Oracle® Fusion Middleware

Securing Oracle Coherence

12*c* (12.2.1)

E55621-02

February 2016

Documentation for developers and system administrators that describes how to secure Oracle Coherence clusters, Oracle Coherence*Extend clients, and Oracle Coherence REST, using technologies that offer varying levels of security

ORACLE

Oracle Fusion Middleware Securing Oracle Coherence, 12c (12.2.1)

E55621-02

Primary Author: Joseph Ruzzi

Contents

7 Securing Oracle Coherence in Oracle WebLogic Server

8 Securing Oracle Coherence REST

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.

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 customers that have purchased support have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

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*
-

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.

What's New in This Guide

The following topics introduce the new and changed features of Oracle Coherence and other significant changes that are described in this guide, and provides pointers to additional information.

New and Changed Features for 12*c* (12.2.1)

Oracle Coherence 12*c* (12.2.1) includes the following new and changed features for this document.

- Programmatic local permissions, which provides a way to set permissions for local (non-clustered) Coherence API operations. See "Programmatically Specifying Local Permissions."
- Security audit logs, which report cluster operations being performed by specific users. See "Enable Security Audit Logs."
- Access control authorization, which allows applications to define their own authorization logic to limit access to cluster operations. See Authorizing Access to Server-Side Operations .
- SSL protocol and cipher suites configuration, which controls which SSL protocols and cipher suites can be used. See "Controlling Cipher Suite and Protocol Version Usage."

Other Significant Changes in This Document for 12*c* (12.2.1)

For 12*c* (12.2.1), no other significant changes have been made to this guide.

1

Introduction to Oracle Coherence Security

This chapter provides an introduction to Oracle Coherence security features. Oracle Coherence security features provide varying levels of security and are generally implemented as required. The security features include industry standards, such as Secure Sockets Layer (SSL), and features specific to Oracle Coherence.

Note:

This guide does not provide detailed instructions for setting up a cluster or creating Oracle Coherence*Extend clients. See *Developing Applications with Oracle Coherence* and *Developing Remote Clients for Oracle Coherence*, respectively, for details on setting up a cluster and creating Oracle Coherence*Extend clients.

This chapter includes the following sections:

Conceptual Overview of Oracle Coherence Security

This section lists and describes the security features available for Oracle Coherence and Oracle Coherence*Extend. Evaluate the security features and determine which features to use based on your security requirements, concerns, and tolerances. The organization in this section (and throughout the book) presents basic security measures before 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" for details.

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" for details.

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" for details.

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" for details.

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" for details.

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 , for details.

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 to Secure Communication , for details.

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. The following list provides a quick start to security and results in a Coherence environment that includes file permissions, SSL, and authorization.

- Configure file system permissions and Java policy permissions to protect against reads and writes of Coherence files. See "Using the Java Security Manager" for details.
- Configure and enable SSL to secure communication between cluster members and protect against unauthorized members joining the cluster. See "Using SSL to Secure TCMP 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 Client Communication" and "Using SSL Authentication With Oracle Coherence REST", respectively, for details.

- Implement authorization policies to restrict client access to specific Coherence operations based on user roles. See "Implementing Extend Client Authorization".

Overview of Security Configuration

Security configuration occurs in both an operational override file and the cache configuration file. See *Developing Applications with Oracle Coherence* for detailed information about configuration.

- 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 Oracle Coherence*Extend. Both the extend client side and the cluster side require a cache configuration file. See *Developing Remote Clients for Oracle Coherence* for details on setting up Oracle Coherence*Extend.

2

Enabling General Security Measures

This chapter provides instructions for enabling general security measures. The measures help to protect against unauthorized use of the Oracle Coherence API and system resources. They also protect against unauthorized connections to a cluster.

This chapter includes the following sections:

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

Using the Java Security Manager

Java provides a security manager that controls access to system resources using explicit permissions. The *COHERENCE_HOME*/lib/security/security.policy policy configuration file specifies a minimum set of permissions for Oracle Coherence. Use the file as provided, or modify the file to set additional permissions. Coherence also includes a set of local (non-clustered) permissions.

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 the *Java SE Security Guide* for details about the file format and syntax:

`http://download.oracle.com/javase/7/docs/technotes/guides/security/permissions.html`

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. For details, 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

Table 2-1 (Cont.) 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 only specified hosts (based on host name or IP address) to 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

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 (see "Using Identity Tokens to Restrict Client Connections") or SSL (Using SSL to Secure Communication).

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. See the *Java API Reference for Oracle Coherence* for details on the `Filter` interface.

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

Rogue clients are 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 protocol safeguards against such abuses.

