Oracle® Coherence

Security Guide

Release 3.7.1

E22841-01

September 2011

Explains key security concepts and provides instructions for implementing various levels of security for both Coherence clusters and Coherence*Extend clients.

ORACLE®

Oracle Coherence Security Guide, Release 3.7.1

E22841-01

Primary Author: Joe Ruzzi

Contents

5 Using SSL to Secure Communication

Preface

Welcome to *Oracle Coherence Security Guide*. This document explains key security concepts and provides instructions for implementing various levels of security for both Coherence clusters and Coherence*Extend clients.

Audience

This guide is intended for the following audiences:

- Primary Audience – Application Developers and Operations who want to secure a Coherence cluster and secure Coherence*Extend client communication with the cluster.
- Secondary Audience – System Architects who want to understand the options and architecture for securing a Coherence cluster and Coherence*Extend clients.

The audience must be familiar with Coherence and Coherence*Extend to use this guide. In addition, users must be familiar with Java and SSL to use this guide. The examples in this guide require the installation and use of the Oracle Coherence product, including Coherence*Extend. The use of an IDE is not required to use this guide, but 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 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 that are included in the Oracle Coherence documentation set:

- *Oracle Coherence Administrator's Guide*
- *Oracle Coherence Developer's Guide*

- *Oracle Coherence Client Guide*
- *Oracle Coherence Getting Started Guide*
- *Oracle Coherence Integration Guide for Oracle Coherence*
- *Oracle Coherence Management Guide*
- *Oracle Coherence Tutorial for Oracle Coherence*
- *Oracle Coherence User's Guide for Oracle Coherence*Web*
- *Oracle Coherence Java API Reference*
- *Oracle Coherence C++ API Reference*
- *Oracle Coherence .NET API Reference*
- *Oracle Coherence Release Notes for Oracle Coherence*

Conventions

The following text conventions are used in this document:

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

1

Introduction to Coherence Security

This chapter provides an introduction to Coherence security features. Coherence security features provide varying levels of security and can be implemented as required. The security features include industry standards and Coherence-specific features.

> **Note:** This guide does not provide detailed instructions for setting up a cluster or creating Coherence*Extend clients. See the *Oracle Coherence Developer's Guide* and *Oracle Coherence Client Guide*, respectively, for details on setting up a cluster or creating Coherence*Extend clients.

The following sections are included in this chapter:

Conceptual Overview of Coherence Security

Coherence security includes securing both cluster members and extend clients. Security is enabled as required based on the application or cluster implementation and an organization's security concerns and security tolerances. This section provides a brief discussion of each security feature and describes the area of concern that each addresses. The features are presented here (and throughout the book) from basic security measures to more advanced security measures.

Java Policy Security

Coherence provides a Java security policy file that contains the minimum set of security permissions necessary to run Coherence. The file is edited to change the permissions based on an application's requirement. The security policy protects against malicious use and alterations of the Coherence library and configuration files. See Chapter 2, "Enabling General Security Measures," for details.

Host-Based Authorization

Host-based authorization is used to explicitly specify which hosts can become members of a cluster and which extend clients can 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 Chapter 2, "Enabling General Security Measures," for details.

Client Suspect Protocol

The client suspect protocol is used to automatically determine if an extend client is acting malicious. If the client is determined to be malicious, it is automatically blocked from connecting to a cluster. The suspect protocol protects against denial of service attacks. See Chapter 2, "Enabling General Security Measures," for details.

Client Identity Tokens

Client identity tokens are used to control whether an extend client can access the cluster. Only clients that present a valid token are permitted to connect to a proxy server. This feature can leverage existing client authentication implementations. Identity tokens protect against unwanted or malicious clients from accessing the cluster. See Chapter 4, "Securing Extend Client Connections," for details.

Client Authorization

Client authorization is used to control which actions a particular user can perform based on their access control rights. Client authorization is performed on a proxy server and occurs before an extend client is allowed to access a resource (cache, cache service, or invocation service). Client authorization is application-specific and protects against unauthorized use of cluster resources. See Chapter 4, "Securing Extend Client Connections," for details.

