC acceptor require authentication. Configure the acceptor's authentication method explicitly and make its asserter return a verified non-null subject.

Control	Setting	Verification
Server TLS policy	`coherence.grpc.server.secure-transport=required` or acceptor XML	Cleartext and insecure providers cannot satisfy the policy.
Java client TLS policy	`coherence.grpc.channels.<name>.secure-transport=required` or channel XML	The client trusts the intended issuer and verifies the server name.
Authentication	`coherence.grpc.auth=basic` or `<auth-method>basic</auth-method>`	Invalid credentials fail and valid credentials establish the expected subject.
Operation authorization	Configured service/storage authorizer	An authenticated but unauthorized operation is denied.

Use Configuring a Secure gRPC Proxy with Connecting a Java gRPC Client Securely. Serializer names and diagnostic exposure are covered in gRPC Serializer and Diagnostic Policy.

Client scope

Use a released Coherence client and supported transport implementation. Before adapting the Java recipe to Go, Python, JavaScript, or another language, verify that client's trust-material, credential, hostname/authority, and mTLS APIs against the intended server. Secure-required servers should continue rejecting insecure clients while client adoption work is completed.

- Configuring a Secure gRPC Proxy
- Connecting a Java gRPC Client Securely
- gRPC Serializer and Diagnostic Policy

Configuring a Secure gRPC Proxy

Configure an SSL socket provider, explicit authentication, and required transport on a gRPC acceptor.
Define `server-tls` in the member operational override using Configuring TLS with Protected Password Sources. Server-authenticated TLS requires a server identity. mTLS also requires trust and client certificates. Provision the certificate for the actual DNS name used by clients.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Place this acceptor fragment in the proxy scheme in cache configuration. It requires the Coherence gRPC proxy module and an identity asserter accepting `UsernameAndPassword`, such as Authenticating Credentials with JAAS.

```
<acceptor-config>
  <grpc-acceptor>
    <local-address>
      <address>0.0.0.0</address>
      <port>1408</port>
    </local-address>
    <socket-provider>server-tls</socket-provider>
    <auth-method>basic</auth-method>
    <channelz>disabled</channelz>
    <error-disclosure>safe</error-disclosure>
    <secure-transport>required</secure-transport>
  </grpc-acceptor>
</acceptor-config>
```

The wildcard bind is appropriate only with the required network controls; use a specific interface when possible. The recipe pairs Basic authentication with required TLS. Basic alone does not impose a startup-time TLS mandate. Hardened policy rejects Basic requests over cleartext; compatibility permits them with a warning.

Subject propagation

The proxy preserves authenticated subjects through supported v0 direct/async and v1 subchannel/cache-protocol paths for downstream authorization. Configure the operational identity asserter and subject scoping appropriate to the application, and wire a storage authorizer to every scheme needing operation checks.

Verification

1. Connect with the configured CA and matching server name, authenticate, and complete an authorized operation.
2. Test an untrusted CA, a name mismatch, and a cleartext channel. Each must fail at the relevant transport boundary.
3. Try missing and invalid Basic credentials. Expect `UNAUTHENTICATED`.
4. With valid credentials for a denied role, try the protected operation. Expect `PERMISSION_DENIED`.
5. Confirm Channelz is unavailable and safe errors omit internal stacks while retaining bounded status information.

Connecting a Java gRPC Client Securely

Configure the product Java client with explicit trust, required transport, and credentials for the selected acceptor.
Define `client-tls` in the client operational override. Server-authenticated TLS needs trust material; mTLS also needs the client's private key and certificate. Connect using a name present in the server certificate. Map application cache names to `secure-grpc`, and add this scheme:

```
<remote-grpc-cache-scheme>
  <scheme-name>secure-grpc</scheme-name>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
  <service-name>SecureGrpcClient</service-name>
  <grpc-channel>
    <remote-addresses>
      <socket-address>
        <address>cache.example.com</address>
        <port>1408</port>
      </socket-address>
    </remote-addresses>
    <socket-provider>client-tls</socket-provider>
    <secure-transport>required</secure-transport>
    <configurer>
      <class-name>com.example.security.BasicGrpcConfigurer</class-name>
    </configurer>
  </grpc-channel>
</remote-grpc-cache-scheme>
```

The following complete configurer uses the product `GrpcChannelConfigurer` and gRPC metadata utilities. It requires Java 21 or later and the product gRPC client dependencies. Its password URL identifies an OS-protected secret file; the password itself is not a system-property value.

```
package com.example.security;

import com.oracle.coherence.grpc.client.common.GrpcChannelConfigurer;
import com.tangosol.net.PasswordProvider;
import com.tangosol.net.URLPasswordProvider;
import io.grpc.ManagedChannelBuilder;
import io.grpc.Metadata;
import io.grpc.stub.MetadataUtils;
import java.nio.charset.StandardCharsets;
import java.util.Base64;
import java.util.Objects;

public final class BasicGrpcConfigurer implements GrpcChannelConfigurer {
    @Override
    public void apply(ManagedChannelBuilder<?> builder) {
        var username = Objects.requireNonNull(
                System.getProperty("example.grpc.username"), "username");
        var url = Objects.requireNonNull(
                System.getProperty("example.grpc.password.url"), "password
URL");
        char[] password = new URLPasswordProvider(url, true).get();
        try {
            if (username.isBlank() || username.contains(":")
                    || password == null || password.length == 0) {
                throw new IllegalArgumentException("Missing or invalid
credentials");
            }
            var value = username + ":" + new String(password);
            var encoded = Base64.getEncoder().encodeToString(
                    value.getBytes(StandardCharsets.UTF_8));
            var metadata = new Metadata();
            metadata.put(Metadata.Key.of("authorization",
Metadata.ASCII_STRING_MARSHALLER),
                    "Basic " + encoded);
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
builder.intercept(MetadataUtils.newAttachHeadersInterceptor(metadata));
        } finally {
            PasswordProvider.reset(password);
        }
    }
}
```

Set `example.grpc.username` and `example.grpc.password.url` before channel creation. This minimal configurer reads credentials once. Its Basic header remains in memory for the channel lifetime; resetting the source array does not erase immutable strings. Recreate the channel after rotating this secret, or use an application integration designed for credential refresh. Never log the metadata or include it in a URL.

Bootstrap and session use

Select the operational override and client cache configuration before startup, and deploy the client module/configurer and application security resources. The application uses the standard bootstrap and typed session APIs:

```
package com.example.security;

import com.tangosol.net.Coherence;
import com.tangosol.net.NamedMap;

public final class SecureClient {
    public static void main(String[] args) {
        try (var coherence = Coherence.client().start().join()) {
            var session = coherence.getSession();
            NamedMap<String, String> accounts = session.getMap("accounts");
            System.out.println(accounts.get("account-1"));
        }
    }
}
```

Keep the instance and session for the application lifetime. A successful read proves that read path; separately test a denied write and a server-name mismatch. Do not use a permissive hostname verifier or authority override to hide a certificate error.

gRPC Serializer and Diagnostic Policy

Constrain client-selected serialization and choose explicit diagnostic exposure for the gRPC service.
Starting with Coherence 26.1.0.0, an unset `coherence.security.mode` selects the hardened column below. With their own settings left at `auto`, Channelz is disabled and error disclosure is safe. Select the compatibility profile explicitly to obtain its different defaults.

Setting	Compatibility profile (explicit)	Hardened profile (default starting with 26.1.0.0)
Empty serializer format; built-in `pof`, `json`, `genson`, Coherence `JsonSerializer` name	Accepted according to serializer configuration.	Accepted according to serializer configuration.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Setting	Compatibility profile (explicit)	Hardened profile (default starting with 26.1.0.0)
`java`, `DefaultSerializer`, arbitrary custom names	Would-reject telemetry outside the allowed set.	Require `coherence.grpc.serializer.allowlist`.
Channelz `auto`	Enabled.	Disabled.
Error disclosure `auto`	Diagnostic detail.	Safe disclosure.

An omitted/empty format selects the default. The literal string `default` is not in the built-in allowed-name set. A custom name must resolve to its configured implementation and be explicitly approved where required. Serializer-name approval does not authorize every materialized class or executable.

JSON `@class` and `@type` metadata is gated before materialization, including nested and statically typed gRPC paths. Configure supported aliases and application classes; an application allowlist does not bypass independent JSON metadata policy.

```
-Dcoherence.grpc.channelz=disabled
-Dcoherence.grpc.error-disclosure=safe
```

Explicit values override `auto`. Diagnostic mode can disclose internal messages, stack metadata, and serialized throwable details; restrict its use appropriately. Channelz policy does not remove normal health-service registration.

Treat a serializer `INVALID_ARGUMENT` as input/configuration failure, `UNAUTHENTICATED` as identity failure, and `PERMISSION_DENIED` as policy failure. Supported listener streams preserve permission-denied semantics for security errors. Use server-side bounded signals when safe disclosure omits cause details.

Copyright © 2008, 2026, Oracle and/or its affiliates.

10

Securing Oracle Coherence REST

Authenticate REST callers, authorize operations, and constrain published resources and request-controlled expressions.
Coherence REST security combines the HTTP acceptor's authentication and TLS configuration with service/storage authorization and request-input policy. Configure each layer explicitly. The following topics retain the HTTP authentication setup; use Constraining REST Resources and Expressions for hardened resource/query policy.

Basic, client-certificate, and combined certificate-plus-Basic authentication are supported by the HTTP acceptor. Use TLS for credentials and verify which subject/principal is passed to authorization. A configured authentication scheme with an unusable security context must not be treated as a successful anonymous login in the hardened profile.

Protect both reads and mutations. A published resource name, a query predicate, and an authenticated user each convey different information; none substitutes for an authorization decision on the requested cache operation. Configure a storage authorizer for protected cache schemes and test denied access with valid credentials.

Serializer boundaries

JSON class metadata and XML parsing are input boundaries even for authenticated requests. Use REST JSON and XML Input Policy and the application serialization configuration. Preserve the selected marshaller and application types across client upgrades.

- Constraining REST Resources and Expressions
- REST JSON and XML Input Policy
- Overview of Securing Oracle Coherence REST
- Using HTTP Basic Authentication with Oracle Coherence REST
- Using SSL Authentication With Oracle Coherence REST
- Using SSL and HTTP Basic Authentication with Oracle Coherence REST
- Implementing Authorization For Oracle Coherence REST

Constraining REST Resources and Expressions

Publish only intended cache resources and let callers select configured queries and expressions.
In the hardened profile, an unconfigured pass-through cache is not automatically published. Declare each public resource in `coherence-rest-config.xml`. An allowed resource still needs authentication and operation authorization where required.

This complete REST configuration illustrates a bounded resource and configured expression aliases. Replace the application value class with the deployed model and package its serializer/security resources. The attribute and child-element ordering follows the REST schema.

```
<rest xmlns="http://xmlns.oracle.com/coherence/coherence-rest-config">
  <resources>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
    <resource name="people" max-results="100">
      <cache-name>people</cache-name>
      <key-class>java.lang.Integer</key-class>
      <value-class>com.example.model.Person</value-class>
      <query engine="DEFAULT">
        <name>adults</name>
        <expression>age &gt;= 18</expression>
      </query>
      <direct-query enabled="false" max-results="100"/>
      <expression-aliases>
        <sort-alias>
          <name>by-age</name>
          <expression>age</expression>
        </sort-alias>
        <projection-alias>
          <name>summary</name>
          <properties>name,age</properties>
        </projection-alias>
        <aggregator-argument-alias>
          <aggregator>long-sum</aggregator>
          <name>total-age</name>
          <expression>age</expression>
        </aggregator-argument-alias>
        <processor-argument-alias>
          <processor>increment</processor>
          <name>age-value</name>
          <expression>age</expression>
        </processor-argument-alias>
      </expression-aliases>
    </resource>
  </resources>
</rest>
```

The aggregator and processor names must exist in the deployed REST registry. Aliases approve expressions, not arbitrary implementations or an operation for every user. Restrict state-changing processor endpoints independently.

Query and alias use

Prefer named queries for public APIs. To permit direct `q` queries, set `<direct-query enabled="true">` explicitly on the resource and choose a result bound. An empty `<direct-query/>` does not enable ordinary HTTP cache operations in either profile. Hardened mode requires enablement on the SSE path too and validates the supported REST-safe subset before evaluation. The legacy SSE compatibility path retains its bypass; do not depend on that behavior for a secured deployment.

Callers select configured aliases in URL sort/projection/aggregator/processor input. For example, a named-query request can select the approved sort with `/people/adults;sort=by-age`. Projection aliases use the projection input with the configured alias name. Test the exact URL, encoding, resource precedence, and processor/aggregator signature used by your released REST module. Compatibility callers can retain historical raw expressions; plan their migration before hardened activation.

Verification

- Verify a declared resource and approved named query succeed for an authorized user.

Copyright © 2008, 2026, Oracle and/or its affiliates.

- In hardened mode, request an unconfigured pass-through resource and expect 404.
- Under hardened policy, test both ordinary HTTP and SSE direct queries: disabled queries must fail with 403; an enabled supported predicate should succeed for an authorized caller, and an unsafe expression must fail with 400. Ordinary HTTP cache operations also enforce direct-query enablement in compatibility mode; the historical SSE compatibility path differs.
- Verify configured aliases succeed and unknown/raw restricted expressions fail with 400.
- Test a denied write with otherwise valid credentials. Resource/alias approval must not bypass operation authorization.

REST JSON and XML Input Policy

Resolve JSON types deliberately and retain XML external-entity protections on every parser path.
Hardened JSON metadata policy rejects class literals and arbitrary class-name fallback. Configure exact server-owned aliases for application types that require polymorphism. Package aliases do not satisfy the hardened `@class` metadata requirement. The resolved class must also meet the application class and executable policies for its eventual use.

For a deliberately configured serializer, the `JsonSerializer` builder modifier is an extension point for application aliases. The following factory fragment requires the Coherence JSON module; deploy the application class on the receiver and reference the factory from the intended serializer configuration.

```
var serializer = new com.oracle.coherence.io.json.JsonSerializer(
        Thread.currentThread().getContextClassLoader(),
        builder -> builder.addAlias("person", com.example.model.Person.class),
        false);
```

Test the actual REST marshaller and gRPC serializer path that consumes the configuration. A separately configured serializer does not reconfigure every JSON consumer. The gRPC JSON materialization path checks denied metadata in nested and statically typed values as well as untyped roots. Other paths have different contextual checks; do not assume that one gRPC check proves coverage for every REST marshaller. Verify unknown aliases, denied class metadata, and malformed input on the actual consuming path.

XML parser protections

Coherence applies external-entity and related parser protections before processing XML. If required protections cannot be applied, hardened policy fails closed. Compatibility can warn and retain its fallback behavior. Use a supported parser/provider that implements the required controls and resolve approved schemas/resources through the supported local mechanisms.

Keep product configuration resources and application XML inputs distinct. A controlled local schema source is not permission to fetch an arbitrary remote entity named by a request. A parser-capability failure requires fixing the parser/runtime configuration; do not globally enable external entities to make a request pass.

Failure diagnosis

Expect bounded class/alias or parser errors rather than serialized internal throwable graphs. Retain a safe request identifier and inspect server logs. Record the response status supplied by the actual marshaller/container, since not every JSON binding failure has the same transport mapping. See Security Errors and Diagnostic Signals.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Overview of Securing Oracle Coherence REST

Choose explicit REST authentication and authorization policies and account for the active security profile.
Configure HTTP Basic, client-certificate, or combined authentication on the REST HTTP acceptor. Choose a JAAS login module or trusted certificate-to-identity mapping that creates the principal required by the application's authorization policy. Authentication remains an explicit service choice; selecting the hardened profile does not automatically create every REST identity policy.

When REST authentication is engaged, hardened policy requires a usable principal. A null/blank principal or a failing security-context accessor produces an authentication failure rather than silently falling back to anonymous access. Compatibility retains the relevant historical behavior.

Explicitly publish allowed resources and configure query/expression aliases before switching to hardened behavior. Use Constraining REST Resources and Expressions. Storage-side authorization complements any Extend-style wrappers used by the REST service; see Authorizing Access to Server-Side Operations.

Using HTTP Basic Authentication with Oracle Coherence REST

You can configure an HTTP acceptor and login module to provide authentication for Coherence REST. HTTP basic authentication provides authentication using credentials (username and password) that are encoded and sent in the HTTP authorization request header. HTTP basic authentication requires a Java Authentication and Authorization Service (JAAS) login module.