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 *Developing Applications with Oracle Coherence* for details on using the `<tcp-acceptor>` element. 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

This chapter provides instructions for enabling 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. This implementation uses Java Authentication and Authorization Service (JAAS) for authentication.

Note:

This chapter does not discuss SSL. See Using SSL to Secure Communication , for detailed SSL instructions.

This chapter includes the following sections:

Overview of Using an Access Controller

An access controller secures 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 a caller's access rights. See the *JAAS Reference Guide* for details on login modules.

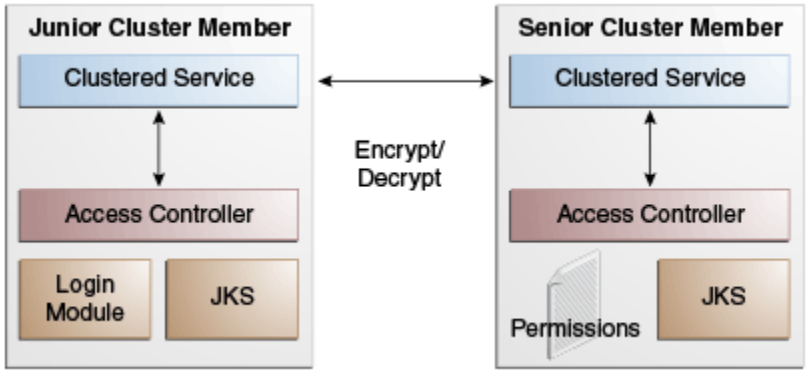
An access controller:

- 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

A default access controller implementation is provided that uses a standard Java keystore. 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 *Developing Applications with Oracle Coherence* for details on the `<security-config>` element and its subelements. To use the default access controller, complete the topics in this section.

This section includes the following topics:

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

Custom access controllers must implement the `com.tangosol.net.security.AccessController` interface. See the *Java API Reference for Oracle Coherence* for details on using this API. 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

This chapter provides instructions for securing access to Coherence operations using authorization.

This chapter includes the following sections:

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

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

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

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

Creating Access Control Authorization Implementations

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

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. For details on the `<storage-auhorizer>` element, see *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

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

This chapter provides instructions for using 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

Identity tokens restrict extend clients from accessing 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

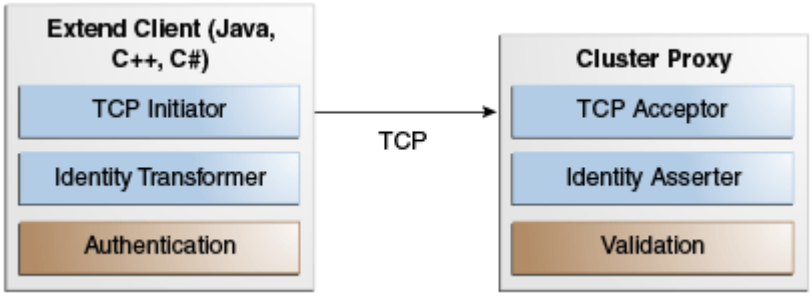
Identity token security uses an identity transformer implementation to create identity tokens and an identity asserter implementation to validate identity tokens. These implementations are described as follows:

- 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.
- See "Using Custom Security Types" for more information about using security object types other than the types that are predefined in Portable Object Format (POF).

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. See "Using Custom Security Types" for more information about using security object types other than predefined POF 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 *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:

See "Using Custom Security Types" for more information about using security object types other than the types that are predefined in POF.

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" below for details on implementing 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.

This section includes the following topics:

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

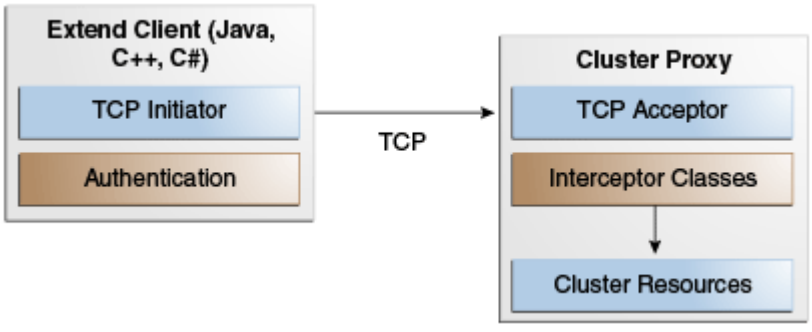
The code samples in this section are based on the Java authorization example, which is included in the examples that are delivered as part of the distribution. The example demonstrates a basic authorization implementation that uses the Principal obtained from a client request and a role-based policy to determine whether to allow operations on the requested service. Download the examples for the complete implementation.