Access Controller Security Framework

The access controller manages access to clustered resources, such as clustered services and caches, and controls the operations that a user can perform on those resources. Cluster members use login modules to provide proof of identity and encrypting/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 protects against malicious cluster members from accessing and also creating clustered resources. See Chapter 3, "Using the Access Controller," for details.

SSL

SSL is used to secure TCMP communication between cluster nodes and the TCP communication between Coherence*Extend clients and proxies. SSL uses digital signatures to establish identity/trust and key-based encryption to ensure data is secure. SSL is an industry standard that is used to protect against unauthorized access and data tampering by malicious clients and cluster members. See Chapter 5, "Using SSL to Secure Communication," for details.

Overview of Security Configuration

Coherence security features are generally enabled and configured in either an operational override file or the cache configuration file. See *Oracle Coherence Developer's Guide* for detailed information on Coherence configuration.

- Operational Override File – The `tangosol-coherence-override.xml` file is used to override the operational deployment descriptor, which is used to specify the operational and run-time settings that are used to create, configure and maintain clustering, communication, and data management services. This file is used to configure security for the cluster. That is, security between cluster members.
- Cache Configuration File – The `coherence-cache-config.xml` file is the default cache configuration file and is used to specify the various types of caches

that can be used within a cluster. This configuration file is used to configure security for Coherence*Extend. A cache configuration file is required on both the client-side and cluster-side for Coherence*Extend. See *Oracle Coherence Client Guide* for details on setting up Coherence*Extend.

2

Enabling General Security Measures

This chapter provides instructions for general security steps that can be taken to help secure a Coherence environment. The steps can be considered first-line security measures that should always be set if possible.

The following sections are included in this chapter:

Specifying Coherence Privileges in the Java Security Policy File

The minimum set of privileges required for Coherence to function are specified in the `security.policy` file which is included as part of the Coherence installation. This file can be found in *`COHERENCE_HOME`*`/lib/security/security.policy`.

The policy file format is fully described in Java SE Security Guide. See

`http://download.oracle.com/javase/6/docs/technotes/guides/security/permissions.html`

To specify Coherence privileges:

1. Enter the minimum set of privileges in the policy file.

   For example:

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

2. Sign the binaries using the JDK `jarsigner` tool, for example:

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

   and then additionally protected in the policy file:

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

3. Use operating system mechanisms to protect all relevant files such as policy format, coherence binaries, and permissions from malicious modifications.

4. To use the security policy file, turn on the Java Security Manager by defining the `java.security.manager` system property and setting the `java.security.policy` system property to the location of this security policy file. You must also set the `tangosol.home` system property to *COHERENCE_HOME*. For example:

```
-Djava.security.manager
-Djava.security.policy=c:/tangosol/lib/security/security.policy
-Dtangosol.home=c:/tangosol
```

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

Using Host-Based Authorization

Host-based authorization is a type of access control that allows only specified hosts to connect to a cluster. The feature can be used for both cluster member connections and extend client connections. This type of access control is ideal for environments were known hosts are joining or accessing the cluster.

The following topics are in this section:

- Specifying Cluster Member Authorized Hosts
- Specifying Extend Client Authorized Hosts
- Using a Filter Class to Determine Authorization

Specifying Cluster Member Authorized Hosts

A cluster's default behavior is to allow any host to connect to the cluster and become a cluster member. This behavior can be changed to only allow hosts to connect to the cluster based on their host name or IP address. A customized filter can also be created to determine whether to accept a particular cluster member.

Authorized hosts for a cluster are configured in an operational override file using the `<authorized-hosts>` element within the `<cluster-config>` element. Specific addresses are entered using the `<host-address>` element or a range of addresses can be defined using the `<host-range>` element.

The following example configures a cluster to only accept 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:

```
<?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>192.168.0.5</host-address>
         <host-address>192.168.0.6</host-address>
         <host-range>
            <from-address>192.168.0.10</from-address>
            <to-address>192.168.0.20</to-address>
```

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

Specifying Extend Client Authorized Hosts

The extend proxy's default behavior is to accept all extend client connections. This behavior can be changed to only allow client connections based on their host name or IP address. A customized filter can also be created to determine whether to accept a particular client.