This section includes the following topics:

- Specify Basic Authentication for an HTTP Acceptor
- Specify a Login Module

Specify Basic Authentication for an HTTP Acceptor

To specify basic authentication for an HTTP Acceptor:

Add an `<auth-method>` element, within the `http-acceptor` element, that is set to `basic`.

```
<proxy-scheme>
   <service-name>RestHttpProxyService</service-name>
   <acceptor-config>
      <http-acceptor>
         ...
         <auth-method>basic</auth-method>
      </http-acceptor>
   </acceptor-config>
   <autostart>true</autostart>
</proxy-scheme>
```

Specify a Login Module

Configure the JAAS entry used by the standard REST HTTP authentication path.

Copyright © 2008, 2026, Oracle and/or its affiliates.

The standard Coherence HTTP server uses the `CoherenceREST` JAAS entry for its Basic identity asserter. Configure a real login module for that entry and deploy its dependencies on the proxy's runtime classpath/module path. A custom HTTP server/identity integration can have a different entry name; verify the actual deployed configuration.

```
CoherenceREST {
  com.tangosol.security.KeystoreLogin required
    keyStorePath="/run/coherence/http-users.p12"
    keyStoreType="PKCS12";
};
```

This example uses the provided `coherence-login` module and a provisioned keystore identity. Set `java.security.auth.login.config` to the protected JAAS configuration file, satisfy the module's credential callback contract, and verify the resulting subject. Use a supported enterprise module where that identity system is required.

Do not install libraries into a JRE `lib/ext` directory on current JDKs. Use the application's supported dependency/classpath mechanism. Test invalid credentials and a denied operation with valid credentials as well as a successful login.

Using SSL Authentication With Oracle Coherence REST

You can use SSL to provide authentication for Coherence REST. SSL provides an authentication mechanism that relies on digital certificates and encryption keys to establish both identity and trust. See Overview of SSL/TLS.
Client-side SSL certificates are passed to the HTTP acceptor to authenticate the client. SSL requires an SSL-based socket provider to be configured for the HTTP acceptor. The below instructions only describe how to configure SSL and define an SSL socket provider on the proxy for an HTTP acceptor. Refer to your REST client library documentation for instructions on setting up SSL on the client side.

This section includes the following topics:

- Specify Basic Authentication for an HTTP Acceptor
- Configure an HTTP Acceptor SSL Socket Provider
- Access Secured REST Services

Specify Basic Authentication for an HTTP Acceptor

To specify basic authentication for an HTTP Acceptor:

Add an `<auth-method>` element, within the `http-acceptor` element, that is set to `basic`.

```
<proxy-scheme>
   <service-name>RestHttpProxyService</service-name>
   <acceptor-config>
      <http-acceptor>
         ...
         <auth-method>basic</auth-method>
      </http-acceptor>
   </acceptor-config>
   <autostart>true</autostart>
</proxy-scheme>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

Configure an HTTP Acceptor SSL Socket Provider

Configure an SSL socket provider for an HTTP acceptor when using SSL for authentication. To configure SSL for an HTTP acceptor, explicitly add an SSL socket provider definition or reference an SSL socket provider definition that is in the operational override file.

Explicitly Defining an SSL Socket Provider

To explicitly configure an SSL socket provider for an HTTP acceptor, add a `<socket-provider>` element within the `<http-acceptor>` element of each `<proxy-scheme>` definition. See socket-provider in *Developing Applications with Oracle Coherence*.

Example 10-1 demonstrates configuring an SSL socket provider that uses the default values for the `<protocol>` and `<algorithm>` element (`TLS` and `SunX509`, respectively). These are shown for completeness but may be left out when using the default values.

Example 10-1 configures both an identity keystore (`server.jks`) and a trust keystore (`trust.jks`). This is typical of two-way SSL authentication, in which both the client and proxy must exchange digital certificates and confirm each other's identity. For one-way SSL authentication, the proxy server configuration must include an identity keystore but need not include a trust keystore.

Example 10-1 Sample HTTP Acceptor SSL Configuration

```
<proxy-scheme>
   <service-name>RestHttpProxyService</service-name>
   <acceptor-config>
      <http-acceptor>
         ...
         <socket-provider>
            <ssl>
               <protocol>TLS</protocol>
               <identity-manager>
                  <algorithm>SunX509</algorithm>
                  <key-store>
                     <url>file:server.jks</url>
                     <password>password</password>
                     <type>JKS</type>
                  </key-store>
                  <password>password</password>
               </identity-manager>
               <trust-manager>
                  <algorithm>SunX509</algorithm>
                  <key-store>
                     <url>file:trust.jks</url>
                     <password>password</password>
                     <type>JKS</type>
                  </key-store>
               </trust-manager>
            </ssl>
         </socket-provider>
         ...
         <auth-method>cert</auth-method>
      </http-acceptor>
   </acceptor-config>
   <autostart>true</autostart>
</proxy-scheme>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

Referencing an SSL Socket Provider Definition

The following example references an SSL socket provider configuration that is defined in the `<socket-providers>` element of the operational deployment descriptor by specifying the `id` attribute (`ssl`) of the configuration. See socket-providers in *Developing Applications with Oracle Coherence*.

> **Note**
>
> A predefined SSL socket provider is included in the operational deployment descriptor and is named `ssl`. The predefined SSL socket provider is configured for two-way SSL connections and is based on peer trust, in which every trusted peer resides within a single JKS keystore. See Coherence PeerX509 Algorithm for details. To configure a different SSL socket provider, use an operational override file to modify the predefined SSL socket provider or to create a socket provider configuration as required.

```
<proxy-scheme>
   <service-name>RestHttpProxyService</service-name>
   <acceptor-config>
      <http-acceptor>
         ...
         <socket-provider>ssl</socket-provider>
         ...
         <auth-method>cert</auth-method>
      </http-acceptor>
   </acceptor-config>
   <autostart>true</autostart>
</proxy-scheme>
```

Access Secured REST Services

The following example demonstrates a Jersey-based client that accesses REST services that require certificate and HTTP basic authentication.

Client SSL Configuration File

The client SSL configuration file (`ssl.xml`) configures the client's keystore and trust keystore.

```
<ssl>
  <identity-manager>
    <key-store>
      <url>file:keystore.jks</url>
      <password>password</password>
    </key-store>
    <password>password</password>
  </identity-manager>
  <trust-manager>
    <key-store>
      <url>file:trust.jks</url>
      <password>password</password>
    </key-store>
  </trust-manager>
</ssl>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

Sample Jersey SSL Client

```
package example;
import com.oracle.coherence.common.net.SSLSocketProvider;
import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.ClientResponse;
import com.sun.jersey.api.client.WebResource;
import com.sun.jersey.api.client.config.DefaultClientConfig;
import com.sun.jersey.client.urlconnection.HTTPSProperties;
import com.sun.jersey.api.client.filter.HTTPBasicAuthFilter;
import com.tangosol.internal.net.ssl.LegacyXmlSSLSocketProviderDependencies;
import com.tangosol.run.xml.XmlDocument;
import com.tangosol.run.xml.XmlHelper;
import javax.net.ssl.HostnameVerifier;
import javax.net.ssl.SSLSession;
import jakarta.ws.rs.core.MediaType;

public class SslExample
   {
   public static Client createHttpsClient(SSLSocketProvider provider)
      {
      DefaultClientConfig dcc = new DefaultClientConfig();
      HTTPSProperties prop = new HTTPSProperties(new HostnameVerifier()
         {
         public boolean verify(String s, SSLSession sslSession)
            {
               return true;
            }
         }, provider.getDependencies().getSSLContext());
      dcc.getProperties().put(HTTPSProperties.PROPERTY_HTTPS_PROPERTIES, prop);
      return Client.create(dcc);
      }

   public static void PUT(String url, MediaType mediaType, String data)
      {
      process(url, "put", mediaType, data);
      }

   public static void GET(String url, MediaType mediaType)
      {
      process(url, "get", mediaType, null);
      }

   public static void POST(String url, MediaType mediaType, String data)
      {
      process(url, "post", mediaType, data);
      }

   public static void DELETE(String url, MediaType mediaType)
      {
      process(url, "delete", mediaType, null);
      }

   static void process(String url, String action, MediaType mediaType, String
      data)
      {
      try
         {
         XmlDocument xml = XmlHelper.loadFileOrResource("/ssl.xml", null);
         SSLSocketProvider provider = new SSLSocketProvider(new
            LegacyXmlSSLSocketProviderDependencies(xml));
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
            Client client = createHttpsClient(provider);
            ClientResponse response = null;
            WebResource webResource = client.resource(url);

            // If you've specified the "cert+basic" auth-method in your Proxy
            // http-acceptor configuration, initialize and add an HTTP basic
            // authentication filter by
            // uncommenting the following line and changing the username and password
            // appropriately.
            //client.addFilter(new HTTPBasicAuthFilter("username", "password"));

            if (action.equalsIgnoreCase("get"))
               {
               response = webResource.type(mediaType).get(ClientResponse.class);
               }
            else if (action.equalsIgnoreCase("post"))
               {
               response = webResource.type(mediaType).post
                  (ClientResponse.class, data);
               }
            else if (action.equalsIgnoreCase("put"))
               {
               response = webResource.type(mediaType).put
                  (ClientResponse.class, data);
               }
            else if (action.equalsIgnoreCase("delete"))
               {
               response = webResource.type(mediaType).delete
                  (ClientResponse.class, data);
               }
            System.out.println("response status:" + response.getStatus());
            if (action.equals("get"))
               {
               System.out.println("Result: " + response.getEntity(String.class));
               }
            }
         catch (Exception e)
            {
            e.printStackTrace();
            }
         }

      public static void main(String args[])
         {
         PUT("https://localhost:8080/dist-http-example/1",
            MediaType.APPLICATION_JSON_TYPE, "{\"name\":\"chris\",\"age\":32}");
         PUT("https://localhost:8080/dist-http-example/2",
            MediaType.APPLICATION_XML_TYPE,
            "<person><name>admin</name><age>30</age></person>");
         DELETE("https://localhost:8080/dist-http-example/1",
            MediaType.APPLICATION_XML_TYPE);
         GET("https://localhost:8080/dist-http-example/2",
            MediaType.APPLICATION_XML_TYPE);
         }
      }
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

Using SSL and HTTP Basic Authentication with Oracle Coherence REST

You can use SSL together with HTTP basic authentication for added protection when securing Coherence REST. See Using HTTP Basic Authentication with Oracle Coherence REST and Using SSL Authentication With Oracle Coherence REST, respectively.
To specify the use of both HTTP basic authentication and SSL, add an `<auth-method>` element, within the `http-acceptor` element, that is set to `cert+basic`.

```
<proxy-scheme>
   <service-name>RestHttpProxyService</service-name>
   <acceptor-config>
      <http-acceptor>
         ...
         <socket-provider>
            <ssl>
            ...
            </ssl>
         </socket-provider>
         ...
         <auth-method>cert+basic</auth-method>
      </http-acceptor>
   </acceptor-config>
   <autostart>true</autostart>
</proxy-scheme>
```

Implementing Authorization For Oracle Coherence REST

Oracle Coherence REST relies on the Oracle Coherence*Extend authorization framework to restrict which operations a REST client performs on a cluster. For detailed instructions on implementing Oracle Coherence*Extend-style authorization, see Implementing Extend Client Authorization.
Oracle Coherence*Extend-style authorization with REST requires basic HTTP authentication or HTTP basic authentication together with SSL authentication. That is, when implementing authorization, both HTTP basic authentication and SSL can be used together for added protection. For details on using HTTP basic authentication, see Using HTTP Basic Authentication with Oracle Coherence REST. For details on using SSL with HTTP Basic Authentication, see Using SSL and HTTP Basic Authentication with Oracle Coherence REST.

> **Note**
>
> When using SSL and HTTP basic authentication together, make sure that SSL is setup as shown in Using SSL Authentication With Oracle Coherence REST in addition to setting up HTTP basic authentication.

Copyright © 2008, 2026, Oracle and/or its affiliates.

11

Securing Management and Health Services

Protect administrative listeners, invocation targets, Reporter resources, health mutators, and connector discovery.
Inventory Management over REST, JMX/RMI, JMXMP, health, and metrics as separate surfaces. Limit their interfaces and network reachability to the required administrators or monitoring systems. Configure authentication and TLS appropriate to each listener; a secured cache client does not secure an independently exposed management endpoint.

Use Management Authentication and Compatibility for profile-aware defaults and explicit overrides, Securing JMX Connectors and Reporter for non-HTTP paths, and Securing Health Administration for route-local health mutation policy. Metrics retains its own chapter and scraper recipe.

Administrative dispatch policy

Management REST permits only the supported JFR diagnostic operations `jfrStart`, `jfrStop`, `jfrDump`, and `jfrCheck`. Unsupported names are invalid route input and return 400 before diagnostic invocation. Wrapped JMX retains its documented compatibility operation for pre-JDK-11 JFR workflows; this is not a general-purpose command interface.

Management invocation checks restrict product function classes, concrete MBean targets, arguments, descriptors, readable/writable attributes, and supported operations. A wildcard ObjectName query is a selection pattern, not authority to invoke every result. Concrete results are validated before use. Direct JMX/TCMP routes converge on the relevant sink policy; REST pre-validation alone does not replace it.

Operational verification

Test each actual path used by administrators, including Reporter scheduled execution and external JMX tools. A harmless HTTP GET does not exercise mutation, diagnostic, file-output, or TCMP invocation policy. Collect the dedicated management/Reporter warnings as well as serialization/executable signals.

- Management Authentication and Compatibility
- Securing JMX Connectors and Reporter
- Securing Health Administration
- About Securing Oracle Coherence HTTP Management Server
- Basic Authentication for Coherence HTTP Management Server HTTP Acceptor
- Using SSL Authentication With Oracle Coherence HTTP Management Server

Management Authentication and Compatibility

Configure explicit management and metrics authentication and account for compatibility-sensitive administrative behavior.
Management security follows the same two-axis contract. `coherence.mode=dev|prod` remains the runtime/license choice. Starting with Coherence 26.1.0.0, an unset `coherence.security.mode` selects `hardened` and enforces the compatibility-sensitive management gates. Explicit `coherence.security.mode=compatibility` preserves the older

Copyright © 2008, 2026, Oracle and/or its affiliates.

behavior and emits bounded shadow warnings where available. Several management floors apply in both profiles.

Property	Owner and default	Override, signal, and rolling-upgrade note
`coherence.management.http.auth`	Security mode owns the shipped management HTTP default: `none` in compatibility and `basic` in hardened mode. Runtime DEV/PROD does not change that choice.	Set an explicit supported HTTP auth method, including `basic`, `cert`, `cert+basic`, or the compatibility escape `none`. A resolved `none` emits a startup WARN. Apply the same explicit setting and credentials to each endpoint before changing the security profile during a rolling restart.
`coherence.metrics.http.auth`	Security mode owns the shipped metrics HTTP default: `none` in compatibility and `basic` in hardened mode. Runtime DEV/PROD does not change that choice.	Set an explicit supported HTTP auth method. Explicit `none` remains available and emits a startup WARN. Update scrapers and every metrics endpoint before a hardened rolling restart so mixed members do not alternate between open and authenticated scrape behavior.
`coherence.health.http.mutators.enabled`	Explicit and mode-independent; default `false`. Only `PUT /suspend*` and `PUT /resume*` are mutators. Read-only health probes remain unauthenticated.	Set `true` on each member that must expose mutators. Disabled requests are rejected with a bounded WARN. There is no security-mode default flip, so stage enablement per member during a rolling restart.
`coherence.health.http.auth`	Explicit and mode-independent. Once mutators are enabled, an absent or blank value defaults to `basic`; supported values are `basic`, `cert`, and `cert+basic`.	Configure credentials or TLS peer identity before enabling mutators. `none`, unknown values, and missing required identity fail closed and log only bounded rejection information. Configure authentication on every enabled member; this gate authenticates administrative access but does not add role-based authorization.
`coherence.management.report.remote.allowed`	Security mode owns unapproved remote Reporter sources. Compatibility permits them with a structured `result=would_reject` WARN; hardened mode rejects them. The property is unset by default.	Set a comma-separated list of exact `http` or `https` scheme/host authorities, including a port when non-default. Approved sources are accepted without a less-secure warning. Use one allowlist across Reporter members before enabling hardened mode; otherwise the same report can succeed on one member and fail on another.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Property	Owner and default	Override, signal, and rolling-upgrade note
`coherence.reporter.output.directory`	Explicit root with mode-sensitive enforcement; default is the process working directory. Compatibility can allow an existing output path outside the root with a structured `result=would_reject` WARN, while hardened mode rejects it. Traversal and final-file root escape remain hard failures.	Set a canonical approved output directory and move existing report targets under it. Use a path valid on every Reporter member before a rolling security profile change. Normal in-root writes emit no policy warning.
`coherence.jmxmp.host`	Explicit and mode-independent; default `127.0.0.1`.	Set a deliberate non-loopback address only when remote JMXMP access is required. A wider bind has no dedicated policy warning, so protect it with network controls and the mandatory authenticator. Configure each member before exposing its listener.
`coherence.jmxmp.port`	Explicit and mode-independent; default `9000`.	Set the required listener port on each JMXMP member. The port does not weaken authentication or serial-filter requirements and has no less-secure warning.
`coherence.jmxmp.authenticator`	Explicit and mode-independent; no default. JMXMP startup fails closed when neither this property nor the remote-management fallback names a valid `JMXAuthenticator`.	Set a `JMXAuthenticator` implementation class. Configure it before enabling or widening JMXMP on each member. Startup failure, rather than a compatibility shadow, is the missing-authenticator signal.
`coherence.management.remote.authenticator`	Explicit and mode-independent; no default. It configures JMX RMI authentication and is the fallback authenticator for JMXMP.	Set a `JMXAuthenticator` implementation class on every remote-management member. JMX RMI has no absent-authenticator policy WARN; an invalid class logs an error and does not produce authenticated connector attributes. JMXMP still fails closed if the fallback is absent or unusable.
`coherence.management.remote.registryport`	Explicit remediation for management JMX publication. When absent, the cluster management connector can produce a dynamic `/stub/...` URL. Compatibility permits that URL with a bounded `result=would_reject` warning; hardened mode rejects its publication. Runtime DEV/PROD does not change the decision.	Set a registry port, commonly `9000`, before hardened rollout so publication uses a static `/jndi/rmi://.../server` URL. Configure all publishing members before changing the security profile; otherwise hardened members can reject URLs still produced by compatibility-profile members.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Verifying Compatibility Shadows

Before enabling hardened security mode, keep the intended runtime `coherence.mode` unchanged and exercise management TCMP, Reporter, management HTTP, metrics, health administration, and JMX publication in compatibility security mode. Inspect two separate signal families:

- `coh.executable.policy_check{result=would_reject,...}` is emitted only by paths that call `DefaultRemoteExecutablePolicy.enforce(...)`. It identifies unclassified remote executable classes; it is not a complete management hardening audit.
- `ManagementInvocationPolicy` and `ReporterSecurity` emit separate bounded structured WARN lines with `result=would_reject`. These warnings cover management invocation and Reporter policy decisions and do not increment the executable-policy counter. Dynamic JMX `/stub/...` publication also emits its own bounded compatibility warning.