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 5-5 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. For details, 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 5-3 is taken from the Oracle Coherence examples. The example demonstrates creating an authorization interceptor class for a proxied cache service by extending the `WrapperCacheService` class. It wraps all `CacheService` methods on the proxy and applies access controls based on the `Subject` passed from an extend client. The implementation allows only a `Principal` with the specified role to access the `CacheService` methods

Example 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 to Secure Communication

This chapter provides instructions for using Secure Sockets Layer (SSL) to secure TCMP communication between cluster nodes and to secure the TCP communication between Oracle Coherence*Extend clients and proxies. Oracle Coherence supports the Transport Layer Security (TLS) protocol, which superseded the SSL protocol; however, the term SSL is used in this documentation because it is the more widely recognized term.

This chapter includes the following sections:

Overview of SSL

This section provides a brief overview of common SSL concepts that are used in this documentation. It is not intended to be a complete guide to SSL. See the following resources for complete documentation. Users who are familiar with SSL can skip this section.

- Formal SSL and TLS specifications – `http://www.ietf.org`
- Java SE Security –

  `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html`

SSL is a security protocol that secures communication between entities (typically, clients and servers) over a network. SSL 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.

Establishing Identity and Trust

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 also contains the public encryption key embedded within it. 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.

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.

Using One-Way Authentication Versus Two-Way Authentication

SSL communication between clients and servers is set up using either one-way or two-way authentication. With 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. Two-way authentication requires both the client and the server to send their respective digital certificates to each other for mutual authentication. Two-way authentication provides stronger security by assuring that the identity on each sides of the communication is known.

Generating Java SSL Artifacts

The Java `keytool` utility that is located in the *`JDK_HOME`*`/bin` directory generates and manages SSL artifacts. This activity includes: creating a keystore; generating a unique public/private key pair; creating a self-signed digital certificate that includes the public key; associating the certificate with the private key; and storing these artifacts in the keystore.

The following example creates a keystore named `server.jks` that is located in the `/test` directory. A public/private key pair is generated for the entity identified by the `-dname` value (`"cn=administrator, ou=Coherence, o=Oracle, c=US"`). A self-signed certificate is created that includes the public key and identity information. The certificate is valid for `180` days and is associated with the private key in a keystore entry referred to by the alias (`admin`). Both the keystore and private key must have a password.

```
keytool -genkeypair -dname "cn=administrator, ou=Coherence, o=Oracle, c=US"
-alias admin -keypass password -keystore /test/server -storepass password
-validity 180
```

The certificate that is generated by the preceding command is adequate for development purposes. However, certificates are typically verified by a trusted CA (such as VeriSign). To have the certificate verified, use the `keytool` utility to generate a Certificate Signing Request (CSR) file:

```
keytool -certreq -file admin.csr
```

Send the CSR file to a CA, which returns a signed certificate. Use the `keytool` utility to import the returned certificate, which replaces the self-signed certificate in the keystore:

```
keytool -importcert -trustcacerts -file signed_admin.cer
```

Lastly, use the `keytool` utility to create a second keystore that acts as a trust keystore. The trust keystore contains digital certificates of trusted CAs. Certificates that have been verified by a CA are considered trusted only if the CA's certificate is also found in the trust keystore. For example, in a typical one-way authentication scenario, a client must have a trust keystore that contains a digital certificate of the CA that signed the server's certificate. For development purposes, a self-signed certificate can be used for both identity and trust; moreover, a single keystore can be used as both the identity store and the trust keystore.

Generating Windows SSL Artifacts

The following steps describe how to set up two-way authentication on Windows to secure Oracle Coherence*Extend .NET clients. See "Configuring a .NET Client-Side Stream Provider" for details on configuring .NET clients. See the Windows documentation for complete instructions on setting up SSL on Windows:

http://technet.microsoft.com/en-us/library/cc782338%28WS.10%29.aspx

To set up two-way authentication on Windows:

1. Run the following commands from the Visual Studio command prompt:

```
c:\>makecert -pe -n "CN=Test And Dev Root Authority" -ss my -sr LocalMachine -a
sha1 -sky signature -r "Test And Dev Root Authority.cer"

c:\>makecert -pe -n "CN=MyServerName" -ss my -sr LocalMachine -a sha1 -sky
exchange -eku 1.3.6.1.5.5.7.3.1 -in "Test And Dev Root Authority" -is MY -ir
LocalMachine -sp "Microsoft RSA SChannel Cryptographic Provider" -sy 12

c:\>makecert -pe -n "CN=MyClient" -ss my -sr LocalMachine -a sha1 -sky exchange -
eku 1.3.6.1.5.5.7.3.1 -in "Test And Dev Root Authority" -is MY -ir LocalMachine -
sp "Microsoft RSA SChannel Cryptographic Provider" -sy 12
```

