Oracle® Fusion Middleware

Securing Oracle Coherence

14c (14.1.2.0.0)
F79655-01
December 2024

Oracle Fusion Middleware Securing Oracle Coherence, 14c (14.1.2.0.0)

F79655-01

Primary Author: Oracle Corporation

Contents

Preface

Audience vii

Documentation Accessibility vii

Diversity and Inclusion viii

Related Documents viii

Conventions viii

1 Introduction to Oracle Coherence Security

Conceptual Overview of Oracle Coherence Security 1-1

Coherence Security Quick Start 1-2

Overview of Security Configuration 1-2

2 Enabling General Security Measures

Using the Java Security Manager 2-1

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

Using Host-Based Authorization 2-3

- Overview of Host-Based Authorization 2-4
- Specify Cluster Member Authorized Hosts 2-4
- Specify Extend Client Authorized Hosts 2-5
- Use a Filter Class to Determine Authorization 2-5

Managing Rogue Clients 2-6

3 Using an Access Controller

Overview of Using an Access Controller 3-1

Using the Default Access Controller Implementation 3-4

- Enable the Access Controller 3-4
- Create a Keystore 3-4
- Include the Login Module 3-5
- Create a Permissions File 3-5

Create an Authentication Callback Handler 3-6
Enable Security Audit Logs 3-6
Using a Custom Access Controller Implementation 3-7

4 Authorizing Access to Server-Side Operations

Overview of Access Control Authorization 4-1
Creating Access Control Authorization Implementations 4-2
Declaring Access Control Authorization Implementations 4-4
Enabling Access Control Authorization on a Partitioned Cache 4-5

5 Securing Extend Client Connections

Using Identity Tokens to Restrict Client Connections 5-1
Overview of Using Identity Tokens 5-1
Creating a Custom Identity Transformer 5-3
Enabling a Custom Identity Transformer 5-4
Creating a Custom Identity Asserter 5-4
Enabling a Custom Identity Asserter 5-5
Using Custom Security Types 5-5
Understanding Custom Identity Token Interoperability 5-6
Associating Identities with Extend Services 5-6
Implementing Extend Client Authorization 5-7
Overview of Extend Client Authorization 5-7
Create Authorization Interceptor Classes 5-8
Enable Authorization Interceptor Classes 5-11

6 Using SSL/TLS to Secure Communication

Overview of SSL/TLS 6-2
Coherence Socket Providers 6-4
Configuring the Identity Manager 6-5
Configuring a Trust Manager 6-7
Resolving the Socket Provider URL 6-8
Using a Socket Provider in Configuration 6-8
Configure a Socket Provider at Runtime 6-9
Using SSL to Secure Cluster Communication 6-10
Cluster Communication Using mTLS 6-11
Cluster Communication with One-Way SSL 6-12
Using SSL to Secure Extend and gRPC Client Communication 6-14
Configuring a Cluster-Side Extend Proxy SSL Socket Provider 6-14
Configuring the Cluster-Side gRPC Proxy SSL Socket Provider 6-17

Configuring a Java Extend or gRPC Client SSL Socket Provider 6-17
Configure a Default Socket Provider for a Cache Configuration File 6-22
Configuring a .NET Client-Side Stream Provider 6-24
Securing the C++ Client with SSL/TLS 6-25
Using SSL to Secure Federation Communication 6-26
Federation with mTLS 6-27
Federation with One-Way SSL 6-28
Coherence PeerX509 Algorithm 6-29
Specifying a Global Socket Provider 6-29
Specifying Passwords in Socket Provider Configuration 6-31
Specify Plain Text Passwords 6-32
Passwords From Java System Properties 6-32
Reading Passwords From a URL 6-33
Custom Password Providers 6-33
Controlling Cipher Suite and Protocol Version Usage 6-38
Using Host Name Verification 6-38
Using the Default Coherence Host Name Verifier 6-38
Using a Custom Host Name Verifier 6-40
Configuring Client Authentication 6-40
Using Private Key and Certificate Files 6-41
Configuring an Identity Manager 6-42
Configuring a Trust Manager 6-42

7 Securing Oracle Coherence in Oracle WebLogic Server

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

Enabling Identity Transformers for Use in Oracle WebLogic Server 7-12
Enabling Identity Asserters for Use in Oracle WebLogic Server 7-13

8 Securing Oracle Coherence REST

Overview of Securing Oracle Coherence REST 8-1
Using HTTP Basic Authentication with Oracle Coherence REST 8-1
Specify Basic Authentication for an HTTP Acceptor 8-1
Specify a Login Module 8-2
Using SSL Authentication With Oracle Coherence REST 8-2
Specify Basic Authentication for an HTTP Acceptor 8-3
Configure an HTTP Acceptor SSL Socket Provider 8-3
Access Secured REST Services 8-4
Using SSL and HTTP Basic Authentication with Oracle Coherence REST 8-7
Implementing Authorization For Oracle Coherence REST 8-7

9 Securing Oracle Coherence HTTP Management Over REST Server

About Securing Oracle Coherence HTTP Management Server 9-1
Basic Authentication for Coherence HTTP Management Server HTTP Acceptor 9-1
Specify the Basic Authentication for Coherence HTTP Management Server HTTP Acceptor 9-1
Specify a Coherence HTTP Management Server Login Module 9-2
Using SSL Authentication With Oracle Coherence HTTP Management Server 9-2
Configure a Coherence HTTP Management Acceptor SSL Socket Provider 9-2

10 Securing Oracle Coherence Metrics

About Securing Oracle Coherence Metrics 10-1
Basic Authentication for Coherence Metrics Http Acceptor 10-1
Specify Basic Authentication for Coherence Metrics HTTP Acceptor 10-1
Specify a Coherence Metrics Login Module 10-1
Specify Basic Authentication for a Coherence Metrics HTTP Client 10-2
Using SSL Authentication With Oracle Coherence Metrics 10-2
Configure a Coherence Metrics HTTP Acceptor SSL Socket Provider 10-2

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.

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.

1
Introduction to Oracle Coherence Security

Oracle Coherence includes many security features that provide varying levels of security. Understanding the security features and the uses cases they cover are important first steps when learning how to secure a Coherence solution.

This chapter includes the following sections:

- Conceptual Overview of Oracle Coherence Security
- Coherence Security Quick Start
- Overview of Security Configuration

Conceptual Overview of Oracle Coherence Security

Oracle Coherence provide security features that support standards such as Java policies and Secure Sockets Layer (SSL) and also includes features that are native to Oracle Coherence. Evaluate the security feature descriptions and determine which features to use based on your security requirements, concerns, and tolerances.

The security features are presented from basic security measures to more advanced security measures.

Java Policy Security

A Java security policy file is provided that contains the minimum set of security permissions necessary to run Oracle Coherence. Edit the file to change the permissions based on an application's requirement. The security policy protects against malicious use and alterations of the Oracle Coherence library and configuration files. See Using the Java Security Manager.

Host-Based Authorization

Host-based authorization explicitly specifies which hosts become members of a cluster and which extend clients connect to a cluster. This type of access control is ideal in environments where host names (or IP addresses) are known in advance. Host-based authorization protects against unauthorized hosts joining or accessing a cluster. See Using Host-Based Authorization.