Classify legitimate remote executables, approve Reporter sources and output roots, set a static RMI registry port, and configure endpoint authentication. Then rerun the full workflow with `coherence.security.mode=hardened` in pre-production. During a rolling change, keep members serving the same external endpoint on one security profile where practical; mixed profiles can accept and reject the same request differently.

Securing JMX Connectors and Reporter

Configure authenticated connectors and deliberate report-source, output, and MBean-target boundaries.

JMXMP

The product JMXMP server defaults to `127.0.0.1:9000`. A non-loopback bind is an explicit exposure decision. The connector requires a usable `JMXAuthenticator`, configured with `coherence.jmxmp.authenticator` or the supported `coherence.management.remote.authenticator` fallback. Missing or invalid authentication configuration prevents safe connector startup.

```
-Dcoherence.management.serverfactory=com.tangosol.net.management.JmxmpServer
-Dcoherence.jmxmp.host=127.0.0.1
-Dcoherence.jmxmp.port=9000
-Dcoherence.jmxmp.authenticator=com.example.security.AdminAuthenticator
```

`AdminAuthenticator` is an application `javax.management.remote.JMXAuthenticator` implementation that validates credentials using the approved identity service and returns a verified subject. Configure the provider-specific TLS/SASL profile supported by the installed JMXMP implementation before exposing it remotely. An authenticator supplies authentication; it does not itself encrypt the connection or implement a complete MBean role policy.

Before creating the connector, Coherence uses an existing JVM-wide `ObjectInputFilter` or installs a constrained fallback and verifies its presence. The fallback affects the JVM, not only a single connector. Review it with other libraries, their required serializable values, and the supported JDK filter policy. It is inaccurate to assume that an absent preconfigured filter always causes failure; inability to install or observe a filter does fail startup.

JMX/RMI and discovery

Configure the supported remote authenticator and a deliberate registry port. In hardened mode, dynamic `/stub/...` management connector publication is rejected; configure

Copyright © 2008, 2026, Oracle and/or its affiliates.

`coherence.management.remote.registryport` so publication uses a static `/jndi/rmi://.../server` form. Apply the configuration on all publishing members before profile activation. Restrict registry and connector network access and use the supported provider's transport security.

Reporter inputs and output

Use approved local report configuration and a dedicated canonical output root. `coherence.reporter.output.directory` defaults to the working directory. Move output under the approved root before enabling hardened enforcement. Traversal, invalid output protocols, and final-file escapes remain hard failures where documented.

Approve necessary remote HTTP(S) report authorities with `coherence.management.report.remote.allowed`, using exact scheme/host authorities and a non-default port when needed. Approval controls source policy; protect transport, remote content, and redirects independently according to supported Reporter behavior. Avoid approving a broad network merely to silence a warning.

Report queries, concrete ObjectNames, method-column provenance, and custom column classes have their own policy checks. Broad patterns do not bypass concrete target validation. Validate the scheduled report through the same JMX/TCMP path used in production, and confirm an unapproved source and outside-root output are rejected under hardened policy.

Securing Health Administration

Keep read-only probes available while enabling authenticated suspend/resume routes only where needed.
Health mutators are disabled by default in both security profiles. Only the supported `PUT /suspend*` and `PUT /resume*` routes are mutators; ordinary read-only health probes remain unauthenticated. Apply network controls appropriate to even read-only operational information.

```
-Dcoherence.health.http.mutators.enabled=true
-Dcoherence.health.http.auth=basic
```

Enable mutators only on the intended members, with TLS and valid credentials already configured. Once enabled, absent/blank auth defaults to `basic`; supported methods are `basic`, `cert`, and `cert+basic`. `none`, unknown methods, and missing required identity fail closed. A disabled mutator is rejected with a bounded warning.

This route policy authenticates administrative access; it does not create a new role-based health authorization model. If only a subset of authenticated identities may administer health state, enforce the required restriction in the approved identity/ingress architecture and verify it explicitly.

Test readiness/liveness requests without credentials, a disabled mutator, enabled mutators without/with incorrect credentials, and an authorized suspend/resume pair on an isolated member. Keep the member available for recovery if the test intentionally suspends a service. Coordinate monitoring and load balancers with the resulting state changes.

About Securing Oracle Coherence HTTP Management Server

Select management HTTP authentication explicitly and match its security-profile default to the deployment.
Starting with Coherence 26.1.0.0.0, the default hardened profile makes the shipped management HTTP authentication default `basic`. Explicit compatibility mode uses `none`. Set

Copyright © 2008, 2026, Oracle and/or its affiliates.

`coherence.management.http.auth` deliberately during migration. A successful login does not replace authorization for the administrative operation. See Management Authentication and Compatibility.

Coherence HTTP Management Server authentication support includes: HTTP basic, client-side SSL certificate, and client-side SSL certificate together with HTTP basic.

See Accessing Management Information Using REST in *Managing Oracle Coherence*.

Basic Authentication for Coherence HTTP Management Server HTTP Acceptor

You can configure an HTTP acceptor to provide authentication for Coherence HTTP Management Server.
HTTP basic authentication provides authentication using credentials (user name and password) that are encoded and sent in the HTTP authorization request header.

This section includes the following topics:

- Specify the Basic Authentication for Coherence HTTP Management Server HTTP Acceptor
- Specify a Coherence HTTP Management Server Login Module

Specify the Basic Authentication for Coherence HTTP Management Server HTTP Acceptor

The default `management-http-config.xml` is in `coherence-management.jar`.

To specify basic authentication for an HTTP Management Acceptor, set the `coherence.management.http.auth` system property to the value `basic` or override the default `management-http-config.xml` and specify `<auth-method>` child xml element to the value `basic`.

Specify a Coherence HTTP Management Server Login Module

Configure and verify the JAAS identity path used by the management HTTP server.
For the standard Coherence HTTP server's Basic path, configure the `CoherenceREST` JAAS entry with the module that validates administrator credentials. The entry name follows the actual HTTP identity asserter; it is not inferred from the management endpoint's display name. A custom provider can define another contract.

Deploy the module and dependencies through the supported runtime classpath/module path and select the protected JAAS file with `java.security.auth.login.config`. Use the provided keystore module only where a provisioned keystore identity is appropriate. Current JDKs do not use the former JRE extension-directory installation mechanism.

Configure management TLS, network placement, and the intended authorization restrictions separately. Verify missing/invalid credentials and a legitimate administrative request. Review Management Authentication and Compatibility before changing profile-sensitive defaults.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Using SSL Authentication With Oracle Coherence HTTP Management Server

You can use SSL to provide authentication for Coherence HTTP Management Server. SSL provides an authentication mechanism that relies on digital certificates and encryption keys to establish both identity and trust. See Overview of SSL/TLS.
Client-side SSL certificates are passed to the HTTP acceptor to authenticate the client. SSL requires an SSL-based socket provider to be configured for the HTTP acceptor.

This section includes the following topics:

- Configure a Coherence HTTP Management Acceptor SSL Socket Provider

Configure a Coherence HTTP Management Acceptor SSL Socket Provider

Configure an SSL socket provider for an HTTP acceptor when using SSL for authentication. To configure SSL for an HTTP acceptor, explicitly add an SSL socket provider definition or reference an SSL socket provider definition that is in the operational override file.

Explicitly Defining an SSL Socket Provider

To explicitly configure an SSL socket provider for an HTTP acceptor, add a `<socket-provider>` element within the `<http-acceptor>` element of each `<proxy-scheme>` definition. See socket-provider in *Developing Applications with Oracle Coherence*. You can override the default values by extracting `management-http-config.xml` from `coherence.jar`, modifying the file and then placing it before `coherence.jar` on the class path. You can also specify the socket provider by using the override `-Dcoherence.management.http.provider=`*`your-socket-provider`*.

Example 12-1 demonstrates configuring an SSL socket provider that uses the default values for the `<protocol>` and `<algorithm>` element (`TLS` and `SunX509`, respectively). These are shown for completeness but may be left out when using the default values.

Example 12-1 configures both an identity keystore (`server.jks`) and a trust keystore (`trust.jks`). This is typical of two-way SSL authentication, in which both the client and proxy must exchange digital certificates and confirm each other's identity. For one-way SSL authentication, the proxy server configuration must include an identity keystore but need not include a trust keystore.

Example 11-1 Sample HTTP Acceptor SSL Configuration

```
<proxy-scheme>
   <service-name>ManagementHttpProxy</service-name>
   <acceptor-config>
      <http-acceptor>
         ...
        <socket-provider>
       <ssl>
            <protocol>TLS</protocol>
            <identity-manager>
              <algorithm>SunX509</algorithm>
              <provider system-
property="coherence.management.http.security.keystore.provider"/>
              <key-store>
                <url system-
property="coherence.management.http.security.keystore">file:server.jks</url>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
                <password system-
property="coherence.management.http.security.keystore.password"/>
                <type>JKS</type>
              </key-store>
              <password system-
property="coherence.management.http.security.identitymanager.password″/>
            </identity-manager>
            <trust-manager>
              <algorithm>SunX509</algorithm>
              <provider system-
property="coherence.management.http.security.truststore.provider"/>
              <key-store>
                <url system-
property="coherence.management.http.security.truststore">file:truststore.jks</url>
                <password system-
property="coherence.management.http.security.truststore.password"/>
                <type>JKS</type>
              </key-store>
            </trust-manager>
          </ssl>
        </socket-provider>
         ...
        <auth-method>cert</auth-method>
      </http-acceptor>
   </acceptor-config>
   <autostart>true</autostart>
</proxy-scheme>
```

Referencing an SSL Socket Provider Definition Using Coherence HTTP Management Over REST

The following example references an SSL socket provider configuration that is defined in the `<socket-providers>` element of the operational deployment descriptor by specifying the `id` attribute (`ssl`) of the configuration. See socket-providers in *Developing Applications with Oracle Coherence*.

> **Note**
>
> A predefined SSL socket provider is included in the operational deployment descriptor and is named `ssl`. The predefined SSL socket provider is configured for two-way SSL connections and is based on peer trust, in which every trusted peer resides within a single JKS keystore. See Coherence PeerX509 Algorithm. To configure a different SSL socket provider, use an operational override file to modify the predefined SSL socket provider or to create a socket provider configuration as required.

```
<proxy-scheme>
  <service-name>ManagementHttpProxy</service-name>
  <acceptor-config>
        <http-acceptor>
           ...
           <socket-provider>ssl</socket-provider>
           ...
           <auth-method>cert</auth-method>
        </http-acceptor>
   </acceptor-config>
   <autostart>true</autostart>
</proxy-scheme>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

For configuring HTTP client access, see Access Secured REST Services.

Copyright © 2008, 2026, Oracle and/or its affiliates.

12

Securing Oracle Coherence Metrics

Authenticate scrapers, protect metric transport, and coordinate endpoint policy across members.
Metrics exposes operational information on its own HTTP endpoint. Restrict its bind/network placement to the intended scrapers and operators. Configure TLS and the required Basic, client-certificate, or combined authentication supported by the acceptor.

Starting with Coherence 26.1.0.0, the default hardened profile makes the shipped metrics HTTP authentication default `basic`. Explicit compatibility mode uses `none`, subject to operator XML/property configuration. An explicit `none` override remains available and emits a warning. Configure credentials and scraper behavior before upgrading; a load-balanced scrape must not alternate unexpectedly between open and authenticated members.

Use Configuring an Authenticated TLS Metrics Scrape. The retained endpoint and client topics provide HTTP acceptor placement. Read-only metrics, management mutation, and health administration remain different surfaces with different controls.

- Configuring an Authenticated TLS Metrics Scrape
- About Securing Oracle Coherence Metrics
- Basic Authentication for Coherence Metrics Http Acceptor
- Using SSL Authentication With Oracle Coherence Metrics

Configuring an Authenticated TLS Metrics Scrape

Use CA-verified HTTPS and a protected credential source for the metrics scraper. Configure the metrics HTTP acceptor with the chosen named TLS socket provider and `coherence.metrics.http.auth=basic` for this recipe. Provide a working `CoherenceREST` JAAS entry for the standard HTTP server's Basic path, or configure the supported replacement identity integration. Choose a least-privilege scraper identity and the actual member/service DNS names.

The following Prometheus fragment uses HTTPS, CA verification, and a mounted password file. Replace the example target/port with the endpoint configured in this deployment and verify the Prometheus version's configuration syntax.

```
scrape_configs:
  - job_name: coherence
    scheme: https
    static_configs:
      - targets: ["metrics.example.com:9612"]
    basic_auth:
      username: coherence-scraper
      password_file: /run/secrets/coherence-metrics-password
    tls_config:
      ca_file: /run/coherence/metrics-ca.pem
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

For certificate authentication, configure the scraper's client certificate and key and the acceptor's corresponding trust/identity mapping. Do not disable certificate verification or reuse an administrator's credential merely to make scraping work.

Verify a successful scrape, missing/wrong credentials, an unrelated CA, and a server-name mismatch. Repeat through any load balancer and across all endpoints during rolling restart. Rotate the credential/certificate with the scraper's supported reload lifecycle, verify fresh requests, and retire old material after the overlap required by the endpoint fleet.

About Securing Oracle Coherence Metrics

Select metrics authentication explicitly and account for shipped profile-sensitive defaults. Use a secured listener and a deliberately configured scraper identity for operational metrics. Supported HTTP acceptor methods include Basic, certificate, and combined authentication. The shipped profile-sensitive default is `basic` under the hardened profile, which is the default starting with Coherence 26.1.0.0; explicit compatibility mode uses `none`. Operator configuration can override that default.

Set `coherence.metrics.http.auth` explicitly when the deployment requires a stable endpoint policy across rolling changes. Configure the corresponding JAAS or certificate identity path and TLS provider. Verify the authenticated client and a denied unauthenticated request with Configuring an Authenticated TLS Metrics Scrape.

Basic Authentication for Coherence Metrics Http Acceptor

You can configure an HTTP acceptor to provide authentication for Coherence Metrics. HTTP basic authentication provides authentication using credentials (user name and password) that are encoded and sent in the HTTP authorization request header.
This section includes the following topics:

- Specify Basic Authentication for Coherence Metrics HTTP Acceptor
- Specify a Coherence Metrics Login Module
- Specify Basic Authentication for a Coherence Metrics HTTP Client

Specify Basic Authentication for Coherence Metrics HTTP Acceptor

The default `metrics-http-config.xml` is in `coherence-metrics.jar`.

To specify basic authentication for an HTTP Acceptor, set the system property `coherence.metrics.http.auth` to the value `basic` or override the default `metrics-http-config.xml` and specify `<auth-method>` child xml element to the value `basic`.