2. Create the certificate trusted root certification authority (for tests only).

```
makecert -pe -n "CN=Test And Dev Root Authority" -ss my -sr LocalMachine -a sha1
-sky signature -r "Test And Dev Root Authority.cer"
```

3. Copy the created certificate from the personal store to the trusted root certification authority store.

4. Create the server certificate based on the trusted root certification.

```
makecert -pe -n "CN=MyServerName" -ss my -sr LocalMachine -a sha1 -sky exchange -
eku 1.3.6.1.5.5.7.3.1 -in "Test And Dev Root Authority" -is MY -ir LocalMachine -
sp "Microsoft RSA SChannel Cryptographic Provider" -sy 12
```

5. From the certificate store of the trusted root certification authority (Test And Dev Root Authority), export a certificate file without a public key (`.cer`).

6. From the certificate store of the trusted root certification authority (Test And Dev Root Authority), export a certificate file with a private key (`.pfx`).

7. Copy the `.cer` file to each client computer. The location must be accessible to the `sslstream` client program.

8. Copy the `.pfx` file to each client computer.

9. Import the `.pfx` file to the trusted root certification authority certificate store of each client computer.

10. On each client computer, delete the `.pfx` file. (This step ensures that the client does not communicate or export the private key.)

Using SSL to Secure TCMP Communication

This section provides instructions for configuring SSL to secure communication between cluster members. The configuration examples in this section assume that valid digital certificates for all clients and servers have been created as required and that the certificates have been signed by a Certificate Authority (CA). The digital

certificates must be found in an identity store, and the trust keystore must include the signing CA's digital certificate. Use self-signed certificates during development as needed. See "Using SSL to Secure Extend Client Communication" for instructions on using SSL with Oracle Coherence*Extend.

This section includes the following topics:

- Overview of Using SSL to Secure TCMP Communication
- Define an SSL Socket Provider
- Using the Predefined SSL Socket Provider

Overview of Using SSL to Secure TCMP Communication

Both one-way and two-way SSL authentication are supported with TCMP. Two-way authentication is typically used more often than one-way authentication, which has fewer use cases in a cluster environment. 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. Carefully consider the implications of SSL on administration and performance.

Figure 6-1 shows a conceptual view of cluster members using two-way SSL. Each cluster member includes a trust keystore and a Java keystore (JKS) that contains digital certificates that are used for mutual authentication.

Figure 6-1 Conceptual Architecture of SSL with TCMP

Define an SSL Socket Provider

Configure SSL for TCMP in an operational override file by overriding the `<socket-provider>` element within the `<unicast-listener>` element. The preferred approach is to use the `<socket-provider>` element to reference an SSL socket provider configuration that is defined within a `<socket-providers>` node. However, the `<socket-provider>` element also supports including an in-line SSL configuration. Both approaches are demonstrated in this section. See *Developing Applications with Oracle Coherence* for a detailed reference of the `<socket-provider>` element.

Note:

The use of Well Known Addresses (WKA) is required to use SSL with TCMP. See *Developing Applications with Oracle Coherence* for details on setting up WKA.

Example 6-1 demonstrates an SSL two-way authentication setup. The setup requires both an identity store and trust keystore to be located on each node in the cluster. The example uses the default values for the `<protocol>` and `<algorithm>` element

(`TLS` and `SunX509`, respectively). These are shown only for completeness; you can omit them when you use the default values. The example uses the preferred approach, in which the SSL socket provider is defined within the `<socket-providers>` node and referred to from within the `<unicast-listener>` element.

Example 6-1 Sample SSL Configuration for TCMP Communication

```
<?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 system-property="coherence.socketprovider>
            mySSLConfig</socket-provider>
         <well-known-addresses>
            <socket-address id="1">
               <address system-property="coherence.wka">198.168.1.5
               </address>
               <port system-property="coherence.wka.port">8088</port>
            </socket-address>
         </well-known-addresses>
      </unicast-listener>

      <socket-providers>
         <socket-provider id="mySSLConfig">
            <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>
               <socket-provider>tcp</socket-provider>
            </ssl>
         </socket-provider>
      </socket-providers>
   </cluster-config>
</coherence>
```

As an alternative, the SSL socket provider supports in-line configuration directly in the `<unicast-listener>` element, as shown in Example 6-2:

Example 6-2 Sample In-line SSL Configuration for TCMP Communication

```
<?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 system-property="coherence.socketprovider>
            <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>
               <socket-provider>tcp</socket-provider>
            </ssl>
         </socket-provider>
         <well-known-addresses>
            <socket-address id="1">
               <address system-property="coherence.wka">198.168.1.5
               </address>
               <port system-property="coherence.wka.port">8088</port>
            </socket-address>
         </well-known-addresses>
      </unicast-listener>
   </cluster-config>
</coherence>
```

Using the Predefined SSL Socket Provider

Oracle Coherence includes a predefined SSL socket provider that allows for configuration of two-way SSL connections. The predefined socket provider is based on peer trust: every trusted peer resides within a single JKS keystore. The proprietary peer trust algorithm (PeerX509) works by assuming trust (and only trust) of the certificates that are in the keystore and leverages the fact that TCMP is a peer-to-peer protocol.