Client Suspect Protocol

The client suspect protocol automatically determines if an extend client is acting malicious and blocks the client from connecting to a cluster. The suspect protocol protects against denial of service attacks. See Managing Rogue Clients.

Client Identity Tokens

Client identity tokens control which extend clients access the cluster. A proxy server allows a connection only if the client presents a valid token. Identity tokens are application-specific and typically reuse existing client authentication implementations. Identity tokens protect against unwanted or malicious clients accessing the cluster. See Using Identity Tokens to Restrict Client Connections.

Client Authorization

Client authorization controls which actions a particular client can perform based on its access control rights. A proxy server performs the authorization check before an extend client accesses a resource (cache, cache service, or invocation service). Client authorization is application-specific and protects against unauthorized use of cluster resources. See Implementing Extend Client Authorization.

Access Controller Security Framework

The access controller manages access to clustered resources, such as clustered services and caches, and controls which operations a user can perform on those resources. Cluster members use login modules to provide proof of identity; while, encrypting and decrypting communication acts as proof of trustworthiness. The framework requires the use of a keystore and defines permissions within a permissions file. The access controller prevents malicious cluster members from accessing and creating clustered resources. See Using an Access Controller .

SSL

SSL secures the Tangosol Cluster Management Protocol (TCMP) communication between cluster nodes. SSL also secures the TCP communication between Oracle Coherence*Extend clients and proxies. SSL uses digital signatures to establish identity and trust, and key-based encryption to ensure that data is secure. SSL is an industry standard that protects against unauthorized access and data tampering by malicious clients and cluster members. See Using SSL/TLS to Secure Communication .

Coherence Security Quick Start

Coherence security features are disabled by default and are enabled as required to address specific security requirements or concerns. Different levels of security can be achieved based on the security features that are enabled. You can quickly get started securing Coherence by configuring a solution to use file permissions, SSL, and role-based authorization.

- Configure file system permissions and Java policy permissions to protect against reads and writes of Coherence files. See Using the Java Security Manager.
- Configure and enable SSL to secure communication between cluster members and protect against unauthorized members joining the cluster. See Using SSL to Secure Cluster Communication.
- When using Coherence*Extend or Coherence REST, configure and enable SSL to secure communication between external clients and Coherence proxy servers. SSL protects against unauthorized clients from using cluster services. See Using SSL to Secure Extend and gRPC Client Communication and Using SSL Authentication With Oracle Coherence REST, respectively.
- Implement authorization policies to restrict client access to specific Coherence operations based on user roles. See Implementing Extend Client Authorization.

Overview of Security Configuration

Coherence security requires the use of multiple configuration files. The configuration files enable, control, and customize security features as required. See Understanding Configuration in *Developing Applications with Oracle Coherence*.
The following files are used to configure security:

- Operational Override File – The `tangosol-coherence-override.xml` file overrides the operational deployment descriptor, which specifies the operational and runtime settings that maintain clustering, communication, and data management services. This file includes security settings for cluster members.
- Cache Configuration File – The `coherence-cache-config.xml` file is the default cache configuration file. It specifies the various types of caches within a cluster. This configuration file includes security settings for cache services, proxy services, and Coherence*Extend clients.

2

Enabling General Security Measures

You can use general security measures to help protect against unauthorized use of Oracle Coherence APIs, system resources, and cluster connections. General security measures are often enabled as a first step when securing Coherence solutions.
This chapter includes the following sections:

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

Using the Java Security Manager

You can control which system resources Coherence accesses and uses by enabling the Java security manager. The security manager uses a policy file that explicitly grants permissions for each resource. The *COHERENCE_HOME*/lib/security/security.policy policy configuration file specifies a minimum set of permissions that are required for Coherence. Use the file as provided, or modify the file to set additional permissions. A set of local (non-clustered) permissions is also provided.
The section includes the following topics:

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

Enable the Java Security Manager

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

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

The `com.tangosol.net.security.LocalPermission` class provides a way to set permissions for local (non-clustered) Coherence API operations. Clients are either allowed or not allowed to perform the declared operations (referred to as targets). For example:

```
LocalPermission lp = new LocalPermission("Cluster.shutdown");
```

To use local permissions, the Java security manager must be enabled. See Enable the Java Security Manager.

Table 2-1 lists and describes the target names that can be declared.

Table 2-1 Local Permission Targets

Target Name	Description
`CacheFactory.setCacheFactoryBuilder`	Protects the programmatic installation of a custom cache factory builder. Special consideration should be given when granting this permission. Granting this permission allows code to set a cache factory builder and intercept any access or mutation requests to any caches and also allows access to any data that flows into and from those caches.
`Cluster.shutdown`	Protects all services from being shutdown. Granting this permission allows code to programmatically shutdown the cluster node.
`BackingMapManagerContext.getBackingMap`	Protects direct access to backing maps. Special consideration should be given when granting this permission. Granting this permission allows code to get a reference to the backing map and access any stored data without any additional security checks.
`BackingMapManagerContext.setClassLoader`	Protect changes to class loaders used for storage. The class loader is used by the cache service to load application classes that might not exist in the system class loader. Granting this permission allows code to change which class loader is used for a particular service.
`Service.getInternalService`	Protects access to an internal service, cluster or cache reference. Granting this permission allows code to obtain direct access to the underlying service, cluster or cache storage implementation.
`Service.registerResource`	Protects service registries. Granting this permission allows code to re-register or unregister various resources associated with the service.
`Service.registerEventInterceptor`	Protects the programmatic installation of interceptors. Special consideration should be given when granting this permission. Granting this permission allows code to change or remove event interceptors associated with the cache service thus either getting access to underlying data or removing live events that are designed to protect the data integrity.

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

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 SSL/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.

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>
         </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.

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.

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

```
<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-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>
```