Specify a Coherence Metrics Login Module

Configure the JAAS entry used by the standard metrics HTTP Basic authentication path. The standard Coherence HTTP server's Basic identity asserter uses the `CoherenceREST` JAAS entry. Configure a module that validates the scraper's credentials and deploy its runtime dependencies through the supported classpath/module path. Verify another entry name only when using a custom HTTP integration that actually selects it.

Protect the JAAS file and secret sources. Current JDKs do not support the former JRE extension-directory mechanism for installing login modules. Set explicit metrics authentication and TLS, and configure the scraper using Configuring an Authenticated TLS Metrics Scrape. Verify wrong/missing credentials and member-by-member rollout behavior.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Specify Basic Authentication for a Coherence Metrics HTTP Client

Prometheus is an HTTP client metrics gathering system that can be configured to scrape metrics data from a Coherence metrics endpoint.

See Prometheus <scrape_config> configuration for parameters on configuring scheme to `https`, `basic_auth` with `username` and `password`.

Using SSL Authentication With Oracle Coherence Metrics

You can use SSL to provide authentication for Coherence Metrics. SSL provides an authentication mechanism that relies on digital certificates and encryption keys to establish both identity and trust. See Overview of SSL/TLS.
Client-side SSL certificates are passed to the HTTP acceptor to authenticate the client. SSL requires an SSL-based socket provider to be configured for the HTTP acceptor.

This section includes the following topics:

- Configure a Coherence Metrics HTTP Acceptor SSL Socket Provider

Configure a Coherence Metrics HTTP Acceptor SSL Socket Provider

Configure an SSL socket provider for an HTTP acceptor when using SSL for authentication. To configure SSL for an HTTP acceptor, explicitly add an SSL socket provider definition or reference an SSL socket provider definition that is in the operational override file.

Explicitly Defining an SSL Socket Provider

To explicitly configure an SSL socket provider for an HTTP acceptor, add a `<socket-provider>` element within the `<http-acceptor>` element of each `<proxy-scheme>` definition. See socket-provider in *Developing Applications with Oracle Coherence*. You can override the default `metrics-http-config.xml` by making a copy of it and placing the revised `metrics-http-config.xml` in classpath before `coherence-metrics.jar` occurs.

Example 12-1 demonstrates configuring an SSL socket provider that uses the default values for the `<protocol>` and `<algorithm>` element (`TLS` and `SunX509`, respectively). These are shown for completeness but may be left out when using the default values.

Example 12-1 configures both an identity keystore (`server.jks`) and a trust keystore (`trust.jks`). This is typical of two-way SSL authentication, in which both the client and proxy must exchange digital certificates and confirm each other's identity. For one-way SSL authentication, the proxy server configuration must include an identity keystore but need not include a trust keystore.

Example 12-1 Sample HTTP Acceptor SSL Configuration

```
<proxy-scheme>
   <service-name>MetricsHttpProxyService</service-name>
   <acceptor-config>
      <http-acceptor>
         ...
        <socket-provider>
       <ssl>
            <protocol>TLS</protocol>
            <identity-manager>
              <algorithm>SunX509</algorithm>
              <provider system-property="coherence.metrics.security.keystore.provider"/>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
              <key-store>
                <url system-
property="coherence.metrics.security.keystore">file:server.jks</url>
                <password system-
property="coherence.metrics.security.keystore.password"/>
                <type>JKS</type>
              </key-store>
              <password system-
property="coherence.metrics.security.identitymanager.password"/>
            </identity-manager>
            <trust-manager>
              <algorithm>SunX509</algorithm>
              <provider system-
property="coherence.metrics.security.truststore.provider"/>
              <key-store>
                <url system-
property="coherence.metrics.security.truststore">file:truststore.jks</url>
                <password system-
property="coherence.metrics.security.truststore.password"/>
                <type>JKS</type>
              </key-store>
            </trust-manager>
          </ssl>
        </socket-provider>
         ...
        <auth-method>cert</auth-method>
      </http-acceptor>
   </acceptor-config>
   <autostart>true</autostart>
</proxy-scheme>
```

Referencing an SSL Socket Provider Definition Using Coherence Metrics

The following example references an SSL socket provider configuration that is defined in the `<socket-providers>` element of the operational deployment descriptor by specifying the `id` attribute (`ssl`) of the configuration. See socket-providers in *Developing Applications with Oracle Coherence*.

> **Note**
>
> A predefined SSL socket provider is included in the operational deployment descriptor and is named `ssl`. The predefined SSL socket provider is configured for two-way SSL connections and is based on peer trust, in which every trusted peer resides within a single JKS keystore. See Coherence PeerX509 Algorithm. To configure a different SSL socket provider, use an operational override file to modify the predefined SSL socket provider or to create a socket provider configuration as required.

```
<proxy-scheme>
  <service-name>MetricsHttpProxy</service-name>
  <acceptor-config>
        <http-acceptor>
           ...
           <socket-provider>ssl</socket-provider>
           ...
           <auth-method>cert</auth-method>
        </http-acceptor>
   </acceptor-config>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
   <autostart>true</autostart>
</proxy-scheme>
```

Configuring HTTP Client-Side in Prometheus Configuration

Prometheus is an HTTP client metrics gathering system that is used to scrape the Coherence Metrics endpoints. See Prometheus <scrape_config> configuration for parameters to configure `scheme` to `https` and `basic_auth` with `username` and `password`. See Prometheus <tls_config> configuration to configure TLS connections.

Copyright © 2008, 2026, Oracle and/or its affiliates.

13

Securing Cluster Peers and Federation

Protect member admission, discovery, advertised topology, federation participants, and authenticated metadata.
Cluster transport and peer identity are foundational trust decisions. Members host trusted code and participate in distributed state changes; a cluster name, role label, host allowlist, or self-advertised proxy address is not cryptographic proof of authority. Restrict membership and deploy appropriate TLS/mTLS or equivalent authenticated peer identity.

Coherence validates relevant pre-trust federation, NameService, membership, and metadata shapes before treating them as operational state. Those controls reduce unsafe materialization and spoofed-state paths, but they do not turn an intentionally unauthenticated deployment into an automatically authenticated cluster.

Discovery and federation

NameService lookup/bind policy constrains resource types and management connector data. Use approved discovery endpoints, expected cluster/service names, and the same network/trust boundaries as the service ultimately contacted. Validate returned proxy/service endpoints; do not treat self-advertised metadata as independent authority.

Federation uses configured participant and endpoint policy. Hardened direct-connect validation rejects an endpoint that cannot be verified against the configured participant contract; compatibility can retain a bounded would-reject path. Keep participant names, expected addresses, identity, and routing consistent on both sides before enabling a stricter policy.

Management discovery has its own safe URL rules. Configure a static management registry URL rather than depending on dynamic serialized RMI stubs in hardened operation. These checks do not replace TLS or credentials on the connector itself.

Subject and senior metadata proofs

Request-subject proofs bind propagated identity to an authorized authority. Senior/topology proofs protect a different authority and recipient relationship. Use Configuring Peer Proof Prerequisites for their configuration, lifecycle, and compatibility boundaries. Required proof without a usable compatible provider fails closed.

Federation replay authorization and proof of a live peer are distinct from cryptographic authentication of stored journal bytes. Do not infer an authenticated snapshot/journal format from these protocol controls.

- Configuring Peer Proof Prerequisites

Configuring Peer Proof Prerequisites

Stage a compatible production proof provider and verify all required producers and recipients before requiring proofs.
Proofs remain opt-in. Hardened mode enforces request-subject proof on a configured authorizer route only when that route requests it. Explicit `coherence.security.peer.senior-metadata-proof.required=true` fails closed in both security profiles. Neither policy is selected by runtime `coherence.mode=prod`.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Use only a production provider/profile supported for the deployed release. The bounded X.509 profile described here requires per-member signing identities, separate subject/senior authority sets, stable member binding, and a complete readiness/rotation procedure. Deterministic test providers are not production cryptography. Confirm the supported provider and publication/deployment prerequisites before enabling required policy.

Provider material

The `x509-signature-v1` implementation uses protected per-member signing identities and separate subject and senior authorities. Provider enablement and provisioning instructions require a supported, release-qualified operator procedure. Confirm that procedure before configuring the provider or making proof required.

The profile requires RSA keys of at least 2048 bits with SHA-256, non-CA digital-signature leaves, role-specific authority and member binding, certificate-fingerprint key IDs, five-minute proof validity, and two-minute proof/certificate skew. A generic TLS leaf is not automatically a suitable proof authority. Member URI SANs use the profile's exact injective UTF-16-code-unit encoding under `urn:coherence:member:utf16be:`; use the matching provisioning/profile contract rather than inventing a URI spelling. Duplicate live member identities fail readiness.

The restricted credential loader accepts absolute local regular-file paths or authority-free `file:` URIs, and rejects symlinks, directories, remote URLs, classpath/JAR resources, and arbitrary handlers. Its bounded profile includes a one-MiB object ceiling, 16 aliases/role leaves, a four-certificate identity chain, 1024 password characters, bounded KDF work, and bounded credential/close operations. Use the approved JCA/backend/password-provider combination; these limits do not authorize arbitrary custom implementations.

Readiness and rollout

1. Deploy compatible code/configuration to every relevant sender and physical recipient. Include storage-disabled application members that legitimately issue subjects, all relevant service seniors, and ownership-enabled recipients.
2. Stage the provider while proofs are optional. Verify identities, role authority, expiry, refresh, duplicate detection, and the exact service membership used by the proof workflow.
3. Use the supported all-member readiness observation for one stable topology epoch. Local provider status alone cannot establish cluster readiness. Joins, leaves, ownership/seniority changes, missing responses, and even a change restored during collection invalidate the observation.
4. A supported rollout procedure must establish readiness before requiring the desired subject route or senior metadata policy. An absent provider, incompatible peer, or incomplete required initial material must fail closed rather than downgrade.

An authorizer definition requests subject proof with `subject-proof-required="true"`; the route must also have a configured storage authorizer and the hardened security profile for this enforcement. Senior metadata has its separate required property. Do not publish either setting as a standalone switch that supplies keys or trust automatically.

Rotation and rollback

Add new authority material before activating a new signer, retain the overlap needed by live producers/recipients, verify cluster readiness for add/activate/overlap/retire, and then remove retired authority. Refresh publishes an atomic complete generation; a failed optional staged load may remain unavailable or retain usable last-known-good state according to its lifecycle, while required initial policy fails closed.

A rollback must explicitly address both required policies and compatibility with already active producers/recipients. Changing the runtime mode, or inventing a legacy runtime value, does

Copyright © 2008, 2026, Oracle and/or its affiliates.

not disable proof requirements. Preserve protected backups of configuration and key material and test the recovery path before retirement.

Copyright © 2008, 2026, Oracle and/or its affiliates.

14

Securing Coherence RAG

Protect optional RAG administration, model downloads, and document imports with their component-specific policies.
This chapter applies when the Coherence RAG component is installed. Its REST policy is distinct from the general Coherence REST cache API. RAG REST accepts configured bearer tokens and API keys. Basic authentication is rejected at this API layer; an approved proxy can authenticate and forward a supported identity form. Cookie/session authentication is not currently part of this contract.

Sensitive operations require authentication. Configuration, model, and store mutations and document imports require the configured admin role. `_config` reads require authentication and return 404 for sensitive keys. Test both authorization and redaction; do not depend on hiding a UI control.

Use RAG Configuration and Model Download Policy and Restricting RAG Document Imports. Protect API keys and provider secrets using the supported runtime credential sources and network architecture. Do not put secrets into example URLs or log complete authorization headers.

- RAG Configuration and Model Download Policy
- Restricting RAG Document Imports

RAG Configuration and Model Download Policy

Limit mutable RAG settings and approve the local model artifacts that may be downloaded. The default administrative role is `admin`. Restrict runtime configuration writes to the explicit allowlist; sensitive fields and protected security/import controls retain their own deny policy. The following values illustrate the shipped defaults:

```
coherence.rag.security.admin-role=admin
coherence.rag.config.write.allowed-
properties=model.embedding,model.chat,model.scoring,coherence.rag.default.pars
er
```

Test an authenticated non-admin, an allowed admin update, a forbidden property, and sensitive readback. A valid administrative role does not authorize arbitrary property names or provider-secret disclosure. Apply corresponding tests to sibling model/store mutations rather than only `_config`.

Local model downloads

The download allowlist applies to local ONNX embedding/scoring models obtained from HuggingFace. JAR-wired CDI `ModelProvider` implementations do not use this gate, and chat-model selectors are not validated by this download policy.

```
coherence.rag.security.huggingface.allowed-models=sentence-transformers/all-
MiniLM-L6-v2,cross-encoder/ms-marco-TinyBERT-L-2-v2
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

Use exact model names or a deliberately approved owner-wide trailing wildcard. With an empty allowlist, compatibility permits downloads and emits a bounded startup warning; hardened mode rejects downloads until an allowlist is configured. Once non-empty, the allowlist constrains values in either profile.

Approval of a model name is not a general integrity guarantee for all downloaded artifacts. Control provider versions, network egress, artifact storage, and the deployment's model provenance process. Verify an approved model and an unapproved name before rollout.

Restricting RAG Document Imports

Approve import schemes and locations and preserve the validated destination through HTTP and filesystem access.
Document imports are denied until allowed schemes and locations are configured. Use the narrowest locations needed by the ingestion job. The defaults include empty scheme/location allowlists, private-address rejection, a 10-second connect timeout, a 30-second read timeout, and a 50 MiB response bound.