The predefined SSL socket provider is defined within the `<socket-providers>` element in the operational deployment descriptor:

```
...
<cluster-config>
   <socket-providers>
      <socket-provider id="ssl">
         <ssl>
            <identity-manager>
               <key-store>
                  <url system-property="coherence.security.keystore">
                     file:keystore.jks
                  </url>
```

```
                  <password system-property="coherence.security.
                     password"/>
               </key-store>
               <password system-property="coherence.security.password"/>
            </identity-manager>
            <trust-manager>
               <algorithm>PeerX509</algorithm>
               <key-store>
                  <url system-property="coherence.security.keystore">
                     file:keystore.jks
                  </url>
                  <password system-property="coherence.security.
                     password"/>
               </key-store>
            </trust-manager>
            <socket-provider>tcp</socket-provider>
         </ssl>
      </socket-provider>
   </socket-providers>
</cluster-config>
...
```

As configured, the predefined SSL socket provider requires a Java keystore named `keystore.jks` that is found on the classpath. Use an operation override file to modify any socket provider values as required. The `coherence.security.keystore` and `coherence.security.password` system properties override the keystore and password instead of using the operational override file. For example:

```
-Dcoherence.security.keystore=/mykeystore.jks -Dcoherence.security.password=password
```

Note:

Ensure that certificates for all nodes in the cluster have been imported into the keystore.

To use the predefined SSL socket provider, override the `<socket-provider>` element in the `<unicast-listener>` configuration and reference the SSL socket provider using its `id` attribute. The following example configures a unicast listener to use the predefined SSL 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>
      <unicast-listener>
         <socket-provider
            system-property="coherence.socketprovider>ssl
         </socket-provider>
         <well-known-addresses>
            <socket-address id="1">
               <address system-property="coherence.wka">198.168.1.5
               </address>
               <port system-property="coherence.wka.port">8088</port>
            </socket-address>
         </well-known-addresses>
```

```
      </unicast-listener>
    </cluster-config>
</coherence>
```

Using SSL to Secure Extend Client Communication

This section provides instructions for configuring SSL to secure communication between extend clients and cluster proxies. The configuration examples in this section assume that valid digital certificates for all clients and servers have been created as required and that the certificates have been signed by a Certificate Authority (CA). The digital certificates must be found in an identity store, and the trust keystore must include the signing CA's digital certificate. Use self-signed certificates during development as needed. See "Using SSL to Secure TCMP Communication" for instructions on using SSL between cluster members.

This section includes the following topics:

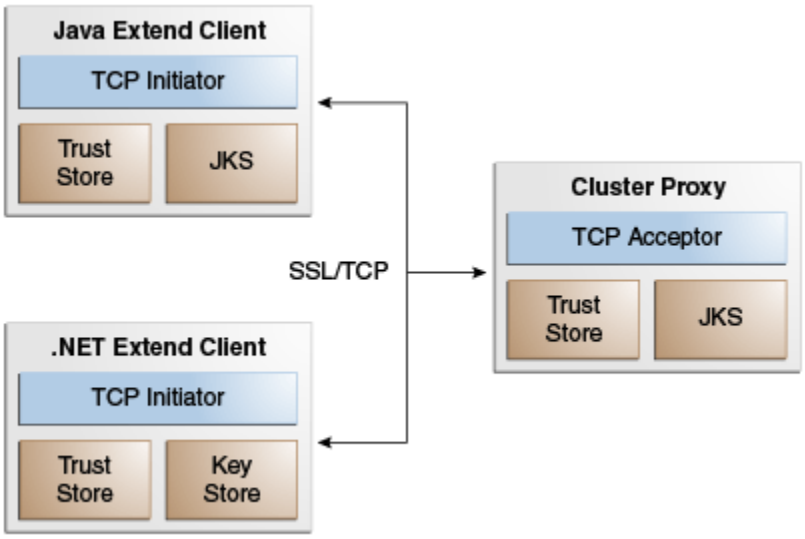
- Overview of Using SSL to Secure Extend Client Communication
- Configuring a Cluster-Side SSL Socket Provider
- Configuring a Java Client-Side SSL Socket Provider
- Configuring a .NET Client-Side Stream Provider

Overview of Using SSL to Secure Extend Client Communication

SSL is used to secure communication between extend clients and extend proxies. SSL requires configuration on both the client side and the cluster side. SSL is supported for both Java and .NET clients but not for C++ clients.

Figure 6-2 shows a conceptual view of extend clients using SSL to communicate with a cluster proxy. The clients and the proxy include a trust keystore and an identity keystore that contain digital certificates that are used for authentication. Extend clients typically use one-way authentication in which only proxies authenticate with clients, and clients remain anonymous to proxies.

Figure 6-2 Conceptual Architecture of SSL with Oracle Coherence*Extend



Configuring a Cluster-Side SSL Socket Provider

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:

- Per Proxy Service – Each proxy service defines an SSL socket provider configuration or references a predefined configuration that is included in the operational configuration file.
- All Proxy Services – All proxy services use the same global SSL socket provider configuration. A proxy service that provides its own configuration overrides the global configuration. The global configuration can also reference a predefined configuration that is included in the operational configuration file.

Configure an SSL Socket Provider per Proxy Service

To configure an SSL socket provider for a proxy service, add a `<socket-provider>` element within the `<tcp-acceptor>` element of each `<proxy-scheme>` definition. See *Developing Applications with Oracle Coherence* for a detailed reference of the `<socket-provider>` element.

Example 6-3 demonstrates a proxy scheme that configures an SSL socket provider that uses the default values for the `<protocol>` and `<algorithm>` elements (`TLS` and `SunX509`, respectively). These are shown only for completeness; you can omit them when you use the default values.

Example 6-3 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 not a trust keystore.

Note:

If the proxy server is configured with a trust manager, then the client must use two-way SSL authentication, because the proxy expects a digital certificate to be exchanged. Make sure a trust manager is not configured if you want to use one-way SSL authentication.

Example 6-3 Sample Cluster-Side SSL Configuration