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>
         <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>
```

3
Using an Access Controller

You can enable an access controller to help protect against unauthorized use of cluster resources. The default access controller implementation is based on the key management infrastructure that is part of the HotSpot JDK and uses Java Authentication and Authorization Service (JAAS) for authentication.
This chapter includes the following sections:

- Overview of Using an Access Controller
- Using the Default Access Controller Implementation
- Using a Custom Access Controller Implementation

Overview of Using an Access Controller

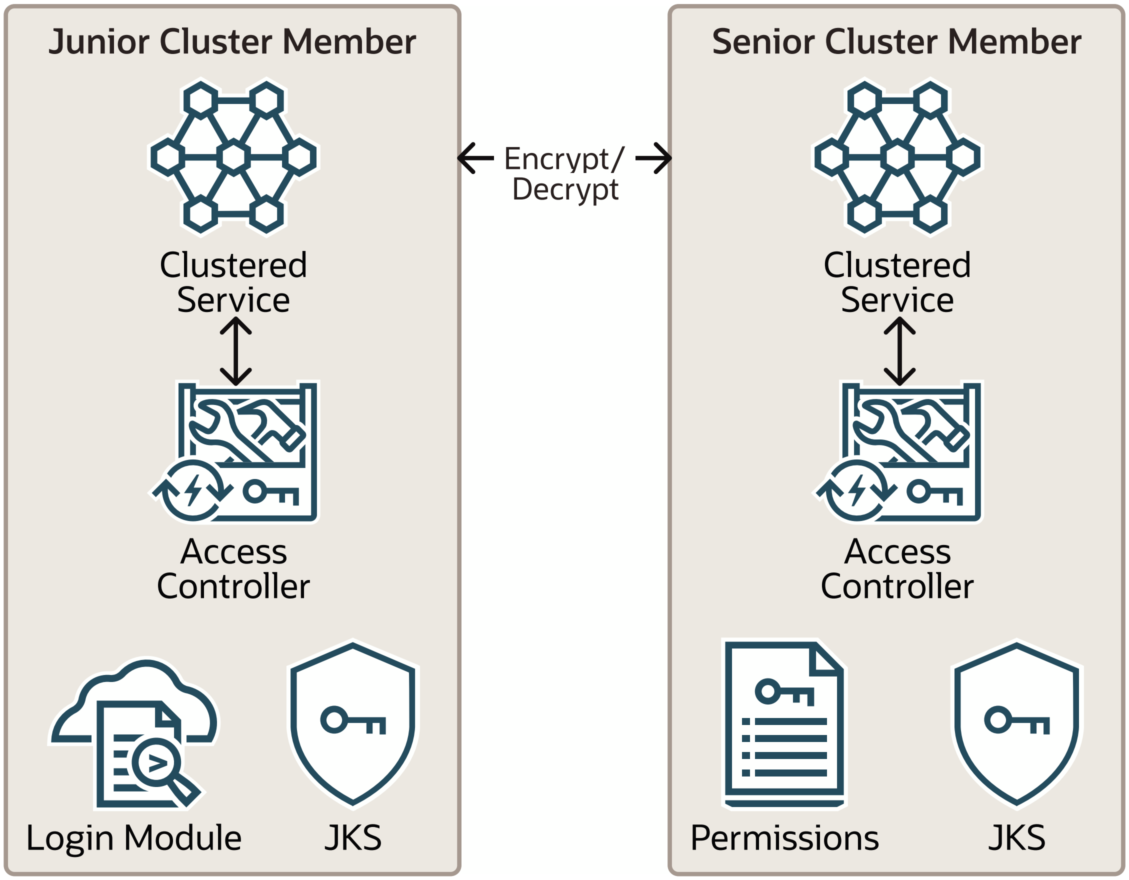
Coherence includes an access controller that is used to secure access to cluster resources and operations. A local login module is used to authenticate a caller, and an access controller on one or more cluster nodes verifies the access rights of the caller. See LoginModule in *Java Authentication and Authorization Service (JAAS) Reference Guide*.
An access controller:

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

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

Figure 3-1 shows a conceptual view of securing two cluster members using access controllers.

Figure 3-1 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, juniors 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.

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.

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 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

The following client code sample authenticates a caller and performs necessary actions:

```
import com.tangosol.net.security.Security;
import java.security.PrivilegedAction;
import javax.security.auth.Subject;

...

Subject subject = Security.login(sName, acPassword);
PrivilegedAction action = new PrivilegedAction()
    {
    public Object run()
        {
        // all processing here is taking place with access
        // rights assigned to the corresponding Subject
        // for example:
         CacheFactory.getCache().put(key, value);
        ...
        }
    };
Security.runAs(subject, action);
```

The caller is authenticated using JAAS on the caller's node during the `login` call. If the authentication is successful, the local access controller:

- Determines whether the local caller has sufficient rights to access the protected clustered resource (local access check)
- Encrypts the outgoing communications regarding the access to the resource with the caller's private credentials retrieved during the authentication phase
- Decrypts the result of the remote check using the requester's public credentials
- Verifies whether the responder has sufficient rights to be granted access

The encryption step provides proof of identity for the responder and blocks a malicious node that pretends to pass the local access check phase.

There are two additional ways to provide the client authentication information. First, pass a reference to a `CallbackHandler` class instead of the user name and password. Second, use a previously authenticated `Subject`. The latter approach is ideal when a Java EE application uses Oracle Coherence and retrieves an authenticated `Subject` from the application container.

If a caller's request does not include any authentication context, a `CallbackHandler` implementation is instantiated and called. The implementation is declared in an operational override file and retrieves the appropriate credentials. However, this lazy approach is much less efficient, because without an externally defined call scope every access to a protected clustered resource forces repetitive authentication calls.

Proof of Trustworthiness

Cluster members use explicit API calls to create clustered resources. The senior service member retains the private credentials that are presented during a call as a proof of trustworthiness. When the senior service member receives an access request to a protected clustered resource, the local access controller:

- Decrypts the incoming communication using the remote caller's public credentials
- Encrypts the access check response using the private credentials of the service.

- Determines whether the remote caller has sufficient rights to access the protected clustered resource (remote access check).

Using the Default Access Controller Implementation

Coherence includes a default access controller implementation that uses a standard Java keystore for authentication. The implementation class is the `com.tangosol.net.security.DefaultController` class. It is configured within the `<security-config>` element in the operational deployment descriptor. See security-config in *Developing Applications with Oracle Coherence*.
This section includes the following topics:

- Enable the Access Controller
- Create a Keystore
- Include the Login Module
- Create a Permissions File
- Create an Authentication Callback Handler
- Enable Security Audit Logs

Enable the Access Controller

To enable the default access controller implementation within the `<security-config>` element, add an `<enabled>` element that is set to `true`. For example:

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

The `coherence.security` system property also enables 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.

Create a Keystore

An access controller requires a keystore that is used by both the controller and login module. Create a keystore with necessary principals using the Java `keytool` utility. Ensure that the keystore is found on the classpath at runtime, or use the `coherence.security.keystore` system property to explicitly enter the name and location of the keystore. For example:

```
-Dcoherence.security.keystore=keystore.jks
```

The following example creates three principals: `admin` (to be used by the Java Security framework), `manager`, and `worker` (to be used by Oracle Coherence).

```
keytool -genkey -v -keystore ./keystore.jks -storepass password -alias admin
-keypass password -dname CN=Administrator,O=MyCompany,L=MyCity,ST=MyState

keytool -genkey -v -keystore ./keystore.jks -storepass password -alias manager
-keypass password -dname CN=Manager,OU=MyUnit

keytool -genkey -v -keystore ./keystore.jks -storepass password -alias worker
-keypass password -dname CN=Worker,OU=MyUnit
```

Include the Login Module

Oracle Coherence includes the `COHERENCE_HOME/lib/security/coherence-login.jar` Java keystore (JKS) login module, which depends only on standard Java run-time classes. Place the library in the JRE `lib/ext` (standard extension) directory. The name in the `<login-module-name>` element, within the `<security-config>` element, serves as the application name in the `COHERENCE_HOME/lib/security/login.config` login module file. The login module declaration contains the path to the keystore. Change the `keyStorePath` variable to the location of the keystore.