Authorized hosts for a cluster are configured in a cache configuration file using the `<authorized-hosts>` element within the `<tcp-acceptor>` element of a proxy scheme definition. Specific addresses are entered using the `<host-address>` element. A range of addresses can be defined using the `<host-range>` element.

The following example configures an extend proxy to only accept client connections from client's 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:

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

Using a Filter Class to Determine Authorization

A filter class can determine whether to accept a particular host connection. A filter class can be used for both extend client connections and cluster member connections. 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. Initialization parameters for the implementation class can also be set using the `<init-params>` element. See *Oracle Coherence Java API Reference* for details on the `Filter` interface.

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

```
<authorized-hosts>
   <host-address>192.168.0.5</host-address>
```

```
   <host-address>192.168.0.6</host-address>
   <host-range>
      <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

Extend clients that operate outside of acceptable limits are considered rogue clients. Rogue clients can be slow responding clients or abusive clients that attempt to overuse a proxy— as is the case with denial of service attacks. In both cases, the proxy could run out of memory and become unresponsive.

The suspect protocol is used to safeguard against such abuses. The suspect algorithm monitors client connections looking for abnormally slow or abusive clients. When a rouge 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 are set to determine when a client is suspect, when it has returned to normal, or when it is considered rogue.

The suspect protocol is configured within the `<tcp-acceptor>` element of a proxy scheme definition. See "tcp-acceptor" in the *Oracle Coherence Developer's Guide* 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 the 95 MB or 60000 messages, then the client is considered unsafe and the connection with the client is closed:

```
<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 the Access Controller

The Access Controller security framework in Coherence is based on the concept of a clustered access controller, which can be turned on (activated) by a configurable parameter or command line attribute.

The following sections are included in this chapter:

- Overview of the Access Controller
- Enabling the Default Access Controller Implementation

> **Note:** This chapter does not cover SSL. See Chapter 5, "Using SSL to Secure Communication," for detailed instructions for using SSL.

Overview of the Access Controller

The Access Controller manages access to clustered resources, such as clustered services and caches and controls operations that include (but are not limited to) the following:

- creating a new clustered cache or service
- joining an existing clustered cache or service
- destroying an existing clustered cache

The Access Controller serves three purposes:

- grant or deny access to a protected clustered resource based on the caller's permissions
- encrypt outgoing communications based on the caller's private credentials
- decrypt incoming communications based on the caller's public credentials

Coherence uses a local Login Module (see *JAAS Reference Guide* for details) to authenticate the caller and an Access Controller on one or more cluster nodes to verify the caller's access rights.

The Access Controller is a pluggable component that could be declared in the Coherence deployment descriptor, `tangosol-coherence.xml`. The specified class must implement the `com.tangosol.net.security.AccessController` interface.

Coherence provides a default Access Controller implementation that is based on the Key Management infrastructure that is shipped as a standard part of Sun's JDK. See"Enabling the Default Access Controller Implementation" on page 3-3.

Each clustered service in Coherence maintains a concept of a senior service member (cluster node), which serves as a controlling agent for a particular service. While the senior member does not have to consult anyone when accessing a clustered resource, any junior node willing to join that service has to request and receive a confirmation from the senior member, which in turn notifies all other cluster nodes about the joining node.

Since Coherence is a system providing distributed data management and computing, the security subsystem is designed to operate in a partially hostile environment. We assume that when there is data shared between two cluster nodes either node could be a malicious one - lacking sufficient credentials to join a clustered service or obtain access to a clustered resource.

Let's call a cluster node that may try to gain unauthorized access to clustered resources by using nonstandard means as a "malicious" node. The means of such an access could vary. They could range from attempts to get protected or private class data using reflection, replacing classes in the distribution (`coherence.jar` or other application binaries), modifying classes on-the-fly using custom class loader(s) and so on. Alternatively, a cluster node that never attempts to gain unauthorized access to clustered resources by using nonstandard means is called a "trusted" node. It's important to note that even a trusted node may attempt to gain access to resources without having sufficient rights, but it does so in a standard way by using the exposed standard API.