```
coherence.rag.import.allowed-schemes=https
coherence.rag.import.http.allowed-hosts=documents.example.com
coherence.rag.import.http.allow-private-addresses=false
coherence.rag.import.connect-timeout-ms=10000
coherence.rag.import.read-timeout-ms=30000
coherence.rag.import.max-bytes=52428800
```

HTTP destination binding

HTTP(S) imports use direct connections. JVM-selected HTTP, HTTPS-tunnel, and SOCKS proxy routes are rejected with `proxy-not-allowed`. A deployment requiring those proxies needs another import mechanism under this contract.

For every initial request and redirect hop, the entire DNS answer is validated. The connection uses only an accepted address without resolving the origin again or reusing a connection from another hop. HTTPS retains the logical hostname for `Host`, SNI, and certificate verification. A mixed allowed/forbidden DNS answer is rejected before connection.

Loopback, link-local, metadata, and private destinations are rejected by default. The private-address opt-in still requires an allowed host and does not permit link-local metadata targets. Redirects are bounded and revalidated at every hop. Test a disallowed redirect, mixed DNS answers, wrong certificate name, selected proxy, timeout, and oversized response in an isolated ingestion environment.

Local files and cloud locations

For local imports, explicitly allow `file` and configure `coherence.rag.import.file.allowed-roots`. Canonical containment, secure parent traversal, and no-follow channel opening constrain filesystem access. Filesystem providers without `SecureDirectoryStream` support fail closed. A pre-open regular-file check rejects stable special files, including FIFOs.

The Java channel APIs used here do not expose the final file's type attributes through the returned channel. Replacing the final entry between its type check and open remains a race. Keep the allowed tree writable only by trusted ingestion principals. Do not present secure parent traversal as a complete guarantee against a hostile writer of the final entry.

Cloud-source allowlists are provider-specific: `s3.allowed-buckets`, `azure.blob.allowed-containers`, `gcs.allowed-buckets`, and `oci.os.allowed-locations` under `coherence.rag.import`. Allow the matching scheme as well as its location, and configure the

Copyright © 2008, 2026, Oracle and/or its affiliates.

provider's least-privilege credentials. A location allowlist does not grant cloud IAM permission, and IAM permission does not waive the import allowlist.

Copyright © 2008, 2026, Oracle and/or its affiliates.

15

Securing Oracle Coherence in Oracle WebLogic Server

Use supported WebLogic security integration while accounting for independent Coherence profile and application policies.
WebLogic supplies its own runtime identity and secured-production-mode integration. Use the supported WebLogic/Coherence/JDK combination and the retained WebLogic-specific configuration topics for TLS, mTLS, federation, and identity transformer/asserter integration.

WebLogic secured production mode, Coherence runtime/license mode, and `coherence.security.mode` have different responsibilities. Do not infer one setting from another or copy standalone security overrides that unintentionally replace a WebLogic-provided integration. Review the effective socket provider, subject propagation, access controller, and endpoint authentication in the deployed container.

Apply the application-artifact and remote-executable policies to classes deployed in the relevant WebLogic classloader/module scope. Verify generated resources are visible on the actual receivers. Exercise real application identities and denied operations, container restarts, and rolling upgrades; a standalone member smoke does not validate WebLogic identity propagation.

The general recipes in Using TLS to Secure Communication, Securing Serialization and Remote Execution, and Rolling Upgrade to Hardened Security remain useful, with their configuration and lifecycle adapted to the supported WebLogic integration.

- Overview of Securing Oracle Coherence in Oracle WebLogic Server
- Securing Coherence using SSL/TLS
  You can use SSL/TLS to secure Managed Coherence servers in a WebLogic Server domain. WebLogic Server will create Java keystores for identity and trust and create a Coherence socket provider configuration that Coherence will be configured to use.
- Securing Oracle Coherence Cluster Membership
- Authorizing Oracle Coherence Caches and Services
- Securing Extend Client Access with Identity Tokens

Overview of Securing Oracle Coherence in Oracle WebLogic Server

Several security features are used to secure cluster members, caches and services, and extend clients when deploying Coherence within an Oracle WebLogic Server domain. The default security configuration allows any server to join a cluster and any extend client to access a cluster's resources.
The following security features should be configured to protect against unauthorized use of a cluster:

- SSL/TLS - enables SSL/TLS for Coherence cluster member connections
- Coherence access controllers - provides authorization between cluster members

Copyright © 2008, 2026, Oracle and/or its affiliates.

- WebLogic Server authorization - provides authorization to Oracle Coherence caches and services
- Coherence identity tokens - provides authentication for extend clients

Much of the security for Oracle Coherence in a Oracle WebLogic Server domain reuses existing security capabilities. Knowledge of these existing security components is assumed. References are provided in this documentation to existing content where applicable.

Securing Coherence using SSL/TLS

You can use SSL/TLS to secure Managed Coherence servers in a WebLogic Server domain. WebLogic Server will create Java keystores for identity and trust and create a Coherence socket provider configuration that Coherence will be configured to use.

By default, WebLogic Server will configure managed Coherence servers to use mTLS authentication, but these settings can be overridden.

You can only use WebLogic Server to configure SSL/TLS for cluster membership, or globally for all Coherence sockets. There is no way to configure individual Coherence services, such as Extend proxies and client, gRPC proxies and client, federation, and so on. If you need to configure these services differently, see Using TLS to Secure Communication.

- Extended Usage Certificates
  If certificates with extended usage are used, it is important to understand how this affects the different SSL/TLS configuration choices available for Coherence.
- Configure Coherence Cluster Traffic Using mTLS
  Coherence clusters form a peer-to-peer network where each JVM is both a server receiving connections from other cluster members, and a client connecting to other cluster members. You can configure Coherence cluster peer-to-peer communication to use SSL/TLS in a WebLogic Server domain using either WebLogic Remote Console or a WLST script.
- Configure Coherence Cluster Traffic Using One-Way SSL/TLS
  In one-way SSL/TLS, a client authenticates a server certificate, but the server does not receive a certificate from the client, so the client is anonymous. You can configure Coherence cluster traffic to use one-way SSL/TLS in a WebLogic Server domain using either WebLogic Remote Console or a WLST script.
- Using a Custom Coherence Operational Configuration File
  As WebLogic Server only supports a limited subset of Coherence configuration options, occasionally, you may require a custom Coherence operational configuration file (also known as an override file).
- Configure the Coherence Global Socket Provider
  WebLogic Server allows you to configure the Coherence global socket provider. The global socket provider can be either the WebLogic Server generated socket provider or a socket provider from the Coherence operational configuration. This may require importing a custom operational configuration file.
- WebLogic Server Secured Production Mode
  When a WebLogic Server domain is in secured production mode, then, by default, Coherence will be configured to use the WebLogic Server socket provider as the global socket provider. The default WebLogic socket provider is configured for mTLS. If you do not require mTLS, then the Coherence configuration can be overridden using WebLogic Remote Console or a WLST script.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Extended Usage Certificates

If certificates with extended usage are used, it is important to understand how this affects the different SSL/TLS configuration choices available for Coherence.

By default, WebLogic will configure Coherence to use mTLS, so the extend usage for any certificates must include both `serverAuth` and `clientAuth`. If only `serverAuth` certificates are available, Coherence must be configured to use one-way SSL/TLS as described in Configure Coherence Cluster Traffic Using One-Way SSL/TLS .

Configure Coherence Cluster Traffic Using mTLS

Coherence clusters form a peer-to-peer network where each JVM is both a server receiving connections from other cluster members, and a client connecting to other cluster members. You can configure Coherence cluster peer-to-peer communication to use SSL/TLS in a WebLogic Server domain using either WebLogic Remote Console or a WLST script.

Configure mTLS for Cluster Traffic Using WebLogic Remote Console

1. In the **Edit Tree**, go to **Environment**, then **Coherence Clusters**.
2. Click the Coherence cluster that you want to edit.
3. On the **General** tab, from the **Transport** drop-down list, select **SSL**. Or, if you want to use datagram as the transport instead of TCMP, then select **SSLUDP** instead.
4. Click **Save**.

Configure mTLS for Cluster Traffic Using WLST Script

To configure Coherence to use mTLS for cluster membership using WLST, use the following script:

> **Note**
>
> In the script below, replace:
>
> - *`DOMAIN_HOME`* with the path to your WebLogic Server domain home.
> - *`defaultCoherenceCluster`* with the name of your Coherence cluster.

```
readDomain('DOMAIN_HOME')
cd('CoherenceClusterSystemResource/CoherenceCluster/CoherenceResource/
defaultCoherenceCluster/CoherenceClusterParams/NO_NAME_0')
set('Transport', 'ssl')
updateDomain()
closeDomain()
```

Configure Coherence Cluster Traffic Using One-Way SSL/TLS

In one-way SSL/TLS, a client authenticates a server certificate, but the server does not receive a certificate from the client, so the client is anonymous. You can configure Coherence cluster

Copyright © 2008, 2026, Oracle and/or its affiliates.

traffic to use one-way SSL/TLS in a WebLogic Server domain using either WebLogic Remote Console or a WLST script.

Configure One-Way SSL for Cluster Traffic Using WebLogic Remote Console

1. In the **Edit Tree**, go to **Environment**, then **Coherence Clusters**.
2. Click the Coherence cluster that you want to edit.
3. On the **General** tab, from the **Transport** drop-down list, select **SSL**. Or, if you want to use datagram as the transport instead of TCMP, then select **SSLUDP** instead.
4. Click **Save**.
5. On the **Security** tab, from the **Client Authentication Mode** drop-down list, select **none**.
6. Click **Save**.

Configure One-Way for Cluster Traffic SSL Using WLST Script

To configure Coherence to use one-way SSL/TLS for cluster traffic using WLST, use the following script:

> **Note**
>
> In the script below, replace:
>
> - *DOMAIN_HOME* with the path to your WebLogic Server domain home.
> - *defaultCoherenceCluster* with the name of your Coherence cluster.

```
readDomain('DOMAIN_HOME')
cd('CoherenceClusterSystemResource/defaultCoherenceCluster/CoherenceResource/
defaultCoherenceCluster/CoherenceClusterParams/NO_NAME_0')
create("cohKS","CoherenceKeystoreParams")
cd('CoherenceKeystoreParams/NO_NAME_0')
set('Transport', 'ssl')
set('CoherenceClientAuth','none')
updateDomain()
closeDomain()
```

Using a Custom Coherence Operational Configuration File

As WebLogic Server only supports a limited subset of Coherence configuration options, occasionally, you may require a custom Coherence operational configuration file (also known as an override file).

A custom operational configuration file can be imported using either WebLogic Remote Console or a WLST script.

Before you can import the custom Coherence operational configuration file, you must make sure that the file exists on the Administration Server and is readable. When you import the operational configuration file, a copy of the file is placed in the *DOMAIN_HOME*/config/coherence/*CoherenceClusterName* directory, where *CoherenceClusterName* is a placeholder that represents the actual name of the Coherence cluster.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Import a Custom Operational Configuration File Using WebLogic Remote Console

1. In the **Edit Tree**, go to **Environment**, then **Coherence Clusters**.
2. Click the Coherence cluster that you want to edit.
3. On the **General** tab, click the **Import Configuration** button.
4. Enter the full path to a custom Coherence operational configuration file that is on the Administration Server.
5. Click **Save**.

Set a Custom Operational Configuration File Using a WLST Script

The file name is the relative path to the domain home. In the example below, if the full path of the operational configuration file is `/ORACLE_HOME/user_projects/domains/base_domain/config/coherence/defaultCoherenceCluster/mySSLOverride.xml`, the file name would be `config/coherence/defaultCoherenceCluster/mySSLOverride.xml`.

Use the following WLST script to configure a custom Coherence operational configuration file.

> **Note**
>
> In the script below, replace:
>
> - *DOMAIN_HOME* with the path to your WebLogic Server domain home.
> - *defaultCoherenceCluster* with the name of your Coherence cluster.
> - *sslSocketProviderName* with the name of the socket provider.

```
readDomain('DOMAIN_HOME')
cd('CoherenceClusterSystemResource/defaultCoherenceCluster/CoherenceResource/
defaultCoherenceCluster')
set('CustomClusterConfigurationFileName', 'config/coherence/
defaultCoherenceCluster/mySSLOverride.xml')
cd('CoherenceClusterParams/NO_NAME_0')
set('GlobalSocketProvider','sslSocketProviderName')
updateDomain()
closeDomain()
```

Configure the Coherence Global Socket Provider

WebLogic Server allows you to configure the Coherence global socket provider. The global socket provider can be either the WebLogic Server generated socket provider or a socket provider from the Coherence operational configuration. This may require importing a custom operational configuration file.

See Specifying a Global Socket Provider and Using a Custom Coherence Operational Configuration File.

Set the WebLogic Server Socket Provider as the Global Socket Provider in WebLogic Remote Console

1. In the **Edit Tree**, go to **Environment**, then **Coherence Clusters**.

Copyright © 2008, 2026, Oracle and/or its affiliates.

2. Click the Coherence cluster that you want to edit.
3. On the **Security** tab, enable the **Secured Production** option.
4. Click **Save**.

The name of the WebLogic Server socket provider will automatically be inserted into the **Global Socket Provider** field.

Set WebLogic Socket Provider as the Global Socket Provider Using a WLST Script

Use the following WLST script to configure the WebLogic Server socket provider as the Coherence global socket provider.

> **Note**
>
> In the script below, replace:
>
> - *DOMAIN_HOME* with the path to your WebLogic Server domain home.
> - *defaultCoherenceCluster* with the name of your Coherence cluster.

```
readDomain('DOMAIN_HOME')
cd('CoherenceClusterSystemResource/defaultCoherenceCluster/CoherenceResource/
defaultCoherenceCluster/CoherenceClusterParams/NO_NAME_0')
set('SecuredProduction', 'true')
updateDomain()
closeDomain()
```

Set a Custom Socket Provider as the Global Socket Provider in WebLogic Remote Console

You can configure Coherence to use a custom socket provider as the global socket provider using WebLogic Remote Console. The named socket provider must be configured in the Coherence operational configuration.

1. In the **Edit Tree**, go to **Environment**, then **Coherence Clusters**.
2. Click the Coherence cluster that you want to edit.
3. On the **Security** tab, enter the name of the custom socket provider in the **Global Socket Provider** field.
4. Click **Save**.

Set a Custom Socket Provider as the Global Socket Provider Using WLST Script

You can configure Coherence to use a custom socket provider as the global socket provider using WLST. The named socket provider must be configured in the Coherence operational configuration.

Use the following WLST script to set the name of the Coherence global socket provider.

Copyright © 2008, 2026, Oracle and/or its affiliates.

> **Note**
>
> In the script below, replace:
>
> - *DOMAIN_HOME* with the path to your WebLogic Server domain home.
> - *defaultCoherenceCluster* with the name of your Coherence cluster.
> - *sslSocketProviderName* with the name of the socket provider.

```
readDomain('DOMAIN_HOME')
cd('CoherenceClusterSystemResource/defaultCoherenceCluster/CoherenceResource/
defaultCoherenceCluster/CoherenceClusterParams/NO_NAME_0')
set('Transport', 'ssl')
set('GlobalSocketProvider', 'sslSocketProviderName')
updateDomain()
closeDomain()
```

WebLogic Server Secured Production Mode

When a WebLogic Server domain is in secured production mode, then, by default, Coherence will be configured to use the WebLogic Server socket provider as the global socket provider. The default WebLogic socket provider is configured for mTLS. If you do not require mTLS, then the Coherence configuration can be overridden using WebLogic Remote Console or a WLST script.

For general information on secured production mode in WebLogic Server, see Understand How Domain Mode Affects the Default Security Configuration in *Securing a Production Environment for Oracle WebLogic Server*.

- Configure Coherence for One-Way SSL/TLS in Secured Production Mode
  If the WebLogic Server domain is in secured production mode, where the WebLogic Server socket provider defaults to mTLS, you can configure Coherence to use one-way SSL/TLS instead using WebLogic Remote Console or a WLST script.
- Disable Coherence SSL/TLS in Secured Production Mode
  If the WebLogic Server domain is in secured production mode but you need Coherence to run with plain TCP, then you can configure this behavior using WebLogic Remote Console or a WLST script.

Configure Coherence for One-Way SSL/TLS in Secured Production Mode

If the WebLogic Server domain is in secured production mode, where the WebLogic Server socket provider defaults to mTLS, you can configure Coherence to use one-way SSL/TLS instead using WebLogic Remote Console or a WLST script.

Configure Coherence for One-Way SSL/TLS Using WebLogic Remote Console

1. In the **Edit Tree**, go to **Environment**, then **Coherence Clusters**.
2. Click the Coherence cluster that you want to edit.
3. On the **Security** tab, from the **Client Authentication Mode** drop-down list, select **none**.
4. Click **Save**.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Configure Coherence for One-Way SSL/TLS Using WLST Script

Use the following script to configure Coherence to use one-way SSL/TLS.

> **Note**
>
> In the script below, replace:
>
> - *DOMAIN_HOME* with the path to your WebLogic Server domain home.
> - *defaultCoherenceCluster* with the name of your Coherence cluster.

```
readDomain('DOMAIN_HOME')
cd('CoherenceClusterSystemResource/defaultCoherenceCluster/CoherenceResource/
defaultCoherenceCluster/CoherenceClusterParams/NO_NAME_0')
create("cohKS","CoherenceKeystoreParams")
cd('CoherenceKeystoreParams/NO_NAME_0')
set('CoherenceClientAuth','none')
updateDomain()
closeDomain()
```

Disable Coherence SSL/TLS in Secured Production Mode

If the WebLogic Server domain is in secured production mode but you need Coherence to run with plain TCP, then you can configure this behavior using WebLogic Remote Console or a WLST script.

> **Note**
>
> If you disable secured production mode in Coherence, it only affects Coherence and does not affect the broader aspects of secured production mode in WebLogic Server.

Disable Coherence SSL/TLS in Secured Production Mode Using WebLogic Remote Console

1. In the **Edit Tree**, go to **Environment**, then **Coherence Clusters**.
2. Click the Coherence cluster that you want to edit.
3. On the **Security** tab, turn off the **Secured Production** option.
4. Click **Save**.

Disable Coherence SSL/TLS in Secured Production Mode Using a WLST Script

Use the following script to configure Coherence to use plain TCP.

Copyright © 2008, 2026, Oracle and/or its affiliates.

> **Note**
>
> In the script below, replace:
>
> - *DOMAIN_HOME* with the path to your WebLogic Server domain home.
> - *defaultCoherenceCluster* with the name of your Coherence cluster.

```
readDomain('DOMAIN_HOME')
cd('CoherenceClusterSystemResource/defaultCoherenceCluster/CoherenceResource/
defaultCoherenceCluster/CoherenceClusterParams/NO_NAME_0')
set('SecuredProduction', 'false')
updateDomain()
closeDomain()
```

Securing Oracle Coherence Cluster Membership

The Oracle Coherence security framework (access controller) can be enabled within a Oracle WebLogic Server domain to secure access to cluster resources and operations. The access controller provides authorization and uses encryption/decryption between cluster members to validate trust. See Overview of Using an Access Controller.
In Oracle WebLogic Server, access controllers use a managed Coherence server's keystore to establish a caller's identity between Oracle Coherence cluster members. The Demo Identity keystore is used by default and contains a default SSL identity (DemoIdentity). The default keystore and identity require no setup and are ideal during development and testing. Specific keystores and identities should be created for production environments. See Configuring Keystores in *Administering Security for Oracle WebLogic Server*.

This section includes the following topics:

- Enabling the Oracle Coherence Security Framework
- Specifying an Identity for Use by the Security Framework

Enabling the Oracle Coherence Security Framework

To enable the security framework in an Oracle WebLogic Server domain using WebLogic Remote Console:

1. In the **Edit Tree**, go to **Environment**, then **Coherence Clusters**.
2. Click the Coherence cluster that you want to edit, then select the **Security** tab.
3. Turn on the **Security Framework Enabled** option.
4. Click **Save**.

Specifying an Identity for Use by the Security Framework

The Oracle Coherence security framework requires a principal (identity) when performing authentication. The SSL Demo Identity keystore is used by default and contains a default SSL identity (DemoIdentity). The SSL Demo keystore and identity are typically used during development. For production environments, you should create an SSL keystore and identity. For example, use the Java `keytool` utility to create a keystore that contains an `admin` identity:

Copyright © 2008, 2026, Oracle and/or its affiliates.