```
// LoginModule Configuration for Oracle Coherence
Coherence {
    com.tangosol.security.KeystoreLogin required
      keyStorePath="${user.dir}${/}security${/}keystore.jks";
};
```

Create a Permissions File

An access controller requires a `permissions.xml` file that declares access rights for principals. See the `COHERENCE_HOME/lib/security/permissions.xsd` schema for the syntax of the permissions file. Ensure that the file is found on the classpath at runtime, or use the `coherence.security.permissions` system property to explicitly enter the name and location of the permissions file. For example:

```
-Dcoherence.security.permissions=permissions.xml
```

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>
```

Create an Authentication Callback Handler

An access controller uses an authentication callback handler to authenticate a client when all other authentication methods have been unsuccessful. To create a callback handler, implement the `javax.security.auth.callback.CallbackHandler` interface.

> **Note:**
>
> the handler approach is much less efficient since without an externally defined call scope every access to a protected clustered resource forces repetitive authentication calls.

To configure a custom callback handler within the `<security-config>` element, add a `<callback-handler>` element that includes the fully qualified name of the implementation class. The following example configures a callback handler named `MyCallbackHandler`.

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

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. For example:

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

Security audit logs are not enabled by default. To enable audit logs within the `<security-config>` element, override the security log initialization parameter within the `<access-controller>` element and set the parameter value to `true`. For example,

```
<security-config>
   <access-controller>
      <init-params>
         <init-param id="3">
            <param-type>boolean</param-type>
            <param-value system-property="coherence.security.log">
               true</param-value>
         </init-param>
      </init-params>
   </access-controller>
</security-config>
```

The `coherence.security.log` system property also enables security audit logs. For example:

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

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>
   <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>
```

4

Authorizing Access to Server-Side Operations

Coherence supports server-side authorization to ensure that only specific users can perform certain operations. Authorization is often used together with authentication to provide increased security assurances.
This chapter includes the following sections:

- Overview of Access Control Authorization
- Creating Access Control Authorization Implementations
- Declaring Access Control Authorization Implementations
- Enabling Access Control Authorization on a Partitioned Cache

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 4-1 lists which authorization checks are caused by `NamedCache` API and `BinaryEntry` API methods.

Table 4-1 Authorization Checks for Common Methods

Authorization Check	NamedCache API Methods	BinaryEntry API Methods
None	• `containsKey` • `containsValue` • `isEmpty` • `size` • `lock` • `unlock`	

Table 4-1 (Cont.) Authorization Checks for Common Methods

Authorization Check	NamedCache API Methods	BinaryEntry API Methods
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

Access control authorization requires an authorizer implementation that contains user-defined authorization logic.

To create access control authorization implementations, create a class that implements the `com.tangosol.net.security.StorageAccessAuthorizer` interface. The implementation should define which callers (based on the `Subject`) are authorized to access entries and backing map contexts (`BinaryEntry` and `BackingMapManagerContext`, respectively).

> **Note:**
>
> The `BinaryEntry` and `BackingMapManagerContext` API provide the ability to retrieve the cache name, the service name, and full access to the service and cluster registries.

Example 4-1 Provides a sample `StorageAccessAuthorizer` implementation that emits a log message for each authorization request. It is based on the `AuditingAuthorizer` class that is provided with Coherence and used by the default access controller implementation.

Example 4-1 Sample StorageAccessAuthorizer Implementation

```
package com.examples.security;

import com.tangosol.net.BackingMapContext;
import com.tangosol.net.CacheFactory;
import com.tangosol.net.security.StorageAccessAuthorizer;
import com.tangosol.util.BinaryEntry;

import javax.security.auth.Subject;

public class MyLogAuthorizer implements StorageAccessAuthorizer
{
    public MyLogAuthorizer()
        {
        this(false);
        }

    public MyLogAuthorizer(boolean fStrict)
        {
        f_fStrict = fStrict;
        }

    @Override
    public void checkRead(BinaryEntry entry, Subject subject, int nReason)
        {
        logEntryRequest(entry, subject, false, nReason);

        if (subject == null && f_fStrict)
            {
            throw new SecurityException("subject is not provided");
            }
        }

    @Override
    public void checkWrite(BinaryEntry entry, Subject subject, int nReason)
        {
        logEntryRequest(entry, subject, true, nReason);

        if (subject == null && f_fStrict)
            {
            throw new SecurityException("subject is not provided");
            }
        }

    @Override
    public void checkReadAny(BackingMapContext context, Subject subject,
        int nReason)
        {
        logMapRequest(context, subject, false, nReason);

        if (subject == null && f_fStrict)
            {
            throw new SecurityException("subject is not provided");
            }
        }

    @Override
    public void checkWriteAny(BackingMapContext context, Subject subject,
        int nReason)
        {
        logMapRequest(context, subject, true, nReason);
        if (subject == null && f_fStrict)
            {
            throw new SecurityException("subject is not provided");
            }
        }

    protected void logEntryRequest(BinaryEntry entry, Subject subject,
        boolean fWrite, int nReason)
        {
        CacheFactory.log('"' + (fWrite ? "Write" : "Read")
            + "\" request for key=\""
            + entry.getKey()
            + (subject == null ?
                "\" from unidentified user" :
                "\" on behalf of " + subject.getPrincipals())
            + " caused by \"" + nReason + "\""
            , CacheFactory.LOG_INFO);
        }

    protected void logMapRequest(BackingMapContext context, Subject subject,
        boolean fWrite, int nReason)
        {
        CacheFactory.log('"' + (fWrite ? "Write-any" : "Read-any")
            + "\" request for cache \""
            + context.getCacheName() + '"'
            + (subject == null ?
                " from unidentified user" :
                " on behalf of " + subject.getPrincipals())
            + " caused by \"" + nReason + "\""
            , CacheFactory.LOG_INFO);
        }

    private final boolean f_fStrict;
    }
```

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>
```