File system mechanisms (the same that is used to protect the integrity of the Java run-time libraries) and standard Java security policy could be used to resolve an issue of guarantying the trustworthiness of a given single node. In a case of inter-node communications there are two dangers to consider:

- A malicious node surpasses the local access check and attempts to join a clustered service or gain access to a clustered resource controlled by a trusted node
- A malicious node creates a clustered service or clustered resource becoming its controller

To prevent either of these two scenarios from occurring Coherence uses two-way encryption algorithm: all client requests must be accompanied by the proof of identity and all service responses must be accompanied by the proof of trustworthiness.

Proof of Identity

In the case of an active Access Controller, the client code can use the following construct to authenticate the caller and perform 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
        ...
        }
    };
Security.runAs(subject, action);
```

During the `login` call, Coherence uses JAAS that runs on the caller's node to authenticate the caller. In a case of successful authentication, it uses the local Access Controller to:

- Determine whether the local caller has sufficient rights to access the protected clustered resource (local access check)
- Encrypt the outgoing communications regarding the access to the resource with the caller's private credentials retrieved during the authentication phase
- Decrypt the result of the remote check using the requester's public credentials
- In the case that access is granted verify whether the responder had sufficient rights to do so

The encrypt step (above) serves the role of the proof of identity for the responder preventing a malicious node pretending to pass the local access check phase.

There are two alternative ways to provide the client authentication information. First, a reference to a `CallbackHandler` could be passed instead of the user name and password. Second, a previously authenticated Subject could be used, which could become handy when Coherence is used by a Java EE application that could retrieve an authenticated Subject from the application container.

If a caller's request comes without any authentication context, Coherence instantiates and call a `CallbackHandler` implementation declared in the Coherence operational descriptor to retrieve the appropriate credentials. However, that "lazy" approach is much less efficient, since without externally defined call scope, every access to a protected clustered resource forces repetitive authentication calls.

Proof of Trustworthiness

Every clustered resource in Coherence is created by an explicit API call. A senior service member retains the private credentials that are presented during that call as a proof of trustworthiness. When the senior service member receives an access request to a protected clustered resource, it use the local Access Controller to:

- Decrypt the incoming communication using the remote caller's public credentials
- Encrypt the response of access check using the private credentials of the service
- Determine whether the remote caller has sufficient rights to access the protected clustered resource (remote access check)

Since the requester accepts the response as valid only after decrypting it, the last step in this cycle serves a role of the proof of trustworthiness for the requester preventing a malicious node pretending to be a valid service senior.

Enabling the Default Access Controller Implementation

Coherence ships with a default Access Controller implementation that uses a standard Java keystore. The implementation class is the `com.tangosol.net.security.DefaultController` class and is configured in the Coherence operational deployment descriptor.

```
<security-config>
  <enabled system-property="tangosol.coherence.security">false</enabled>
  <login-module-name>Coherence</login-module-name>
  <access-controller>
    <class-name>com.tangosol.net.security.DefaultController</class-name>
    <init-params>
```

```
      <init-param id="1">
        <param-type>java.io.File</param-type>
        <param-value>./keystore.jks</param-value>
      </init-param>
      <init-param id="2">
        <param-type>java.io.File</param-type>
        <param-value>./permissions.xml</param-value>
      </init-param>
    </init-params>
  </access-controller>
  <callback-handler>
    <class-name/>
  </callback-handler>
</security-config>
```

The default access controller implementation is not enabled by default. To enable the default implementation, override the `<enabled>` element within the `<security-config>` node in an operational override file and set it to `true`. For example:

```
<security-config>
   <enabled>true</enabled>
</security-config>
```

The default access controller implementation can also by enabled by setting the `tangosol.coherence.security` system property to `true`.

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

The `<login-module-name>` element serves as the application name in a login configuration file (see JAAS Reference Guide for complete details). Coherence is shipped with a Java keystore (JKS) based login module that is contained in the `coherence-login.jar`, which depends only on standard Java run-time classes and could be placed in the JRE's `lib/ext` (standard extension) directory. The corresponding login module declaration would look like:

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

The `<access-controller>` element defines the Access Controller implementation that takes two parameters to instantiate.

- The first parameter is a path to the same keystore that is used by both controller and login module.
- The second parameter is a path to the access permission file (see discussion below).

The <callback-handler> is an optional element that defines a custom implementation of the javax.security.auth.callback.CallbackHandler interface that would be instantiated and used by Coherence to authenticate the client when all other means are exhausted.

Two more steps have to be performed to make the default Access Controller implementation usable in your application:

1. Create a keystore with necessary principals.
2. Create the permissions file that would declare the access right for the corresponding principals.

Consider the following example that creates three principals: admin to be used by the Java Security framework; manager and worker to be used by 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
```

Consider the following example that 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>
```

4

Securing Extend Client Connections

Coherence*Extend includes an additional set of features that are used to secure communication between extend clients and extend proxies. See *Oracle Coherence Client Guide* for details on creating Coherence*Extend clients.

The following sections are included in this chapter:

Using Identity Tokens to Restrict Client Connections

Extend client can be restricted from accessing a cluster by using an identity token. 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.

On the extend client, identity tokens are created by an identity transformer and sent as part of the connection request. On the cluster side, an identity asserter is used to validate the identity token before the connection is accepted. Coherence*Extend includes a default identity transformer (`DefaultIdentityTransformer`) and identity asserter (`DefaultIdentityAsserter`) that use the Subject (Java) or Principal (.NET) for the identity token. The default behavior can be overridden by providing custom identity transformer and identity asserter implementations and enabling them in the Coherence operational override file.

Note:

- At run time, 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" on page 4-4 for more information on using security object types other than the types that are predefined in POF.

The following topics are included in this section:

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

Creating a Custom Identity Transformer

An identity transformer is a client-side component that converts a Subject, or Principal, into an identity token that can be passed to an extend proxy. The identity token can be any type of object useful for identity validation; it is not required to be a well-known security type. In addition, the identity token can be specific to a remote service to support clients that connect to multiple proxy servers and authenticate to each proxy server in a unique fashion. At run time, the token is serialized and sent as part of the extend connection request.

> **Note:** Identity tokens that are of a type that Coherence can serialize are automatically serialized. For .NET and C++ clients, the type must be a POF type. See "Using Custom Security Types" on page 4-4 for more information on using security object types other than the types that are predefined in POF.

For Java and C++, the identity transformer must implement the `IdentityTransformer` interface. C# clients implement the `IIdentityTransformer` interface.

Example 4–1 is a Java implementation that restricts client access by requiring a client to supply a password to access the proxy. The `IdentityTransformer` gets a password from a system property on the client and returns it as an identity token.

Example 4–1 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");
        }
    }
```

If client authentication is being done, a new Principal could be added to the Subject, with the Principal name as the password. The password Principal could be added to the Subject during JAAS authentication by modifying an existing JAAS login module or by adding an additional required login module that would add the password Principal. JAAS allows multiple login modules each of which can modify the Subject. Similarly, in .NET a password identity could be added to the Principal. The asserter on the cluster-side would then validate the "normal" Principals plus validate the password Principal. See "Creating a Custom Identity Asserter" below.

Enabling a Custom Identity Transformer

An identity transformer implementation is enabled in the client-side `tangosol-coherence-override.xml` file using the `<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

An identity asserter is 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 can validate identity tokens unique for each proxy service to support multiple means of token validation.

The token gets 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. For Java and C++, the identity asserter must implement the `IdentityAsserter` interface. C# clients implement the `IIdentityAsserter` interface.

Example 4–2 is a Java implementation that checks a security token to ensure a valid password is given. In this case, the password is checked against a system property on the cache server. This asserter implementation is used for the identity transformer sample in Example 4–1.

Example 4–2 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 creating an identity asserter. For example, the asserter could reject connections based on a list of Principals, check role Principals, validate signed Principal name, and so on. The asserter can block any connection attempt that do not prove the correct identity.