```
keytool -genkey -v -keystore ./keystore.jks -storepass password -alias admin
-keypass password -dname CN=Administrator,O=MyCompany,L=MyCity,ST=MyState
```

> **Note**
>
> If you create an SSL keystore and identity, you must configure Oracle WebLogic Server to use that SSL keystore and identity. In addition, the same SSL identity must be located in the keystore of every managed Coherence server in the cluster. Use the Keystores and SSL tabs on the **Environment**: **Servers**: *myServer*: **Security** tab for a managed Coherence server to configure a keystore and identity.

To override the default SSL identity and specify an identity for use by the security framework:

1. In the **Edit Tree**, go to **Environment**, then **Coherence Clusters**.
2. Click the Coherence cluster that you want to edit, then select the **Security** tab.
3. Make sure the **Security Framework Enabled** option is enabled.
4. Optional: Turn on the **Global Socket Provider** option. If you enable this option, Coherence uses the Oracle WebLogic Server's SSL as its global socket provider. For more information about configuring global SSL socket provider for Coherence, see Configuring a Cluster-Side Extend Proxy SSL Socket Provider.
5. In the **Client Authentication Mode** field, specify the client authentication mode for SSL. The valid values are:
   - `none`
   - `required`
   - `wanted`

   The default value is `none`.
6. In the **Private Key Pass Phrase** field, enter the password for the identity.
7. Click **Save**.

Authorizing Oracle Coherence Caches and Services

Oracle WebLogic Server authorization can be used to secure Oracle Coherence resources that run within a domain. In particular, different roles and policies can be created to control access to caches and services. Authorization is enabled by default and the default authorization policy gives all users access to all Oracle Coherence resources. See Overview of Securing WebLogic Resources in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*. Authorization roles and policies are explicitly configured for caches and services. You must know the cache names and service names that are to be secured. In some cases, inspecting the cache configuration file may provide the cache names and service names. However, because of wildcard support for cache mappings in Oracle Coherence, you may need to consult an application developer or architect that knows the cache names being used by an application. For example, a cache mapping in the cache configuration file could use a wildcard (such as `*` or `dist-*`) and does not indicate the name of the cache that is actually used in the application.

Copyright © 2008, 2026, Oracle and/or its affiliates.

> **Note**
>
> Deleting a service or cache resource does not delete roles and policies that are defined for the resource. Roles and policies must be explicitly deleted before deleting a service or cache resource.

This section includes the following topics:

- Specifying Cache Authorization
- Specifying Service Authorization

Specifying Cache Authorization

Oracle WebLogic Server authorization can be used to restrict access to specific Oracle Coherence caches. To specify cache authorization:

1. In the **Edit Tree**, go to **Environment**, then **Coherence Clusters**, then *myCoherenceCluster*, then **Coherence Caches**.
2. Click **New**.
3. In the **Name** field, enter a name for the Coherence cache. The name of the cache must *exactly* match the name of the cache used in an application.
4. Click **Create** and commit your changes.
5. Define security roles and policies that are scoped to the Coherence cache. See Create a Scoped Role in *Oracle WebLogic Remote Console Online Help*.

   For example, you can create a policy that allows specific users to access the cache. The users can be selected based on their membership in a global role, or a Coherence-specific scoped role can be created and used to define which users can access the cache. See Overview of Securing WebLogic Resources in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

Specifying Service Authorization

Oracle WebLogic Server authorization can be used to restrict access to Oracle Coherence services. Specifying authorization on a cache service (for example a distributed cache service) affects access to all the caches that are created by that service.

To specify service authorization:

1. In the **Edit Tree**, go to **Environment**, then **Coherence Clusters**, then *myCoherenceCluster*, then **Coherence Services**.
2. Click **New**.
3. In the **Name** field, enter a name for the Coherence service. The name of the service must *exactly* match the name of the service used in an application.

Copyright © 2008, 2026, Oracle and/or its affiliates.

> **Note**
>
> The exact name must include the scope name as a prefix to the service name. The scope name can be explicitly defined in the cache configuration file or, more commonly, taken from the deployment module name. For example, if you deploy a GAR named `contacts.gar` that defines a service named `ContactsService`, then the exact service name is `contacts:ContactsService`.

4. Click **Create** and commit your changes.
5. Define security roles and policies that are scoped to the Coherence service. See Create a Scoped Role in *Oracle WebLogic Remote Console Online Help*.

   For example, you can create a policy that allows specific users to access the service. The users can be selected based on their membership in a global role, or a Coherence-specific scoped role can be created and used to define which users can access the service. See Overview of Securing WebLogic Resources in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

Securing Extend Client Access with Identity Tokens

Identity tokens are used to protect against unauthorized access to an Oracle Coherence cluster through an Oracle Coherence proxy server. Identity tokens are used by local (within WebLogic Server) extend clients and remote (outside of WebLogic Server) Java, C++, and .NET extend clients.

Only clients that pass a valid identity token are permitted to access cluster services. If a `null` identity token is passed (a client connecting without being within the scope of a `Subject`), then the client is treated as an Oracle WebLogic Server anonymous user. The extend client is able to access caches and services that the anonymous user can access.

> **Note**
>
> Upon establishing and identity, an authorization policy should be used to restrict that identity to specific caches and services. See Authorizing Oracle Coherence Caches and Services.

Identity token security requires an identity transformer implementation that creates an identity token and an identity asserter implementation that validates the identity token. A default identity transformer implementation (`DefaultIdentityTransformer`) and identity asserter implementation (DefaultIdentityAsserter) are provided. The default implementations use a `Subject` or `Principal` as the identity token. However, custom implementations can be created as required to support any security token type (for example, to support Kerberos tokens). See Using Identity Tokens to Restrict Client Connections.

This section includes the following topics:

- Enabling Identity Transformers for Use in Oracle WebLogic Server
- Enabling Identity Asserters for Use in Oracle WebLogic Server

Copyright © 2008, 2026, Oracle and/or its affiliates.

Enabling Identity Transformers for Use in Oracle WebLogic Server

An identity transformer associates an identity token with an identity. For local (within Oracle WebLogic Server) extend clients, the default identity transformer cannot be replaced. The default identity transformer passes a token of type `weblogic.security.acl.internal.AuthenticatedSubject` representing the current Oracle WebLogic Server user.

For remote (outside of Oracle WebLogic Server) extend clients, the identity transformer implementation class must be included as part of the application's classpath and the fully qualified name of the implementation class must be defined in the client operational override file. See Enabling a Custom Identity Transformer. The following example enables the default identity transformer:

```
...
<security-config>
   <identity-transformer>
      <class-name>
         com.tangosol.net.security.DefaultIdentityTransformer</class-name>
   </identity-transformer>
</security-config>
...
```

Remote extend clients must execute cache operations within the `Subject.doAs` method. For example,

```
Principal principal = new WLSUserImpl("user");
Subject subject = new Subject();
subject.getPrincipals().add(principal);

Subject.doAs(subject, new PrivilegedExceptionAction()
   {
      NamedCache cache = CacheFactory.getCache("mycache");
      ...
```

Enabling Identity Asserters for Use in Oracle WebLogic Server

Identity asserters must be enabled for an Oracle Coherence cluster and are used to assert (validate) a client's identity token. For local (within Oracle WebLogic Server) extend clients, an identity asserter is already enabled for asserting a token of type `weblogic.security.acl.internal.AuthenticatedSubject`.

For remote (outside of Oracle WebLogic Server) extend clients, a custom identity asserter implementation class must be packaged in a GAR. However, an identity asserter is not required if the remote extend client passes `null` as the token. If the proxy service receives a non-null token and there is no identity asserter implementation class configured, a `SecurityException` is thrown and the connection attempt is rejected.

You can use WebLogic Remote Console or WLST to enable an identity asserter for a cluster.

- If using WebLogic Remote Console, perform the following steps:
  1. In the **Edit Tree**, go to **Environment**, then **Coherence Clusters**, then *myCoherenceCluster*, then **Coherence Identity Asserter**.
  2. In the **Class Name** field, enter the fully qualified name of the asserter class. For example, to use the default identity asserter, enter `com.tangosol.net.security.DefaultIdentityAsserter`.

Copyright © 2008, 2026, Oracle and/or its affiliates.

3. Click **Save**.
4. If there are any arguments, open the **Identity Asserter Constructor Arguments** node and click **New** to add class constructor arguments.
5. Click **Save** and then commit your changes.

- If using WLST, perform the following steps:

  Invoke WLST and connect to the domain. Then, configure an identity asserter. Use the script below as an example:

  In the script below, replace:

  – *DOMAIN_HOME* with the path to your WebLogic Server domain home.

  – *defaultCoherenceCluster* with the name of your Coherence cluster.

```
readDomain('DOMAIN_HOME')
cd('CoherenceClusterSystemResource/defaultCoherenceCluster/
CoherenceResource/defaultCoherenceCluster/CoherenceClusterParams/
NO_NAME_0')
set('SecurityFrameworkEnabled', 'true')
cd('CoherenceIdentityAsserter/NO_NAME_0')
set('ClassName',"com.tangosol.net.security.DefaultIdentityAsserter")
updateDomain()
closeDomain()
```

Restart the cluster servers or redeploy the GAR for the changes to take effect.

Copyright © 2008, 2026, Oracle and/or its affiliates.

16

Upgrading and Operating a Secured Deployment

Coordinate artifacts, identities, configuration, diagnostics, and recovery across secured members and clients.
Security adoption is an application and operational rollout. Keep a recorded set of configuration, artifact, native-library, client, and credential versions. Use Rolling Upgrade to Hardened Security for the profile change and Assembling Secure Deployment Recipes for the combined service configuration.

Observe the boundary that makes the decision: transport, token assertion, class materialization, executable installation, operation authorization, resource limit, or stored/imported input. Security Errors and Diagnostic Signals maps the principal signals to corrective actions. Prefer a narrow fix to the producing artifact, identity mapping, or configuration over a global relaxation.

Routine operation

- Rotate certificates and credentials with their actual provider/channel lifecycle and verify fresh connections.
- Recheck policy after adding a service, client language, module, application class, native binary, or persisted subscriber definition.
- Protect diagnostic artifacts and report output. Keep bounded logs and request IDs without sensitive payloads.
- Exercise backup restoration and startup under the intended policy in an isolated environment.
- Measure authorization and policy overhead with representative load and failure behavior, including credential-service outages.
- Rolling Upgrade to Hardened Security
- Assembling Secure Deployment Recipes

Rolling Upgrade to Hardened Security

Prepare for the hardened default before upgrading, using an explicitly selected compatibility phase only where the migration plan requires one.

Preparation

Inventory each client/member, enabled listener, serializer, remote executable family, configured identity/authorization chain, persistence/transaction workflow, and administrative tool. Record explicit settings as well as defaults. Include scrapers, health administration, Reporter schedules, JMX discovery, optional RAG imports, and native HNSW binaries.

Starting with Coherence 26.1.0.0, an unset `coherence.security.mode` selects `hardened` from the first startup. Validate the application and service policies before that startup. If an older deployment needs a temporary compatibility phase, explicitly set `coherence.security.mode=compatibility` on the affected upgraded processes before starting

Copyright © 2008, 2026, Oracle and/or its affiliates.

them. Preserve a deliberately hardened deployment. The runtime `coherence.mode=prod` setting remains independent of this choice.

Stage the application and service policies

1. A 26.1 Java client using `compute` or `replaceAll` against 14.1.2 or 15.1.1 servers must explicitly select `coherence.security.mode=compatibility` on the client before Coherence initializes. The hardened client can select newer named processors that those older servers do not recognize. Verify the supported client/server and serializer combinations; client interoperability and the profile of upgraded 26.1 servers are separate deployment settings.

   Generate and deploy per-artifact security configuration for serialized types, remote executables, and custom lambda targets. Verify the receiving classloader sees every required resource.

2. If a temporary compatibility phase is selected, exercise real workflows with `coherence.security.mode=compatibility` explicitly configured. Review executable `would_reject` signals and separate management/Reporter/JMX warnings. These are incomplete readiness evidence; they do not inventory every accepted class or prove lambda/decoder readiness.

3. In an isolated environment, run the same client/server workloads with hardened policy. Set `coherence.remote.dynamic.unauthenticated=deny` explicitly if remote dynamic code is outside the deployment policy; the current integration default remains `allow` in either profile. Include legitimate largest payloads, denied operations, failover/reconnection, and replay/recovery.

4. Configure TLS, supported identity assertion, storage/service authorization, endpoint credentials, query/JSON aliases, finite serializer limits, approved Reporter paths, and static JMX publication as needed.

5. Coordinate credentials and clients before changing endpoint defaults. Verify metrics scraping, management tooling, and health administration on every relevant member.

Mixed-version data and capabilities

Transaction result/XID bridges retain their documented narrow type universes in both profiles. Drain or test custom XA XIDs before crossing the tightened bridge. A new transaction-binding reader uses typed fields and treats a legacy-only binding map as empty. Do not send binding-dependent traffic across that boundary without a coordinated plan.

Persisted topic subscriber metadata is rechecked on replay. Hardened policy defaults to rejection of executable-policy drift; `warn-allow` is an explicit migration bridge, not proof that the stored definition is still approved. Keep the defining classes and generated policy available on recovery readers.

Proof-required subject/senior workflows have physical-recipient and capability prerequisites beyond an ordinary patch rolling upgrade. Use Configuring Peer Proof Prerequisites and do not require proof merely because one member is ready.

Activation and recovery

Validate hardened behavior before the first 26.1.0.0-or-later startup unless a temporary compatibility phase has been explicitly selected. To finish that phase, remove `coherence.security.mode=compatibility` or set `hardened` explicitly after preparing the required members and clients. Keep consistent profiles for processes serving the same endpoint, and verify both success and intended refusals at each stage.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Prepare rollback before activation: configuration, application artifacts, credential overlap, data compatibility, and explicitly required independent policies. Returning to compatibility does not bypass built-in deny rules, malformed-input checks, explicit finite limits, health-mutator authentication, bridge filters, or explicitly required senior proof. A runtime mode change is not a security rollback.

Assembling Secure Deployment Recipes

Combine the shared TLS, identity, authorization, and application-policy recipes into complete service deployments.

A secured accounts cache with Extend

Create the protected TLS material and operational socket/password providers from Configuring TLS with Protected Password Sources. Configure the server JAAS identity asserter, protected JAAS file, and login-module dependencies from Authenticating Credentials with JAAS. Register `accounts-policy` using Implementing an Account Storage Authorizer. Merge these sections into one operational override in schema order.

The following complete server cache configuration exposes only the intended cache mapping and an mTLS Extend proxy. The named `server-tls` provider and `accounts-policy` declaration must already exist in that server's operational context.

```
<cache-config xmlns="http://xmlns.oracle.com/coherence/coherence-cache-
config">
  <caching-scheme-mapping>
    <cache-mapping>
      <cache-name>accounts</cache-name>
      <scheme-name>accounts-store</scheme-name>
    </cache-mapping>
  </caching-scheme-mapping>
  <caching-schemes>
    <distributed-scheme>
      <scheme-name>accounts-store</scheme-name>
      <service-name>AccountsService</service-name>
      <backing-map-scheme>
        <storage-authorizer>accounts-policy</storage-authorizer>
        <local-scheme/>
      </backing-map-scheme>
      <autostart>true</autostart>
    </distributed-scheme>
    <proxy-scheme>
      <scheme-name>accounts-proxy</scheme-name>
      <service-name>AccountsProxy</service-name>
      <acceptor-config>
        <tcp-acceptor>
          <socket-provider>server-tls</socket-provider>
          <address-provider>
            <local-address><address>0.0.0.0</address><port>20000</port></
local-address>
          </address-provider>
        </tcp-acceptor>
      </acceptor-config>
      <autostart>true</autostart>
    </proxy-scheme>
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

```
  </caching-schemes>
</cache-config>
```

The client uses its own `client-tls` provider, the explicit credential transformer accepted by the server, and this cache configuration:

```
<cache-config xmlns="http://xmlns.oracle.com/coherence/coherence-cache-
config">
  <caching-scheme-mapping>
    <cache-mapping>
      <cache-name>accounts</cache-name>
      <scheme-name>accounts-client</scheme-name>
    </cache-mapping>
  </caching-scheme-mapping>
  <caching-schemes>
    <remote-cache-scheme>
      <scheme-name>accounts-client</scheme-name>
      <service-name>AccountsProxy</service-name>
      <initiator-config>
        <tcp-initiator>
          <socket-provider>client-tls</socket-provider>
          <remote-addresses>
            <socket-address><address>cache.example.com</address><port>20000</
port></socket-address>
          </remote-addresses>
        </tcp-initiator>
      </initiator-config>
    </remote-cache-scheme>
  </caching-schemes>
</cache-config>
```

Use a unique cluster name for the isolated recipe deployment and choose the server/client operational and cache configuration files explicitly. Keep InvocationService and Concurrent proxies disabled unless the use case needs them. Package any custom payload/executable resources. With this mTLS example, every intended client must have a trusted client certificate as well as the configured application credential.