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>
```

5
Securing Extend Client Connections

You can use identity tokens and interceptor classes to provide authentication and authorization for Oracle Coherence*Extend clients. Identity tokens protect against unauthorized access to an extend proxy. Interceptor classes control which operations are available to an authenticated client.
This chapter includes the following sections:

- 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
- Enabling a Custom Identity Transformer
- Creating a Custom Identity Asserter
- Enabling a Custom Identity Asserter
- Using Custom Security Types
- Understanding Custom Identity Token Interoperability

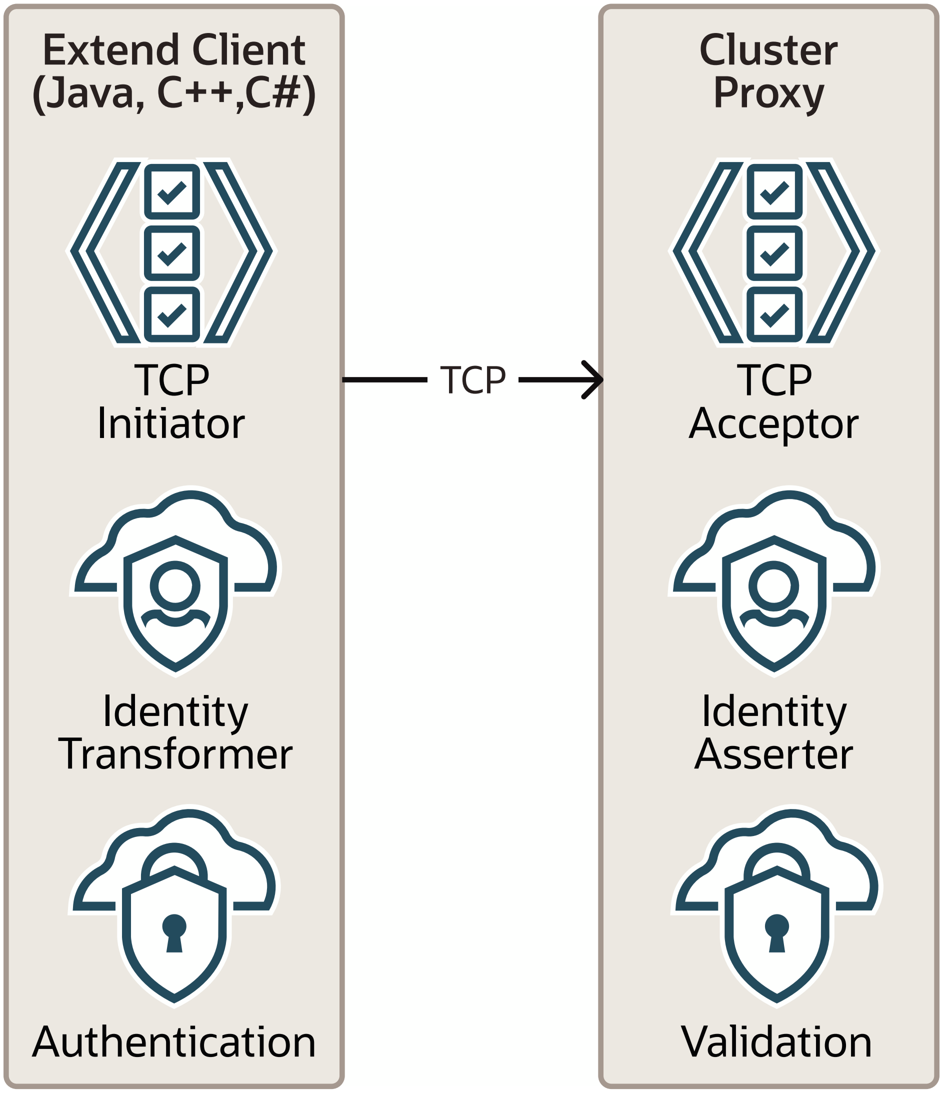
Overview of Using Identity Tokens

Identity token security uses an identity transformer implementation to create identity tokens and an identity asserter implementation to validate identity tokens. These implementations are described as follows:

- Identity transformer – a client-side component that converts a `Subject`, or `Principal`, into an identity token that is passed to an extend proxy. An identity token can be any type of object that is useful for identity validation; it is not required to be a well-known security type. In addition, clients can connect to multiple proxy servers and authenticate to each proxy server differently.
- Identity asserter – A cluster-side component that resides on the cache server that is hosting an extend proxy service. The asserter validates an identity token that is created by an identity transformer on the extend client. The asserter validates identity tokens unique for each proxy service to support multiple means of token validation. The token is passed when an extend client initiates a connection. If the validation fails, the connection is refused and a security exception is thrown. The transformer and asserter are also invoked when a new channel within an existing connection is created.

Figure 5-1 shows a conceptual view of restricting client access using identity tokens.

Figure 5-1 Conceptual View of Identity Tokens



An identity transformer (`DefaultIdentityTransformer`) and identity asserter (`DefaultIdentityAsserter`) are provided and enabled by default. The implementations simply use the `Subject` (Java) or `Principal` (.NET) as the identity token. The default behavior is overridden by providing custom identity transformer and identity asserter implementations and enabling them in the operational override file.

Note:

- At runtime, identity transformer implementation classes must be located on the extend client's classpath and identity asserter implementation classes must be located on the extend proxy server's classpath.
- You can use security object types other than the types that are predefined in Portable Object Format (POF). See Using Custom Security Types.

Creating a Custom Identity Transformer

A default identity transformer implementation (`DefaultIdentityTransformer`) is provided that simply returns a `Subject` or `Principal` that is passed to it. If you do not want to use the default implementation, you can create your own custom transformer implementation.

Note:

At runtime, identity tokens are automatically serialized for known types and sent as part of the extend connection request. For .NET and C++ clients, the type must be a POF type. You can use security object types other than the predefined POF types. See Using Custom Security Types.

For Java and C++, create a custom identity transformer by implementing the `IdentityTransformer` interface. C# clients implement the `IIdentityTransformer` interface.

Example 5-1 demonstrates a Java implementation that restricts client access by requiring a client to supply a password to access the proxy. The implementation gets a password from a system property on the client and returns it as an identity token.

Example 5-1 A Sample Identity Transformer Implementation

```
import com.tangosol.net.security.IdentityTransformer;
import javax.security.auth.Subject;
import com.tangosol.net.Service;

public class PasswordIdentityTransformer
        implements IdentityTransformer
    {
    public Object transformIdentity(Subject subject, Service service)
            throws SecurityException
        {
        return System.getProperty("mySecretPassword");
        }
    }
```

One possible solution for preexisting client authentication implementations is to add a new `Principal` to the `Subject` with the `Principal` name as the password. Add the password `Principal` to the `Subject` during JAAS authentication by modifying an existing JAAS login module or by adding an additional required login module that adds the password `Principal`. The JAAS API allows multiple login modules, each of which modifies the `Subject`. Similarly, in .NET, add a password identity to the `Principal`. The asserter on the cluster side then validates both the Principal and the password Principal. See Creating a Custom Identity 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

A default identity asserter implementation (DefaultIdentityAsserter) is provided that asserts that an identity token is a Subject or Principal. If you do not want to use the default implementation, you can create your own custom asserter implementation.

For Java and C++, create an identity asserter by implementing the IdentityAsserter interface. C# clients implement the IIdentityAsserter interface.

Example 5-2 is a Java implementation that checks a security token to ensure that a valid password is given. In this case, the password is checked against a system property on the cache server. This asserter implementation is specific to the identity transformer sample in Example 5-1.

Example 5-2 A Sample Identity Asserter Implementation

```
import com.tangosol.net.security.IdentityAsserter;
import javax.security.auth.Subject;
import com.tangosol.net.Service;


public class PasswordIdentityAsserter
        implements IdentityAsserter
    {
    public Subject assertIdentity(Object oToken, Service service)
            throws SecurityException
        {
        if (oToken instanceof String)
            {
            if (((String) oToken).equals(System.getProperty("mySecretPassword")))
                {
                return null;
                }
            }
        throw new SecurityException("Access denied");
        }
    }
```

There are many possible variations when you create an identity asserter. For example, you can create an asserter that rejects connections based on a list of principals, that checks role principals, or validates the signed principal name. The asserter blocks any connection attempts that do not prove the correct identity.

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*.

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

Subject scoping allows remote cache and remote invocation service references that are returned to a client to be associated with the identity from the current security context. By default, subject scoping is disabled, which means that remote cache and remote invocation service references are globally shared.
With subject scoping enabled, clients use their platform-specific authentication APIs to establish a security context. A `Subject` or `Principal` is obtained from the current security context whenever a client creates a `NamedCache` and `InvocationService` instance. All requests are then made for the established `Subject` or `Principal`.

> **Note:**
>
> You can use security object types other than the types that are predefined in POF. See Using Custom Security Types.

For example, if a user with a trader identity calls `CacheFactory.getCache("trade-cache")` and a user with the manager identity calls `CacheFactory.getCache("trade-cache")`, each user gets a different remote cache reference object. Because an identity is associated with that remote cache reference, authorization decisions can be made based on the identity of the caller. See Implementing Extend Client Authorization.

For Java and C++ clients, enable subject scope in the client-side `tangosol-coherence-override.xml` file using the `<subject-scope>` element within the `<security-config>` node. 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>
      <subject-scope>true</subject-scope>
   </security-config>
</coherence>
```