Enabling a Custom Identity Asserter

An identity asserter implementation is enabled in the cluster-side `tangosol-coherence-override.xml` file using the `<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/deserialized using Portable Object Format (POF) when they are passed between extend clients and extend proxies. Security objects of types that are predefined in POF can be used without any configuration or programming changes. Security objects of custom types that are not predefined in POF (for example, when using Kerberos authentication) causes an error.

For custom security types, an application can choose to either convert the custom type or define the type in POF:

Convert the Type

In this approach, a custom identity transformer 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, a custom identity asserter would convert it back (after validation) to a subject.

For example, a subject may contain credentials that are not serialized. The identity transformer would extract the credential and convert it to a character array, returning that array as the token. On the proxy server, the identity asserter would convert the character array to the proper credential type, validate it, and then construct a subject to return.

Define the Custom Type in POF

In this approach, custom security types are defined in a POF configuration file. The type must be defined in both the client's POF configuration file and the POF configuration file on the proxy server.

For detailed information on using POF with Java, see *Oracle Coherence Developer's Guide*. For more information on using POF with C++, see *Oracle Coherence Client Guide*. For more information on using POF with C#, see *Oracle Coherence Client Guide*.

Understanding Custom Identity Token Interoperability

Solutions that choose to 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 rolling out a custom identity token solution.

Coherence Upgrades

Interoperability issues may occur during the process of upgrading Coherence. In this scenario, there may be different client versions that interoperate with different proxy server versions. In particular:

- When a 3.5 extend client is connecting to a 3.6 extend proxy, the custom identity asserter that is implemented on the extend proxy must be able to handle identity tokens sent by the 3.5 extend client. A 3.5 extend client sends either a `null` token or a `Subject`. The custom identity asserter must be prepared to handle those token types in addition to any custom tokens originating from a 3.6 extend client.
- Conversely, when a 3.6 extend client is connecting to a 3.5 extend proxy, the client must not use a custom identity transformer that sends a token that the 3.5 extend proxy cannot handle. The client must send either a `null` token or a `Subject`.

Custom Identity Token Rollout

Interoperability issues may occur between extend clients and extend proxies when rolling out a custom identity token solution. In this scenario, as extend proxies are migrated to use a custom identity asserter, there are proxies that continue to use the default asserter until the roll out is completed. Likewise, as extend clients are migrated to use a custom identity transformer, there are clients that continue to use the default transformer until the roll out is completed. In both cases, the extend clients and extend proxies must be able to handle a `null` token or a `Subject` until the rollout is complete.

A possible strategy for such scenarios may be to have a custom identity asserter that accepts `null` or `Subject` tokens temporarily as clients are updated. The identity asserter could check an external source for a policy that indicates whether those tokens are accepted. After all clients have been updated to use a custom token, the policy could be changed and the identity asserter would no longer accept those 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 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" on page 4-4 for more information on using security object types other than the types that are predefined in POF.