Verify writer success, reader read-only behavior, missing/invalid credentials, an untrusted/mismatched certificate, and a denied executable. Test reconnection and storage failover. These checks distinguish transport, identity, caller authorization, and executable policy.

An authenticated gRPC application

Use the same protected storage scheme and server identity setup with Configuring a Secure gRPC Proxy. Configure the product client with Connecting a Java gRPC Client Securely and the appropriate client cache mapping. Require transport on both sides, set diagnostic exposure explicitly, and approve only the intended serializer names. Verify the same authorized/denied operations through the actual gRPC protocol and client version.

Management, metrics, and health

Choose administrative/scraper identities, protected TLS providers, network placement, and explicit HTTP auth settings. Use Securing Management and Health Services and Configuring an Authenticated TLS Metrics Scrape. Leave health mutators disabled unless operationally required; if enabled, test their separate authentication without breaking read-only probes.

Copyright © 2008, 2026, Oracle and/or its affiliates.

If JMXMP is enabled, use its authenticator, process-wide filter review, and provider-specific secure transport. If JMX/RMI discovery is used, configure static publication before hardened activation. Exercise the real JMX/Reporter schedules and monitoring tools in addition to HTTP reads.

Copyright © 2008, 2026, Oracle and/or its affiliates.

A

Security Configuration Reference

Locate profile-sensitive defaults, explicit feature controls, and the configuration that owns each policy.
This reference describes the controls used by this guide. Match them to the target distribution and supported modules; consult release/patch documentation for another code line. Explicit application configuration can differ from shipped marker-aware defaults. Set startup-resolved policy before starting the relevant process and use the owning recipe to coordinate changes.

Starting with Coherence 26.1.0.0, the hardened column/value applies when `coherence.security.mode` is unset. Compatibility behavior below requires explicitly selecting `coherence.security.mode=compatibility`; feature-specific overrides retain their documented precedence.

Application and Execution Controls

Property	Type	Default	Description	See
`coherence.mode`	enum: `eval`, `dev`, `development`, `prod`, `production`	`dev`	Selects the released runtime/license mode. It does not enable the compatibility-sensitive serialization and remote-execution gates. Blank, unknown, `legacy`, and `legacy-compatibility` values fail fast.	Runtime Modes
`coherence.security.mode`	enum: `compatibility`, `hardened`	`hardened` (starting with 26.1.0.0)	Selects the global security profile. Starting with Coherence 26.1.0.0, unset selects `hardened` and enables compatibility-sensitive checks. Explicit `compatibility` preserves the older behavior and shadow telemetry where available. The removed pre-release `coherence.security.hardened` property is not an alias.	Runtime Modes

Copyright © 2008, 2026, Oracle and/or its affiliates.

Property	Type	Default	Description	See
`coherence.security.subject.validation.ttl`	duration	`10s`	Controls how long Standard security caches a successfully validated subject. Unitless values are milliseconds. Valid values are from 1 through `Integer.MAX_VALUE` milliseconds; malformed or out-of-range values use the 10-second default with a warning. This operator control is independent of `coherence.mode` and `coherence.security.mode`.	Standard Subject Validation Cache
`coherence.remote.dynamic.unauthenticated`	enum: `allow`, `deny`	`allow` in both profiles in the current integration implementation; confirm target-release policy before publication.	Explicit `deny` refuses wire-arriving DYNAMIC lambda bytecode, reflective `MethodInvocationProcessor`, script execution, and dynamic remote installs governed by the shared gate. Dynamic installs may pass without a named executable entry when allowed. Coherence Config also resolves `COHERENCE_REMOTE_DYNAMIC_UNAUTHENTICATED`. Does not bypass lambda-target checks, bytecode/MIP deny rules, Graal HostAccess, or caller authorization.	Lambda Serialization, MethodInvocationProcessor / ScriptProcessor

Copyright © 2008, 2026, Oracle and/or its affiliates.

Property	Type	Default	Description	See
`coherence.invocation.enabled`	boolean	Compatibility: enabled; hardened: disabled, unless XML or this property explicitly overrides.	Controls exposure of the Coherence*Extend InvocationService proxy. An explicit system property overrides operational `<invocation-service-proxy><enabled>`; explicit XML overrides the profile default. Enabled remote invocation still requires executable classification.	InvocationService
`coherence.concurrent.extend.enabled`	boolean	security mode `compatibility`: `true`; security mode `hardened`: `false`	Overrides whether the Coherence Concurrent `ConcurrentProxy` Extend service is exposed. When unset, the shipped `ConcurrentProxy` autostart marker resolves to the security-mode-aware default. An explicit cache-config autostart override without the shipped system-property marker wins when the property is absent. This property does not bypass `@Remote.Executable` enforcement when the proxy is enabled.	Coherence Concurrent Executor

Copyright © 2008, 2026, Oracle and/or its affiliates.

Property	Type	Default	Description	See
`coherence.topics.persisted.policy-drift`	enum: `reject`, `warn-allow`	security mode `compatibility`: `warn-allow`; security mode `hardened`: `reject`	Controls persisted topic subscriber filter/extractor policy drift on replay. `reject` refuses persisted metadata whose class is no longer executable; `warn-allow` logs a deduped WARN and allows replay for migration. The property is resolved through Coherence `Config`, so the `COHERENCE_TOPICS_PERSISTED_POLICY_DRIFT` environment variable alias is also honoured.	Topic Subscriber Filter / Extractor Install
`coherence.serialization.allowed`	class-fqn-list	empty	Adds exact class names or package wildcards, such as `com.acme.*`, to the runtime serialization allowlist. It supplements generated security config and does not bypass deny-listed gadget classes.	Generated `security-config.xml`
`coherence.lambda.bytecode.allow`	class-fqn-list	empty	Incident-response override for specific class references in the lambda bytecode deny-list. Structural bytecode rules such as native methods still apply.	Error Message Reference
`coherence.security-config.skip`	boolean	`false`	Maven plugin switch that skips `security-config.xml` generation for the current build.	Maven

Copyright © 2008, 2026, Oracle and/or its affiliates.

Property	Type	Default	Description	See
coherence.lambdas	enum: dynamic, static	dynamic	Lambda serialization mode that has shipped in prior Coherence releases. Set to static when migrating production applications away from DYNAMIC lambda bytecode at the cost of lockstep client/server code-version coordination.	Lambda Serialization

Protocol and Administrative Controls

Property	Default or profile behavior	Configuration note
coherence.serializer.limit.maxContainerBytes	Compatibility: unbounded default; hardened: 64 MiB	Global byte limit default; serializer/service XML can refine it.
coherence.serializer.limit.maxElements	Compatibility: unbounded default; hardened: 262144	Array/collection/object count default for supported consumers.
coherence.serializer.limit.maxMapEntries	Compatibility: unbounded default; hardened: 131072	Map-entry count default for supported consumers.
coherence.grpc.server.secure-transport	optional	Set required to refuse insecure server transport configuration.
coherence.grpc.channels.<name>.secure-transport	optional	Product Java-client named channel policy; use required with a real TLS provider.
coherence.grpc.auth	none	Explicit basic enables gRPC Basic assertion; does not by itself mandate TLS at startup.
coherence.grpc.serializer.allowlist	empty additional set	Additional client-selectable serializer names in hardened operation.
coherence.grpc.channelz	auto	auto is enabled in compatibility and disabled in hardened; explicit enabled/disabled override.
coherence.grpc.error-disclosure	auto	auto is diagnostic in compatibility and safe in hardened; explicit safe/diagnostic override.
coherence.management.http.auth	Compatibility: none; hardened: basic	Shipped marker-aware default; explicit operator method configuration takes precedence.
coherence.metrics.http.auth	Compatibility: none; hardened: basic	Update scrapers before changing profile. Explicit none remains available and warns.
coherence.health.http.mutators.enabled	false	Independent opt-in for supported PUT suspend/resume routes.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Property	Default or profile behavior	Configuration note
`coherence.health.http.auth`	Enabled mutators: basic when absent/blank	basic, cert, cert+basic supported; none/invalid/missing required identity fail closed.
`coherence.management.report.remote.allowed`	empty	Exact approved HTTP(S) report-source authorities; profile-sensitive treatment of unapproved sources.
`coherence.reporter.output.directory`	.	Canonical report output root; hard malformed/traversal checks remain separate.
`coherence.management.remote.registryport`	unset	Configure static JMX publication to avoid dynamic /stub URL rejection under hardened policy.
`coherence.management.remote.authenticator`	unset	JMXAuthenticator class for remote management and JMXMP fallback; does not itself configure TLS.
`coherence.management.serverfactory`	Deployment-specific	Select com.tangosol.net.management.JmxmpServer only when that connector is intended.
`coherence.jmxmp.host`	127.0.0.1	JMXMP listener address; wider bind is an explicit network exposure decision.
`coherence.jmxmp.port`	9000	JMXMP listener port; authentication and filter requirements still apply.
`coherence.jmxmp.authenticator`	unset	Usable JMXAuthenticator required directly or through the documented remote-management fallback.
`coherence.security.hostname.verification`	Shipped SSL-provider marker: allow with profile-sensitive resolution	Use explicit default-verifier configuration in secure recipes; custom providers/verifiers retain their contracts.
`coherence.security.config`	DefaultController resource	Location of the DefaultController XML resource, including its signature algorithm configuration.
`coherence.security.peer.senior-metadata-proof.required`	false	Explicit true fails closed in both security profiles; requires compatible production proof setup.

RAG Component Controls

Property	Default	Configuration note
`coherence.rag.security.admin-role`	admin	Administrative role required for protected mutations/imports.
`coherence.rag.config.write.allowed-properties`	model.embedding,model.chat,model.scoring,coherence.rag.default.parser	Allowed runtime write keys; sensitive/protected fields retain deny policy.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Property	Default	Configuration note
`coherence.rag.security.huggingface.allowed-models`	empty	Empty permits/warns in compatibility and rejects downloads in hardened; configured values constrain both.
`coherence.rag.import.allowed-schemes`	empty	Explicitly approve each supported import scheme.
`coherence.rag.import.file.allowed-roots`	empty	Canonical local import roots; filesystem contract and final-entry race still apply.
`coherence.rag.import.http.allowed-hosts`	empty	Allowed origin hosts; every redirect/address is validated.
`coherence.rag.import.http.allow-private-addresses`	false	Does not waive host approval or permit link-local metadata destinations.
`coherence.rag.import.connect-timeout-ms`	10000	Bounded connection timeout.
`coherence.rag.import.read-timeout-ms`	30000	Bounded response-read timeout.
`coherence.rag.import.max-bytes`	52428800	Maximum accepted document response bytes.
`coherence.rag.import.s3.allowed-buckets`	empty	Approved S3 locations; configure scheme and provider IAM separately.
`coherence.rag.import.azure.blob.allowed-containers`	empty	Approved Azure Blob locations.
`coherence.rag.import.gcs.allowed-buckets`	empty	Approved Google Cloud Storage locations.
`coherence.rag.import.oci.os.allowed-locations`	empty	Approved OCI Object Storage locations.

XML and Artifact-Owned Policy

Configuration	Owner	Important boundary
`META-INF/coherence/security-config.xml`	Each contributing application JAR	Allowed/executable/lambda-target entries are merged; built-in deny rules remain effective.
Serializer `<limits>`	Operational template and cache service serializer	Missing values inherit; system-property values override the element fallback; custom consumers must implement the policy.
REST `<direct-query enabled="true">`	Resource query configuration	An empty element does not enable direct querying.
REST `<expression-aliases>`	Server/global or resource REST configuration	Named sort/projection/aggregator/processor expressions; not caller authorization.
`<storage-authorizer>`	Backing-map scheme referencing an operational declaration	Absent remains opt-in; auditing/strict-auditing do not supply a business role policy.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Configuration	Owner	Important boundary
`subject-proof-required="true"`	Operational storage-authorizer declaration	Subject proof is enforced for the configured route under hardened policy with compatible provider prerequisites.
`<peer-proof>`	Operational security configuration	Provider material and authority configuration; required policy is separate and cannot supply missing trust.
Named JAAS entries	Protected JAAS configuration and module classpath	Endpoint entry names must match the actual HTTP/Coherence/application integration.
Named password providers	Operational provider registry and consuming component	Secret-source lifecycle and failure behavior are provider-specific.

Copyright © 2008, 2026, Oracle and/or its affiliates.

B

Security Behavior Matrix

Distinguish security-profile defaults, independent hard protections, and controls requiring explicit opt-in.
Runtime/license mode and security profile are independent. Starting with Coherence 26.1.0.0, an unset `coherence.security.mode` selects `hardened`. Select `compatibility` explicitly to use its column below. Explicit safe/deny settings and feature-specific policy retain their own contracts.

Compatibility-Sensitive Behavior

Security behavior change	Compatibility profile (explicit)	Hardened profile (default starting with 26.1.0.0)	Separate switch / override
Serialization allowlist default	Does not require classes to be allowlisted, except denylisted classes still fail.	Enforces serialization allowlist.	Security config/manual allowlist can allow classes; denylist still blocks.
Serializer size/count limits	Unlimited/default-unbounded behavior allowed; threshold excess warns.	Finite defaults; `unlimited` values rejected.	`coherence.serializer.limit.maxContainerBytes`, `coherence.serializer.limit.maxElements`, `coherence.serializer.limit.maxMapEntries`; XML `<limits>`.
XML external-entity protections	Warns/falls back/ continues if required parser protections cannot be applied.	Fails closed if protections cannot be applied.	No separate feature switch found.
Dynamic remote payload default	Unset `coherence.remote.dynamic.unauthenticated` currently resolves to `allow` during integration.	The same current integration default is `allow`. Hardened alone does not select dynamic default-deny.	`coherence.remote.dynamic.unauthenticated=deny|allow`.
Lambda bytecode missing security config	Missing `security-config.xml` is `would_reject` shadow telemetry.	Missing config rejects.	Add `@Remote.Allowed` / generated security config. Other lambda denylist/invalid checks can still reject independently.
Remote executable policy	Missing named-class classification records `would_reject` and allows.	Missing named-class classification rejects. Dynamic installs may instead pass the shared dynamic gate; persisted replay is separate.	Classify named executable classes; set `coherence.remote.dynamic.unauthenticated=deny` explicitly to refuse dynamic installs.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Security behavior change	Compatibility profile (explicit)	Hardened profile (default starting with 26.1.0.0)	Separate switch / override
Function-based cache processors	Returns historical lambda/anonymous processors.	Uses classified wrapper processors needed for executable policy enforcement.	No separate switch.
Persisted topic policy drift	Default is `warn-allow`.	Default is `reject`.	`coherence.topics.persisted.policy-drift=reject|warn-allow`.
REST auth fail-closed behavior	Historical fail-open behavior when REST auth context is engaged but unusable.	Requires usable principal; otherwise returns 401.	Normal REST auth configuration controls whether auth is engaged.
REST pass-through resources	Auto-publishes unconfigured pass-through cache resources.	Requires explicit REST resource configuration.	Explicit REST resource config allows.
REST direct query	Ordinary HTTP cache operations still require direct-query enablement. The legacy SSE path retains its compatibility bypass and logs.	Requires explicit resource enablement and the supported safe query subset for HTTP and SSE.	Query config direct-query enablement.
REST URL expressions	Raw sort/projection/aggregator/processor expressions allowed.	Requires configured aliases, except approved built-in numeric processor forms.	REST expression alias config.
JSON class metadata	Historical Genson class-name/fallback behavior retained.	Class literals denied; `@class` metadata must resolve via configured aliases; denied metadata rejected at hardened sites.	Configured JSON type aliases.
gRPC remote serializer format	Non-allowlisted client formats only record `would_reject`.	Non-default/non-allowlisted/unsafe formats reject.	`coherence.grpc.serializer.allowlist`.
gRPC diagnostics `auto` defaults	`auto` keeps richer diagnostics: Channelz enabled, diagnostic error detail.	`auto` hardens: Channelz disabled, safe error disclosure.	`coherence.grpc.channelz=enabled|disabled|auto`, `coherence.grpc.error-disclosure=safe|diagnostic|auto`.
gRPC Basic auth over cleartext	Allowed with warning.	Rejected as unauthenticated.	Use TLS/in-process; auth method controlled by `coherence.grpc.auth`.
Management/metrics HTTP auth defaults	Default auth is `none`.	Default auth is `basic`.	Use explicit supported HTTP methods, including basic, cert, cert+basic, or none where the endpoint permits it.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Security behavior change	Compatibility profile (explicit)	Hardened profile (default starting with 26.1.0.0)	Separate switch / override
Management TCMP unsafe request checks	Compatibility-shadow reasons are logged/ allowed.	Unsafe management TCMP requests reject.	Safe allowlists/policies still allow.
Reporter source/output/ ObjectName checks	Compatibility-shadow reasons are logged/ allowed.	Unsafe Reporter paths, sources, protocols, broad ObjectNames reject.	Reporter approved roots/URL/object policies.
Management JMX publish URL	Dynamic RMI stub publish URL warns and is allowed.	Dynamic RMI stub publish URL rejects.	Use a safe/static JMX service URL.
TLS hostname verification	Null/`allow` verifier behavior preserved with `would_reject` logging.	Default verifier enforced; explicit unsafe `allow` rejected except shipped fallback maps to default verifier.	`coherence.security.hostname.verification`; custom verifier still works.
DefaultController signature algorithm	Legacy weak signature algorithm allowed with warning.	Modern default; weak algorithms rejected.	External config can set a strong algorithm.
DefaultController subject binding	Preserves broader multi-principal compatibility once signer is verified.	Binds authorization to verified signer principal and rejects unsafe multi-principal/unbound cases.	Certificate/keystore config.
Subject proof for storage access	Not enforced just because a route requests proof.	Enforced only for proof-required storage-authorizer routes.	Requires `subject-proof-required="true"` on storage authorizer.
Federation direct-connect endpoint validation	Allows unverified endpoint after configured-participant validation, with `would_reject` telemetry.	Rejects unverified endpoint.	Configure expected direct-connect endpoints.
RAG HuggingFace model download empty allowlist	Empty allowlist warns and permits unrestricted downloads.	Empty allowlist rejects.	`coherence.rag.security.huggingface.allowed-models`.
Extend InvocationService proxy default	Default enabled.	Default disabled.	`coherence.invocation.enabled=true|false` or XML `<invocation-service-proxy><enabled>`.
Concurrent Extend proxy default	Shipped default autostart remains enabled.	Shipped default autostart disabled.	`coherence.concurrent.extend.enabled=true|false`.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Independent Protections and Requirements