For .NET clients, enable subject scope in the client-side `tangosol-coherence-override.xml` file using the `<principal-scope>` element within the `<security-config>` node. For example:

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

<coherence 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/coherence.xsd">
  <security-config>
     <principal-scope>true</principal-scope>
  </security-config>
</coherence>
```

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 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:

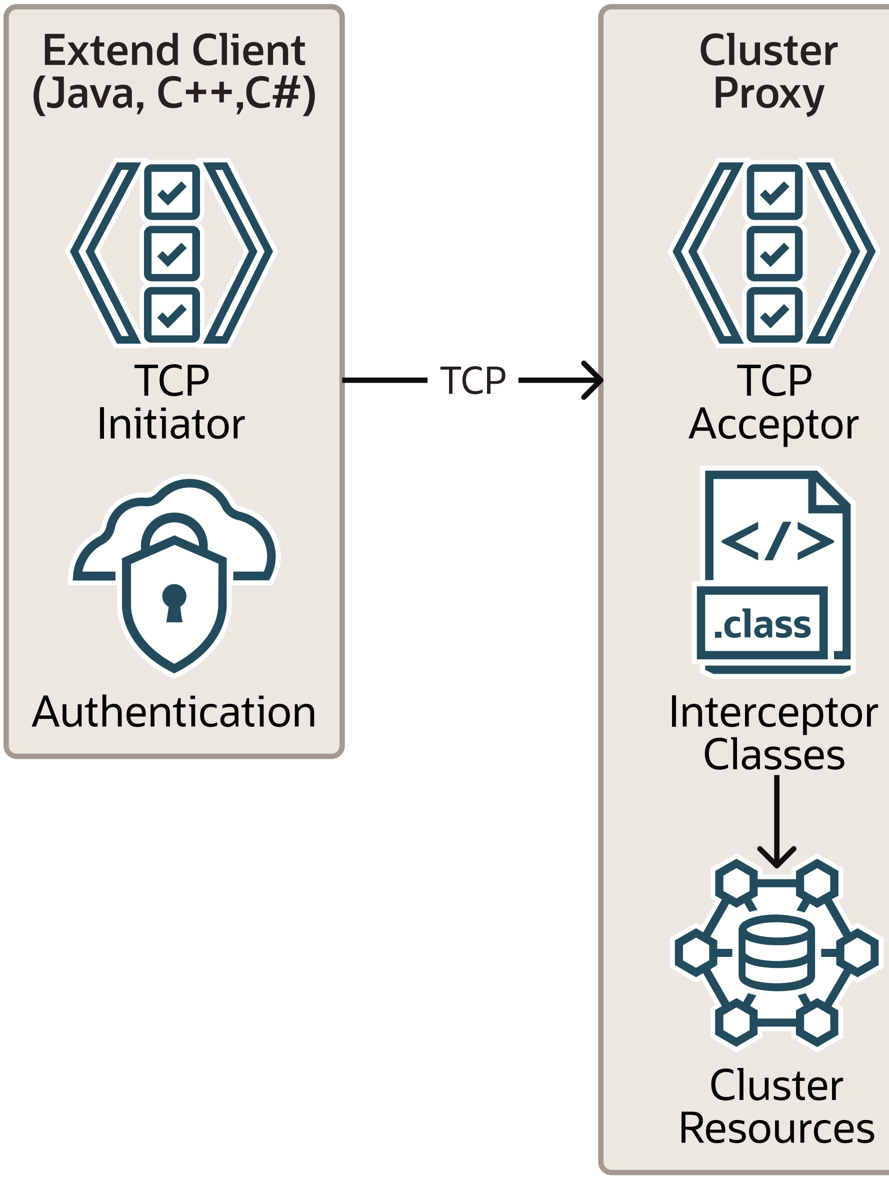
- Overview of Extend Client Authorization
- Create Authorization Interceptor Classes
- Enable Authorization Interceptor Classes

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 5-2 shows a conceptual view of extend client authorization.

Figure 5-2   Conceptual View of Extend Client Authorization



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 5-3 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 5-3 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.

> **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 5-4 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 5-4 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 5-5 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 5-5 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)
      {
      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 5-3 and Example 5-5.

```
<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>
```

6 Using SSL/TLS to Secure Communication

Oracle Coherence supports using the Transport Layer Security (TLS) protocol to secure communication between entities (typically clients and servers) over a network. TLS supersedes the now deprecated Secure Sockets Layer (SSL) protocol.

In Coherence, TLS is configured using Socket Providers, which you can modify to meet your specific security scenarios. Examples for these configuration options are provided throughout this chapter.

> **Note:**
>
> Although the terms TLS, SSL, and SSL/TLS are used interchangeably throughout Coherence documentation, it is expected and encouraged that you use a currently supported version of TLS, *not* SSL, to secure communication in Coherence.

This chapter includes the following sections:

- 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.
- 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.
- Using Private Key and Certificate Files

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.

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 6-1 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 6-1   Basic mTLS socket provider configuration

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

Although Example 6-1 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 6-1 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 6-2, the <identity-manager> loads the key and certificate from a keystore file named server.jks.

Example 6-2 <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 6-3, the configuration uses the <password> element in the <key-store> to set the password as foo.

Example 6-3 <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 6-4, the `<identity-manager>` loads the private key from a file named `server.key` and the certificate from a file named `server.cert`.

Example 6-4 `<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 6-5, 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 6-5 `<identity-manager>` configuration using password protected private keys

```
<identity-manager>
   <key-store>
       <url>file:server.jks</url>
  </key-store>
  <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 6-6, 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 6-6 `<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 6-7, the `<trust-manager>` element loads the CA certificates from a keystore file named `trust.jks`.

Example 6-7 `<trust-manager>` configuration

```
<identity-manager>
   <key-store>
       <url>file:trust.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 6-8, the configuration uses the `<password>` element in the `<key-store>` to set the password as `foo`.

Example 6-8 `<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 6-9, the <trust-manager> loads the certificates from the files named ca-one.cert and ca-two.cert.

Example 6-9 <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 named socket provider in the operational configuration file, or as an in-line socket-provider configuration.

Example 6-10 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 6-10 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 6-11 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 6-11   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>
                <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>
            </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.

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 6-12 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 6-12 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 6-13 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 6-13 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>
```

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 6-14, 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 6-14 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.
- 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 6-15, 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 6-15 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.

> **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 6-16 shows the unicast listener configuration for one-way SSL.

Example 6-16 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>
                 <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.
- 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 6-17 demonstrates a proxy scheme that configures an SSL socket provider directly in the proxy configuration in the cache configuration file.