```
...
<proxy-scheme>
   <service-name>ExtendTcpSSLProxyService</service-name>
   <acceptor-config>
      <tcp-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>
               <socket-provider>tcp</socket-provider>
            </ssl>
         </socket-provider>
         <local-address>
            <address>192.168.1.5</address>
            <port>9099</port>
         </local-address>
      </tcp-acceptor>
   </acceptor-config>
   <autostart>true</autostart>
</proxy-scheme>
...
```

The following example references an SSL socket provider configuration that is defined in the `<socket-providers>` node of the operational deployment descriptor by specifying the `id` attribute (`ssl`) of the configuration. See *Developing Applications with Oracle Coherence* for a detailed reference of the `<socket-providers>` element.

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 "Using the Predefined SSL Socket Provider" 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>ExtendTcpSSLProxyService</service-name>
   <acceptor-config>
      <tcp-acceptor>
         <socket-provider>ssl</socket-provider>
         <local-address>
            <address>192.168.1.5</address>
            <port>9099</port>
         </local-address>
      </tcp-acceptor>
   </acceptor-config>
   <autostart>true</autostart>
</proxy-scheme>
...
```

Configure an SSL Socket Provider for All Proxy Services

To configure a global SSL socket provider for use by all proxy services, use a `<socket-provider>` element within the `<defaults>` element of the cache

configuration file. With this approach, no additional configuration is required within a proxy scheme definition. See *Developing Applications with Oracle Coherence* for a detailed reference of the `<default>` element.

The following example uses the same SSL socket provider configuration from Example 6-3 and configures it for all proxy services.

```
<?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>
         <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>
            <socket-provider>tcp</socket-provider>
         </ssl>
      </socket-provider>
   </defaults>
   ...
```

The following example configures a global SSL socket provider by referencing an SSL socket provider configuration that is defined in the operational deployment descriptor:

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

Configuring a Java Client-Side SSL Socket Provider

Configure SSL in the client-side cache configuration file by defining an SSL socket provider for a remote cache scheme and, if required, for a remote invocation scheme. There are two options for configuring an SSL socket provider, depending on the level of granularity that is required:

- Per Remote Service – Each remote service defines an SSL socket provider configuration or references a predefined configuration that is included in the operational configuration file.

- All Remote Services – All remote services use the same global SSL socket provider configuration. A remote service that provides its own configuration overrides the global configuration. The global configuration can also reference a predefined configuration that is included in the operational configuration file.

Configure an SSL Socket Provider per Remote Service

To configure an SSL socket provider for a remote service, add a `<socket-provider>` element within the `<tcp-initiator>` element of a remote scheme definition. See *Developing Applications with Oracle Coherence* for a detailed reference of the `<socket-provider>` element.