Security feature / control	Activation / switch	Relationship to `coherence.security.mode`	Notes
Senior metadata proof / senior broadcast proof	`coherence.security.peer.senior-metadata-proof.required=true`	Independent. Disabled by default in both modes; when explicitly required, it fails closed in both modes.	`coherence.security.mode=hardened` does not implicitly require senior metadata proof.
Serialization built-in denylist	Built-in `SerializationAllowlist` exact/prefix deny entries.	Independent. Denylisted classes reject in both modes.	This hard deny happens before the mode-dependent allowlist-enforcement decision.
Object input filters and bridge-local filters	JVM-wide `ObjectInputFilter.Config.getSerialFilter()` plus Coherence bridge filters such as transfer metadata, persistent-store info, and management publish filters.	Independent. A rejection from the default, JVM, or bridge filter rejects in both modes.	These filters intersect; any rejecting delegate rejects the class.
Wire `ClassIdentity` package allowlist	Built-in `ClassIdentityAllowlist`.	Independent. Disallowed wire-supplied class-identity packages reject in both modes.	This protects class-resolution metadata separately from the security-mode allowlist default.
Lambda bytecode hard checks	Built-in lambda bytecode denylist resources; emergency `coherence.lambda.bytecode.allow` can remove denylist entries.	Independent for denylisted class/method references, invalid bytecode, unsafe bytecode, and disallowed lambda targets.	Only missing security config shadowing and the dynamic-remote default are mode-dependent.
RAG configured allowlists	RAG allowlist configuration, including `coherence.rag.security.huggingface.allowed-models` when non-empty.	Independent once configured. Unallowlisted values reject in both modes.	Only the empty HuggingFace model-download allowlist default is mode-dependent.
Reporter hard input-shape rejects	Built-in Reporter validation for empty/unsafe names, invalid URIs, output URI schemes, path traversal, invalid output paths, and related malformed inputs.	Independent. These reject in both modes.	The main matrix covers Reporter cases that shadow in compatibility mode, such as outside-root or unapproved remote sources.
Management REST DiagnosticCommand allowlist	Built-in JFR DiagnosticCommand allowlist.	Independent. Unsupported diagnostic commands reject in both modes.	Rejections are logged with the `jfr-allowlist` gate.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Policies Requiring Their Own Configuration

TLS selection, client identity, storage authorizers, JAAS/service permissions, health-mutator enablement, and proof-provider setup remain deliberate deployment choices. Hardened policy can change endpoint defaults but does not install every required identity integration. Use Security Configuration Reference and the owning protocol chapter for explicit configuration and precedence.

Copyright © 2008, 2026, Oracle and/or its affiliates.

C

Security Errors and Diagnostic Signals

Identify the rejecting boundary and fix the responsible artifact, identity, or configuration with bounded diagnostic data.
Start with the process, protocol, resource, and operation that failed. Retain a safe request identifier and the bounded reason/status. Avoid logging credentials, full subjects, arbitrary keys/values, or complete remote throwable graphs.

Serialization and Executable Failures

Serialization and executable rejection reasons fall into four families. When debugging a failing deployment, identify the family first, then locate the specific row in the table below.

- Missing allowlist — `security-config-missing`, `ClassIdentity package is not allowed`, `unknown user type`, `Deserialization of class ... was rejected`. The class is not known to the receiver; fix by annotating and regenerating, or by registering with POF.
- Remote execution policy — `dynamic-remote-denied-by-mode`, `method-invocation-denied-by-mode`, `script-eval-denied-by-mode`, `lambda-target-not-allowed`, `class-name-on-denylist`, `method-on-denylist`. A lambda-shaped payload, MIP reflection request, or script request was refused by mode policy, target-interface policy, or by the explicit deny-list.
- Bytecode shape — `bytecode-references-gadget`, `native-method-declared`, `dynamic-class-forname`, `invalid-bytecode`. Wire-supplied bytecode failed structural or deny-list checks.
- Dynamic class definition — `Invalid class definition: missing-magic`, `invalid-version`, `invalid-shape`, `trailing-bytes`. The class-definition payload did not pass file-format validation.

Reason or message shape	Subsystem	Typical cause	Fix
`Lambda bytecode rejected: site=%s, reason=security-config-missing, denied=%s`	Allowlist	An application class arrived as a class-name payload but is not in generated security config or the manual allowlist.	Annotate the producing class with `@Remote.Allowed` or `@PortableType`, run the Maven or Gradle generator, and package the generated config.
`Lambda bytecode rejected: site=LAMBDA, reason=dynamic-remote-denied-by-mode`	Lambda mode gate	A DYNAMIC lambda arrived while `coherence.remote.dynamic.unauthenticated` resolved to an explicit deny.	Switch to STATIC lambda serialization or explicitly set `-Dcoherence.remote.dynamic.unauthenticated=allow`.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Reason or message shape	Subsystem	Typical cause	Fix
`method-invocation-denied-by-mode`	MIP mode gate	A `MethodInvocationProcessor` arrived while `coherence.remote.dynamic.unauthenticated` resolved to an explicit `deny`.	Set `-Dcoherence.remote.dynamic.unauthenticated=allow` only where reflective method invocation is intentionally exposed, or replace the call with a purpose-built `@Remote.Executable` processor.
`script-eval-denied-by-mode`	Script mode gate	A `ScriptProcessor` arrived while `coherence.remote.dynamic.unauthenticated` resolved to an explicit `deny`.	Set `-Dcoherence.remote.dynamic.unauthenticated=allow` only where remote script evaluation is intentionally exposed, or replace the script with a purpose-built `@Remote.Executable` processor.
`Lambda bytecode rejected: site=%s, reason=lambda-target-not-allowed, denied=%s`	Lambda target	A serialized lambda targets a functional interface that is not marked as a lambda target in merged security config. The denied value is the functional interface FQN, not the lambda capture class.	Annotate the functional interface with `@Remote.Executable` and regenerate security config, or add a manual `<class name="..." lambda-target="true"/>` entry for that interface.
`reason=bytecode-references-gadget`	Bytecode scan	Generated or supplied lambda bytecode references a deny-listed gadget class or method.	Remove the reference or use a safe server-side class. Use `coherence.lambda.bytecode.allow` only for emergency class-reference overrides.
`reason=class-name-on-denylist`	Bytecode scan / MIP reflection	A static lambda, class-name payload, MIP supplier, or MIP target value names a deny-listed class.	Remove the deny-listed class from the payload path or reflective MIP target.
`reason=method-on-denylist`	MIP reflection	A `MethodInvocationProcessor` targets a deny-listed `class#method` pair.	Remove the reflective method call and use a purpose-built safe server-side processor.
Graal `PolyglotException` or script error naming a denied Java type	Script HostAccess	A script attempted `Java.type(...)` or guest-side host access for a deny-listed class such as `java.lang.Runtime` or `java.lang.ProcessBuilder`.	Remove the host access from the script. `coherence.remote.dynamic.unauthenticated` does not bypass the HostAccess hard floor.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Reason or message shape	Subsystem	Typical cause	Fix
`reason=native-method-declared`	Bytecode scan	Wire-supplied bytecode declares a native method.	Do not send dynamic bytecode that declares native methods.
`reason=dynamic-class-forname`	Bytecode scan	Wire-supplied bytecode calls `Class.forName` with a non-constant value.	Resolve the class server-side or remove dynamic class loading from the lambda.
`reason=invalid-bytecode`	Bytecode scan / Class definition	The supplied class bytes are empty, malformed, or cannot be parsed.	Regenerate the payload with a supported Coherence version and check for transport corruption.
`Invalid class definition: missing-magic`	Class definition	A class-definition payload does not start with the Java class-file magic number.	Reject the payload source; resend only valid class files.
`Invalid class definition: invalid-version`	Class definition	A class-definition payload uses an unsupported class-file version.	Build the producing artifact for a Java version supported by the receiving member.
`Invalid class definition: invalid-shape`	Class definition	A class-definition payload has an unexpected constant-pool, method, or class shape.	Regenerate the payload and verify client/server version compatibility.
`Invalid class definition: trailing-bytes`	Class definition	A class-definition payload contains bytes after the parsed class file.	Reject the payload source; resend a clean class definition.
`ClassIdentity package is not allowed: %s`	Class identity / Allowlist	A dynamic class identity names a package not present in generated security config or the runtime allowlist.	Annotate the producing type or package and regenerate security config.
`unknown user type: %s. Unregistered POF type ... strict mode`	POF strict / Allowlist	A POF payload names a user type that is not registered in the receiver's POF configuration.	Register the class in `<pof-config>`, annotate it with `@PortableType`, or add `@Remote.Allowed` and regenerate security config.
`Deserialization of class %s was rejected`	JEP-290 filter	The JEP-290 filter rejected a Java-serialization class in a fail-closed path.	Use a registered safe type or add the application type to generated security config or `coherence.serialization.allowed`.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Reason or message shape	Subsystem	Typical cause	Fix
`coh.executable.policy_check{result=would_reject, class=<FQN>, reason=<OperationReason>, role=<SerializationRole>}`	Executable policy (shadow)	Compatibility security mode classified a remote executable object that would be rejected by `DefaultRemoteExecutablePolicy.enforce(...)` in hardened security mode.	Treat each tuple as migration work: annotate the class with `@Remote.Executable`, regenerate `security-config.xml`, or add a manual `executable="true"` entry before setting `-Dcoherence.security.mode=hardened`.

Policy Telemetry

Serialization gate decisions are visible through `SerializationGates` MBeans. ObjectNames include the route, gate, result, reason, and mode tuple, for example:

```
Coherence:type=SerializationGates,route=GRPC,gate=filter,result=rejected,reason=serialization-allowlist-rejected,mode=prod
```

Lambda bytecode checks also maintain the in-process counter key:

```
coh.lambda.bytecode_check{result=rejected,reason=dynamic-remote-denied-by-mode,site=lambda}
```

Use these counters to identify which boundary is rejecting traffic before loosening any runtime property.

Live remote executable enforcement checks in hardened security mode maintain:

```
coh.executable.policy_check{reason=<OperationReason>,
role=<SerializationRole>, result=allowed|rejected, mode=<runtime-mode>}
```

For Extend `InvocationService`, the tuple uses `reason=INVOKE` and `role=EXTEND_PROXY`. `MethodInvocationProcessor` and `ScriptProcessor` use `reason=PROCESS_ENTRY` and `reason=SCRIPT_EVAL`, respectively. Remote `MapTrigger` installation uses `reason=TRIGGER` and `role=EXTEND_PROXY`. Topic subscriber filter/extractor installation and persisted replay use `reason=EVALUATE_FILTER|EXTRACT` and `role=TOPICS`. Cache data-plane processor, aggregator, filter, extractor, and comparator installs through Extend, REST, and gRPC use `reason=PROCESS_ENTRY|AGGREGATE|EVALUATE_FILTER|EXTRACT|COMPARE` and `role=EXTEND_PROXY|REST|GRPC`. Coherence Concurrent executor task and callback installs use `reason=CONCURRENT_TASK` and `role=CONCURRENT`. Live tuples may also include `sub_reason=policy|mode_gate|denylist|replay_drift`; the tag is additive and callers must not assume a closed set. Successful policy checks use `sub_reason=policy`; mode and deny-list sub-reasons are emitted only for refusals. Topic replay under `warn-allow` records `result=allowed,sub_reason=replay_drift` to make migration drift visible.

Remote executable policy shadow checks in compatibility security mode maintain:

```
coh.executable.policy_check{result=would_reject, class=<FQN>,
reason=<OperationReason>, role=<SerializationRole>}
```


Copyright © 2008, 2026, Oracle and/or its affiliates.

This counter is emitted by `DefaultRemoteExecutablePolicy.enforce(...)` in compatibility mode. It identifies classes that fail that policy check; a dynamic install may subsequently be allowed by the independent dynamic gate, so a shadow tuple is not proof that the final hardened request will fail. The `mode` tag is absent from these shadow tuples, while `class` identifies migration candidates. It does not measure serialization allowlist or lambda bytecode readiness. Test named and dynamic paths with the intended profile and an explicit dynamic setting. Persisted topic replay has its own drift policy; install-time dynamic shadows may include both policy and mode-gate tuples.

Protocol and Operations Symptoms

Symptom	Likely boundary	Next step
TLS handshake or server-name failure	Trust, certificate purpose/name, client certificate, or transport policy	Verify configured provider, real endpoint name, trust and identity material on both ends; do not bypass verification.
HTTP 401 / gRPC UNAUTHENTICATED	Missing/unusable identity or cleartext Basic under hardened policy	Check the acceptor auth method, token type, JAAS entry/module, and required transport.
gRPC PERMISSION_DENIED	Executable or caller policy	Inspect the bounded reason and receiver role; classify code or correct role/resource policy as appropriate.
REST 403 for q	Direct query disabled	Prefer a configured named query, or explicitly approve direct querying and bounds.
REST 404 for a pass-through cache	Resource not published under hardened policy	Declare only the intended REST resource; do not expose all caches to fix a single name.
REST 400 for expressions	Unknown alias or unsafe expression	Use the configured alias/ supported query subset and verify URL encoding.
JSON binding/metadata failure	Alias/class policy or malformed input	Configure the actual consuming serializer/marshaller; inspect nested metadata and supported class policy.
XML parser capability failure	Required external-entity protections unavailable	Use a supported provider/runtime and approved resource resolution.
Management/Reporter result=would_reject	Administrative compatibility shadow	Approve exact targets/sources/ output roots and retest under hardened policy; these warnings do not increment executable counters.
JMXMP startup failure	Authenticator or process-wide filter initialization	Configure the supported authenticator and review existing/ fallback filter behavior.
Peer proof unavailable/required-policy rejection	Provider, authority, capability, or topology readiness	Check the supported provider and all-member prerequisites; never substitute a deterministic test provider.
RAG proxy-not-allowed	JVM selected a proxy for a direct-only import	Use an approved direct import path or another supported ingestion mechanism.

Copyright © 2008, 2026, Oracle and/or its affiliates.

Symptom	Likely boundary	Next step
RAG path-not-allowed / unsupported filesystem	Import root or secure traversal requirement	Use protected supported roots/ filesystem or another approved import source.

Copyright © 2008, 2026, Oracle and/or its affiliates.