For example, if the "trader" user calls `CacheFactory.getCache("trade-cache")` and the "manager" user calls `CacheFactory.getCache("trade-cache")`, each user gets a different remote cache reference object. Since 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, this feature is enabled in the client-side `tangosol-coherence-override.xml` file using the `<subject-scope>` element within the `<security-config>` node. The following example enables subject scoping and ensures each subject gets a unique remote cache and remote invocation service reference.

```
<?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, this feature is enabled in the client-side `tangosol-coherence-override.xml` file using the `<principal-scope>` element within the `<security-config>` node. The following example enables subject scoping and ensures each subject gets a unique remote cache and remote invocation service reference.

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

Implementing Extend Client Authorization

Authorization is used to control which actions a particular user can perform based on their access control rights. In Coherence*Extend, authorization is implemented with interceptor classes. 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 and must provide the necessary authorization logic before passing the request to the proxied resources.

The following topics are included in this section:

- Creating Authorization Interceptor Classes

- Enabling Authorization Interceptor Classes

The code samples in this section are based on the Java authorization example, which is included in the Coherence examples that are delivered as part of the documentation library. 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.

Creating Authorization Interceptor Classes

An interceptor class can be created for both a proxied cache service and a proxied invocation service by implementing the `CacheService` and `InvocationService` interfaces, respectively. However, a set of wrapper classes are typically extended when implementing authorization: `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 4–3 is taken from the Coherence examples and demonstrates creating an authorization interceptor class for a proxied cache service by extending `WrapperCacheService`. It wraps all `CacheService` methods on the proxy and applies access controls based on the Subject passed from an extend client. The implementation only allows a Principal with the specified role to access the wrapped `CacheService`.

Example 4–3 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 this class requires a named cache implementation. For this example, the `WrapperNamedCache` class is extended and wraps each method of the `NamedCache` instance. This allows access controls to be applied to different cache operations. Example 4–4 is a code excerpt taken from the Coherence examples that demonstrates overriding the `NamedCache` methods and applying access checks before allowing the method to be executed. See the Coherence examples for the complete class.

Example 4–4 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 4–5 is taken from the Coherence examples and 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 only allows Principals with a specified role name to access the wrapped `InvocationService`.

Example 4–5 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, it then calls the `query()` method on the proxied `InvocationService` instance.

Enabling Authorization Interceptor Classes

Interceptor classes for a proxied cache service and a proxied invocation service are enabled in a proxy scheme definition within the `<cache-service-proxy>` element and `<invocation-service-proxy>` element, respectively. The `<class-name>` element is used to enter the fully qualified name of the interceptor class. Initialization parameters required by the class can be specified using the `<init-params>` element. See "cache-service-proxy" and "invocation-service-proxy" in *Oracle Coherence Developer's Guide* for detailed information on using these elements.

The following example demonstrates enabling both a proxied cache service and proxied invocation service interceptor class. The example uses the interceptor classes that were created in Example 4–3 and Example 4–5.

```
<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>
```

5

Using SSL to Secure Communication

Coherence provides a Secure Socket Layer (SSL) implementation that secures TCMP communication between cluster nodes and the TCP communication between Coherence*Extend clients and proxies. Coherence supports the Transport Layer Security (TLS) 1.0 protocol which is the next version of the SSL 3.0 protocol; however, the term SSL is used in this documentation since it is the more widely recognized term.

The following sections are included in this chapter:

Overview of SSL

This section provides a brief overview of common SSL concepts that are used in this documentation and is not intended to be a complete guide to SSL. Those new to SSL should refer to the formal specification maintained at `http://www.ietf.org` and the Java SE Security resources located at `http://java.sun.com/javase/technologies/security/index.jsp`. Those familiar with SSL can skip this section.

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/decrypting communication using unique keys that are associated with authenticated clients and servers.

Establishing Identity and Trust

An entity's identity is established 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

An entity's digital certificate contains a public encryption key that is paired with the entity's 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 an entity's public key can only be decrypted by the entity's 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 can be 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 both sides of the communication are known.

Generating Java SSL Artifacts

The Java `keytool` utility that is located in the *`JDK_HOME`*`/bin` directory is used to generate and manage SSL artifacts. This includes: creating a key store; 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 key store.

The following example creates a key store named `server.jks` that is located in the `/test` directory. A public and private key pair are 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 key store entry referred to by the alias (`admin`). Passwords must be entered for both the key store and private key.

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

The certificate generated by this 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
```

This CSR file must be sent to the CA, which returns a signed certificate. Use the `keytool` utility to import the returned certificate which replaces the self-signed certificate in the key store:

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

Lastly, the `keytool` utility is used to create a second key store that acts as a trust store. The trust store contains digital certificates of trusted CAs. Certificates that have been verified by a CA are only considered trusted if the CA's certificate is also found in the trust store. For example, in a typical one-way authentication scenario, a client must have a trust store 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 store.

Generating Windows SSL Artifacts

The following steps describe how to setup two-way authentication on Windows and is required when securing Coherence*Extend .NET clients. See "Configure a .NET Client-Side Stream Provider" on page 5-13. Refer to 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 setup 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. On each client machine, copy the `.cer` file on a location accessible for the `sslstream` client program.
8. On each client machine, copy the `.pfx` file.
9. On each client machine, import the `.pfx` file to the trusted root certification authority certificate store
10. On each client machine, delete the `.pfx` file (this step ensure the client can not communicate or export the private key).