Example 6-4 demonstrates a remote cache scheme that configures a socket provider that uses SSL. The example uses the default values for the `<protocol>` and `<algorithm>` elements (`TLS` and `SunX509`, respectively). These are shown only for completeness; you can omit them when you use the default values.

Example 6-4 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 client configuration must include a trust keystore but need not include an identity keystore, which indicates that the client does not exchange its digital certificate to the proxy and remains anonymous. The client's trust keystore must include the CA's digital certificate that was used to sign the proxy's digital certificate.

Note:

If the proxy server is configured with a trust manager, then the client must use two-way SSL authentication, because the proxy expects a digital certificate to be exchanged. Remove the proxy's trust manager configuration if you want to use one-way SSL authentication.

Example 6-4 Sample Java Client-Side SSL 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">
   <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>
               <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>
                     <socket-provider>tcp</socket-provider>
                  </ssl>
               </socket-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>
```

The following example references an SSL socket provider configuration that is defined in the `<socket-providers>` node of the operational deployment descriptor by specifying the `id` attribute (`ssl`) of the configuration. See *Developing Applications with Oracle Coherence* for a detailed reference of the `<socket-providers>` element.

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 for "Using the Predefined SSL Socket Provider" 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.

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

Configure an SSL Socket Provider for All Remote Services

To configure a global SSL socket provider for use by all remote services, use a `<socket-provider>` element within the `<defaults>` element of the cache configuration file. With this approach, no additional configuration is required within a remote scheme definition. See *Developing Applications with Oracle Coherence* for a detailed reference of the `<default>` element.

The following example uses the same SSL socket provider configuration from Example 6-4 and configures it for all remote services.

```
<?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>
         <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>
          <socket-provider>tcp</socket-provider>
       </ssl>
    </socket-provider>
 </defaults>
 ...
```

The following example configures a global SSL socket provider by referencing an SSL socket provider configuration that is defined in the operational deployment descriptor:

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

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. See "Generating Windows SSL Artifacts" for a generic example. For more information about managing certificates, see

`http://technet.microsoft.com/en-us/library/cc782338(WS.10).aspx`

Example 6-5 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.

Example 6-5 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>Tls</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>
```

Using SSL to Secure Federation Communication

Communication between cluster participants in a federation can be secured using SSL Communication is secured between federated service members and requires SSL configuration on each cluster participant that requires SSL security. This section assumes that valid digital certificates for all servers that run the federated service have been created as required and that the certificates have been signed by a Certificate Authority (CA). The digital certificates must be found in an identity store, and the trust keystore must include the signing CA's digital certificate.

To use SSL to secure federation communication:

1. Edit the operational override file on each cluster and include an SSL socket provider definition or use and existing SSL socket provider definition. For details about configuring SSL socket providers, see "Define an SSL Socket Provider".

2. Edit the federated cache scheme on each cluster to use the SSL socket provider definition. For example:

```
<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>mySSLConfig</socket-provider>
   <topologies>
```

```
         <topology>
            <name>MyTopology</name>
         </topology>
      </topologies>
   </federated-scheme>
```