Example 6-17   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 6-18 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`.

Example 6-18   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 6-19 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 6-19 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 6-20 shows a cluster side proxy socket provider configured for one-way SSL.

Example 6-20 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 6-21 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 6-21 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.

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 6-22 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 6-22 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>
      <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 6-23 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 6-23 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>
         <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 6-24 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 6-24 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>
     </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 6-25 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 6-25   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 6-26 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 6-26   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>
  </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 6-27, 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 6-27 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>
            <scheme-name>remote-grpc</scheme-name>
            <service-name>RemoteGrpcCache</service-name>
        </remote-grpc-cache-scheme>
    </caching-schemes>
</cache-config>
```

In Example 6-28, 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 6-28

```
<?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>
            </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 6-29 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 6-29   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>
```

```
               <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.

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 6-30 shows a `<federated-scheme>` with a `<socket-provider>` configured within the scheme definition.

Example 6-30 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 6-31 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).

Example 6-31 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 6-32, 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 6-32 Configuration for securing federated cluster communication over mTLS

```
<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>
```

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 6-33 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 6-33 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>
```

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 6-34, 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 6-34 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.

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

In Example 6-35, 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 6-35   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 6-36, 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 6-36   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>
```

```
      <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
  You can specify a plain text password directly in the XML configuration using the <password> element.
- 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
  A password provider allows you to get the SSL passwords from any source, including those using encryption. Password providers implement the `com.tangosol.net.PasswordProviderinterface`. The class has a `get` method that returns a password as a Java `char` array.

Specify Plain Text Passwords

You can specify a plain text password directly in the XML configuration using the `<password>` element.

Configuring a plain text password directly in the XML is the least secure way to specify passwords, but it is simple to use and is often used in cases such as integration testing that does not access production data. However, hardcoding a password is also inflexible; whenever the password changes, you need to update the configuration file.

In Example 6-37, the `<password>` element specifies a plain text password for a key store in an `<identity-manager>` element.

Example 6-37 Configuration with a plain text password

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

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 6-38, 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 6-38 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 6-39 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.
- The keystore password used by the trust manager is read from the /coherence/security/trust-pass.txtfile.

Although Example 6-39 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 6-39 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

A password provider allows you to get the SSL passwords from any source, including those using encryption. Password providers implement the com.tangosol.net.PasswordProviderinterface. The class has a get method that returns a password as a Java char array.

Example 6-40 shows a simple password provider implementation that supplies a password char array. A real password provider would obtain the password from somewhere more secure than a hardcoded char array.

Example 6-40 Custom password provider implementation using a password `char` array

```
package com.example.security;

import com.tangosol.net.PasswordProvider;

public class GetPassword implements PasswordProvider {

    public GetPassword() {
    }

    @Override
    public char[] get()
        {
        return new char[]{'s', 'e', 'c', 'r', 'e', 't'};
        }
    }
```

You can specify custom password providers in a socket provider configuration using the `<password-provider>` element. Either provide the full configuration inside the `<password-provider>` element, or set the value of the `<password-provider>` element to the name of a password provider that is configured in the `<password-providers>` section of the operational configuration file.

Example 6-41 uses the password provider in Example 6-40 to obtain the password for a private key in an identity manager configuration.

Example 6-41 Configuration for using a custom password provider to retrieve a password

```
<socket-provider>
   <ssl>
      <identity-manager>
         <key>server.key</key>
         <cert>server.cert</cert>
         <password-provider>
             <class-name>com.example.security.GetPassword</class-name>
         </password-provider>
      </identity-manager>
      <trust-manager>
         <key-store>
            <url>file:trust.jks</url>
         </key-store>
      </trust-manager>
   </ssl>
</socket-provider>
```

Named Password Provider References

If a common password provider configuration will be used multiple times, it is simpler to provide the configuration once in the `<password-providers>` section of the operational configuration file and then reference the named provider from the socket provider configuration.

Example 6-42 shows a password provider specified in the `<password-providers>` section of the operational configuration file. The password provider has a name of `MyPasswordProvider`.

The named password provider can now be referenced from a socket provider.

Example 6-42 Configuration for naming a password provider for reference from a 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>
      <password-providers>
          <password-provider id="MyPasswordProvider">
              <class-name>com.example.security.GetPassword</class-name>
          </password-provider>
      <password-providers>
   </cluster-config>
</coherence>
```

In Example 6-43, the socket provider uses `MyPasswordProvider` to provide the credentials for an encrypted private key file.

This can provide a flexible method of providing passwords. The socket provider configuration refers to a named password provider, rather than a hardcoded value. At runtime, different operational configuration files can be used to provide different configurations or implementations of that named provider.

Example 6-43 Configuration for a socket provider that uses a custom password provider to provide credentials

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

Parameterized Password Providers

A `PasswordProvider` implementation can be parameterized using constructor arguments or using a static factory method with arguments.

In Example 6-44, the simple `PasswordProvider` has a constructor with a single `int` parameter. The value of the parameter determines the password returned. A real password provider would obtain the password from somewhere more secure than a hardcoded `char` array.

The password provider can be defined in the `<password-providers>` section of the operational configuration file.

Example 6-44   Parameterized password provider implementation for retrieving a password

```
package com.oracle.coherence.examples;

import com.tangosol.net.PasswordProvider;

public class GetPassword implements PasswordProvider {

    private final int param;

    public GetPassword(int param) {
        this.param = param;
    }

    @Override
    public char[] get()
        {
        if (param == 0) {
            return new char[]{'s', 'e', 'c', 'r', 'e', 't'};
        }
        return new char[]{'p', 'a', 's', 's', 'w', 'o', 'r', 'd'};
    }
}
```

In Example 6-45, the password provider in Example 6-44 is configured with a name of `MyPasswordProvider`. The `<init-params>` element is used to specify the constructor parameters. In this case, a single `int` parameter named `password-id` with a value of 0 (zero).

The named password provider can now be referenced from a socket provider.

Example 6-45   Configuration for specifying constructor parameters

```
<?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>
      <password-providers>
          <password-provider id="MyPasswordProvider">
              <class-name>com.example.security.GetPassword</class-name>
              <init-params>
                  <init-param>
                      <param-name>password-id</param-type>
                      <param-value>0</param-value>
                  </init-param>
              </init-params>
          </password-provider>
      <password-providers>
   </cluster-config>
</coherence>
```

In Example 6-46, the socket provider uses `MyPasswordProvider` to provide the credentials for an encrypted private key file. In this case, the password provider is configured with a parameter of 0 (zero), so an `int` value of 0 will be passed to the constructor.

Example 6-46 Configuration for using a parameterized password provider to retrieve a password

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

In Example 6-47, `MyPasswordProvider` is used again but this time the `password-id` parameter is overridden to be a 1, so an `int` value of 1 will be passed to the password provider constructor.