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>
```

7

Securing Oracle Coherence in Oracle WebLogic Server

This chapter provides instructions for using authentication and authorization to secure Oracle Coherence in an Oracle WebLogic Server domain. The instructions are specific to the Oracle WebLogic Server Administration Console and do not include details for the Oracle WebLogic Scripting Tool (WLST). For details on using the Oracle WebLogic Server Administration Console, see . For details on using WLST, see .

This chapter includes the following sections:

Overview of Securing Oracle Coherence in Oracle WebLogic Server

There are several security features that can be used when deploying Oracle 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:

- Oracle Coherence access controllers – provides authorization between cluster members
- Oracle WebLogic Server authorization – provides authorization to Oracle Coherence caches and services
- Oracle 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 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. For details on the access controller, 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. For details on configuring keystores, identity and trust in Oracle WebLogic Server, see *Administering Security for Oracle WebLogic Server*.

Enabling the Oracle Coherence Security Framework

To enable the security framework in an Oracle WebLogic server domain:

1. From the Summary of Coherence Clusters page, click a Coherence Cluster to configure its settings.
2. From the cluster's settings page, click the Security tab.
3. From the General tab, click the Security Framework Enabled option to enable the security framework.
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 Settings page 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. From the Summary of Coherence Clusters page, click a Coherence Cluster to configure its settings.
2. From the cluster's settings page, click the Security tab.
3. From the General tab, click the Security Framework Enabled option to enable the security framework if it has not already been enabled.
4. In the Private Key Alias field, enter the alias for the identity.
5. In the Private Key Pass Phrase field, enter the password for the identity.

6. In the Confirm Private Key Pass Phrase field, re-enter the password.
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. For details on creating roles and policies in Oracle WebLogic Server, see *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 names 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.

Specifying Cache Authorization

Oracle WebLogic Server authorization can be used to restrict access to specific Oracle Coherence caches. To specify cache authorization:

1. From the Summary of Coherence Clusters page, click a Coherence Cluster to configure its settings.
2. From the cluster's settings page, click the Security tab and Caches subtab.
3. Click **New** to define a cache on which roles and polices will be defined. The Create a Coherence Cache page displays.
4. Enter the name of a cache in the Name field. The name of the cache must exactly match the name of the cache used in an application.
5. Click **Finish**. The cache is listed on the Coherence Caches page.
6. Click the cache to access its settings page where you can define scoped roles and policies using the Roles and Policies tab, respectively. 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. For details on specifying scoped roles and policies, see *Oracle WebLogic Server Administration Console Online Help*.

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. From the Summary of Coherence Clusters page, click a Coherence Cluster to configure its settings.
2. From the cluster's settings page, click the Security tab and Services subtab.
3. Click **New** to define a service to which roles and polices will be defined. The Create a Coherence Service page displays.
4. Enter the name of a service in the Name field. 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`.

5. Click **Finish**. The service is listed on the Coherence Services page.
6. Click the service to access its settings page where you can define scoped roles and policies using the Roles and Policies tab, respectively. 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. For details on specifying scoped roles and policies, see *Oracle WebLogic Server Administration Console Online Help*.

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:

Once an identity is established, an authorization policy should be used to restrict that identity to specific caches and services. For details on configuring authorization, 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). For details on creating transformer and asserter implementations, see "Using Identity Tokens to Restrict Client Connections".

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. For details on enabling an identity transformer, 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, the 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.

To enable an identity asserter for a cluster:

1. From the Summary of Coherence Clusters page, click a Coherence Cluster to configure its settings.
2. From the cluster's settings page, click the Security tab.
3. From the General tab, use the Identity Assertion fields to enter the fully qualified name of the asserter class and, if required, any class constructor arguments. For example, to use the default identity asserter, enter:

   ```
   com.tangosol.net.security.DefaultIdentityAsserter
   ```

4. Click **Save**.
5. Restart the cluster servers or redeploy the GAR for the changes to take effect.

8

Securing Oracle Coherence REST

This chapter provides instructions for securing Oracle Coherence REST and does not include general instructions for using Oracle Coherence REST. For detailed information on using Oracle Coherence REST, see .

This chapter includes the following sections:

Overview of Securing Oracle Coherence REST

Oracle Coherence REST security uses both authentication and authorization to restrict access to cluster resources. 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.

Security for Oracle Coherence REST is disabled by default and is enabled as required. Authentication and authorization are configured separately. Authentication is configured in a cache configuration file using the `<auth-method>` element within the `<http-acceptor>` element. For details on the `<auth-method>` element, see the *Developing Applications with Oracle Coherence*. For detailed information about implementing authorization, see "Implementing Extend Client Authorization".

Using HTTP Basic Authentication with Oracle 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 as described in this section.

To specify basic authentication, 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. Refer to the JAAS Reference Guide for instructions about JAAS and creating a `LoginModule` implementation:

http://docs.oracle.com/javase/7/docs/technotes/guides/security/jaas/JAASRefGuide.html

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. For details, see:

http://docs.oracle.com/javase/7/docs/technotes/guides/security/jaas/JAASLMDevGuide.html

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. For instructions about creating a keystore, see "Generating Java SSL Artifacts".

Using SSL Authentication With Oracle Coherence REST

SSL provides an authentication mechanism that relies on digital certificates and encryption keys to establish both identity and trust. For an overview of SSL, including generating SSL artifacts, see "Overview of SSL".

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.

To specify SSL authentication, add an `<auth-method>` element, within the `http-acceptor` element, that is set to `cert`.

```
<proxy-scheme>
   <service-name>RestHttpProxyService</service-name>
   <acceptor-config>
      <http-acceptor>
         ...
         <auth-method>cert</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 *Developing Applications with Oracle Coherence* for a detailed reference of the `<socket-provider>` element.

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 *Developing Applications with Oracle Coherence* for a detailed reference of the `<socket-providers>` element.

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 "Using the Predefined SSL Socket Provider" 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. For details about creating Jersey clients, see `http://jersey.java.net/nonav/documentation/latest/index.html`.

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.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

The use of HTTP basic authentication does not preclude the use of SSL authentication. That is, both HTTP basic authentication and SSL can be used together for added protection. For details about setting up both SSL and HTTP basic authentication, 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.