Example 6-47 Configuration for using a parameterized password provider to retrieve a password with an inline parameter override

```
<socket-provider>
    <ssl>
        <identity-manager>
            <key>server.key</key>
            <cert>server.cert</cert>
            <password-provider>
                <name>MyPasswordProvider</name>
                <init-params>
                    <init-param>
                        <param-name>password-id</param-name>
                        <param-value>1</param-value>
                    </init-param>
                </init-params>
            </password-provider>
        </identity-manager>
        <trust-manager>
            <key-store>
                <url>file:trust.jks</url>
            </key-store>
        </trust-manager>
    </ssl>
</socket-provider>
```

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.
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.
- 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.

By default, the host name verifier is not enabled for backward compatibility. However, it is enabled in the secured production mode by default. To enable or disable the default host name verifier, see the description for the `<hostname-verifier>` element in ssl.

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 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 6-48 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.

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 6-49 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>
         </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 6-50 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 6-50   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.

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 6-51 shows a `<trust-manager>` configuration that uses a certificate loaded from the `/coherence/security/server-ca.cert` file.

Example 6-51   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.

7 Securing Oracle Coherence in Oracle WebLogic Server

Authentication and authorization can be used to secure Coherence in an Oracle WebLogic Server domain.

This chapter includes the following sections:

- 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
- 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 SSL/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.

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 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.

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**.
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.

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**.

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.

> **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:

```
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.

> **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.

   > **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

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`.
  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.

8
Securing Oracle Coherence REST

Authentication and authorization can be used to secure Oracle Coherence REST. If you are new to Coherence REST, See Using Coherence REST in *Developing Remote Clients for Oracle Coherence*.
This chapter includes the following sections:

- 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

Overview of Securing Oracle Coherence REST

Oracle Coherence REST security uses both authentication and authorization to restrict access to cluster resources. Authentication and authorization are disabled by default and are enabled as required. Authentication support includes: HTTP basic, client-side SSL certificate, and client-side SSL certificate together with HTTP basic. Authorization is implemented using Oracle Coherence*Extend-styled authorization, which relies on interceptor classes that provide fine-grained access for cache service and invocation service operations. Oracle Coherence REST authentication and authorization reuses much of the existing security capabilities of Oracle Coherence: references are provided to existing content where applicable.

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

HTTP basic authentication requires a JAAS `javax.security.auth.spi.LoginModule` implementation that authenticates client credentials which are passed from the HTTP basic authentication header. The resulting `Subject` can then be used for both Oracle Coherence*Extend-style and Oracle Coherence Security Framework authorization as required. See LoginModule in *Java Authentication and Authorization Service (JAAS) Reference Guide*.

To specify a login module, modify the *`COHERENCE_HOME`*`/lib/security/login.config` login configuration file and include a `CoherenceREST` entry that includes the login module implementation to use. For example:

```
CoherenceREST {
    package.MyLoginModule required debug=true;
};
```

At runtime, specify the `login.config` file to use either from the command line (using the `java.security.auth.login.config` system property) or in the Java security properties file.

As a convenience, a Java keystore (JKS) `LoginModule` implementation which depends only on standard Java run-time classes is provided. The class is located in the *`COHERENCE_HOME`*`/lib/security/coherence-login.jar` file. To use the implementation, either place this library in the proxy server classpath or in the JRE's `lib/ext` (standard extension) directory.

Specify the JKS login module implementation in the `login.config` configuration file as follows:

```
CoherenceREST {
    com.tangosol.security.KeystoreLogin required
      keyStorePath="${user.dir}${/}security${/}keystore.jks";
};
```

The entry contains a path to a keystore. Change the `keyStorePath` variable to the location of a keystore.

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>
```

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 8-1 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 8-1 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 8-1 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>
```

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>
```

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

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.

9

Securing Oracle Coherence HTTP Management Over REST Server

Oracle Coherence HTTP Management Server security is used to restrict HTTP access to Coherence MBeans exposed as REST resources.
This chapter includes the following sections:

- 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

About Securing Oracle Coherence HTTP Management Server

Coherence HTTP Management Server authentication and authorization are disabled by default and are enabled as required.
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

HTTP basic authentication requires a JAAS javax.security.auth.spi.LoginModule implementation that authenticates client credentials which are passed from the HTTP basic authentication header. The resulting Subject can then be used for Oracle Coherence Security Framework authorization as required. See LoginModule in *Java Authentication and Authorization Service (JAAS) Reference Guide*.

To specify a login module, modify the *COHERENCE_HOME*/lib/security/login.config login configuration file and include a Coherence entry that includes the login module implementation to use. For example:

```
CoherenceManagement {
    package.MyLoginModule required;
};
```

At runtime, specify the login.config file to use either from the command line (using the java.security.auth.login.config system property) or in the Java security properties file.

As a convenience, a Java keystore (JKS) LoginModule implementation which depends only on standard Java run-time classes is provided. The class is located in the *COHERENCE_HOME*/lib/security/coherence-login.jar file. To use the implementation, place this library either in the proxy server classpath or in the JRE's lib/ext (standard extension) directory.

Specify the JKS login module implementation in the login.config configuration file as follows:

```
CoherenceManagement {
    com.tangosol.security.KeystoreLogin required
      keyStorePath="${user.dir}${/}security${/}keystore.jks";
};
```

The entry contains a path to a keystore. Change the keyStorePath variable to the location of a keystore.

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 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 9-1 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>
                <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>
```

For configuring HTTP client access, see Access Secured REST Services.

10
Securing Oracle Coherence Metrics

Oracle Coherence Metrics security is used to restrict access to metrics data through authentication and authorization configuration.
This chapter includes the following sections:

- About Securing Oracle Coherence Metrics
- Basic Authentication for Coherence Metrics Http Acceptor
- Using SSL Authentication With Oracle Coherence Metrics

About Securing Oracle Coherence Metrics

Coherence Metrics authentication and authorization are disabled by default and are enabled as required.
Coherence Metrics authentication support includes: HTTP basic, client-side SSL certificate, and client-side SSL certificate together with HTTP basic.

See Using Coherence Metrics in *Managing Oracle Coherence*.

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 `cohererence.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

HTTP basic authentication requires a JAAS `javax.security.auth.spi.LoginModule` implementation that authenticates client credentials which are passed from the HTTP basic authentication header. The resulting `Subject` can then be used for Oracle Coherence Security Framework authorization as required. See LoginModule in *Java Authentication and Authorization Service (JAAS) Reference Guide*.

To specify a login module, modify the *COHERENCE_HOME*/lib/security/login.config login configuration file and include a Coherence entry that includes the login module implementation to use. For example:

```
CoherenceMetrics {
    package.MyLoginModule required;
};
```

At runtime, specify the login.config file to use either from the command line (using the java.security.auth.login.config system property) or in the Java security properties file.

As a convenience, a Java keystore (JKS) LoginModule implementation which depends only on standard Java run-time classes is provided. The class is located in the *COHERENCE_HOME*/lib/security/coherence-login.jar file. To use the implementation, either place this library in the proxy server classpath or in the JRE's lib/ext (standard extension) directory.

Specify the JKS login module implementation in the login.config configuration file as follows:

```
CoherenceMetrics {
    com.tangosol.security.KeystoreLogin required
      keyStorePath="${user.dir}${/}security${/}keystore.jks";
};
```

The entry contains a path to a keystore. Change the keyStorePath variable to the location of a keystore.

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 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>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"/>
              <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>
   <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.