Using SSL to Secure TCMP Communication

A Coherence cluster can be configured to use SSL with TCMP. Coherence supports both one-way and two-way authentication. 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. The implications of SSL on administration and performance should be carefully considered.

The following topics are include in this section:

- Defining a SSL Socket Provider

- Using the Pre-Defined SSL Socket Provider

Defining a SSL Socket Provider

SSL for TCMP is configured 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 a SSL socket provider configuration that is defined within a `<socket-providers>` node. The `<socket-provider>` element can also be used to define a full configuration within the `<unicast-listener>` element. Both approaches are demonstrated below. See *Oracle Coherence Developer's Guide* for a detailed reference of the `<socket-provider>` element.

Note: A cluster must be configured to use well known addresses to use SSL with TCMP. See *Oracle Coherence Developer's Guide* for details on setting up well known addresses.

Example 5–1 demonstrates a SSL two-way authentication setup and requires both an identity store and trust store 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 for completeness but may be left out when using the default values. The example uses the preferred approach where the SSL socket provider is defined within the `<socket-providers>` node and referred to from within the `<unicast-listener>` element.

Example 5–1 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>mySSLConfig</socket-provider>
         <well-known-addresses>
            <socket-address id="1">
               <address>198.168.1.5</address>
               <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>
            </ssl>
         </socket-provider>
      </socket-providers>
   </cluster-config>
</coherence>
```

As an alternative, the SSL socket provider can also be directly defined in the `<unicast-listener>` element as shown below:

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

<coherence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://xmlns.oracle.com/coherence/coherence-operational-config"
   xsi:schemaLocation="http://xmlns.oracle.com/coherence/
   coherence-operational-config coherence-operational-config.xsd">
   <cluster-config>
      <unicast-listener>
         <socket-provider>
            <ssl>
               <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>
         <well-known-addresses>
            <socket-address id="1">
               <address>198.168.1.5</address>
               <port>8088</port>
            </socket-address>
         </well-known-addresses>
      </unicast-listener>
   </cluster-config>
</coherence>
```

Using the Pre-Defined SSL Socket Provider

Out-of-box, a pre-defined SSL socket provider is included that allows for configuration of two-way SSL connections that is based on peer trust where every trusted peer resides within a single JKS key store. The proprietary peer trust algorithm (PeerX509) works by assuming trust (and only trust) of the certificates that are in the key store. The peer algorithm can increase the performance of SSL by leveraging the fact that TCMP is a peer-to-peer protocol.

The pre-defined SSL socket provider is located 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="tangosol.coherence.security.keystore">
                     file:keystore.jks
                  </url>
                  <password system-property="tangosol.coherence.security.
                     password"/>
               </key-store>
               <password system-property="tangosol.coherence.security.password"/>
            </identity-manager>
            <trust-manager>
               <algorithm>PeerX509</algorithm>
               <key-store>
                  <url system-property="tangosol.coherence.security.keystore">
                     file:keystore.jks
                  </url>
                  <password system-property="tangosol.coherence.security.
                     password"/>
               </key-store>
            </trust-manager>
         </ssl>
      </socket-provider>
   </socket-providers>
</cluster-config>
...
```

As configured, the pre-defined SSL socket provider requires a Java key store named `keystore.jks` that is found on the classpath. This name can be overridden using the `tangosol.coherence.security.keystore` system property to specify a different key store. In addition, the `tangosol.coherence.security.password` system property can specify the required password for the key store and certificate. As an alternative, an operational override file may be used to modify the pre-defined SSL socket provider definition as required.

Note: Ensure that certificates for all nodes in the cluster have been imported into the key store.

To use the pre-defined SSL socket provider, override the `<socket-provider>` element in the `<unicast-listener>` configuration and reference the SSL socket provider using it's `id` attribute. The following example configures a unicast listener to use the pre-defined 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>ssl</socket-provider>
         <well-known-addresses>
            <socket-address id="1">
               <address>198.168.1.5</address>
               <port>8088</port>
            </socket-address>
         </well-known-addresses>
      </unicast-listener>
   </cluster-config>
</coherence>
```

Using SSL to Secure Extend Client Communication

Communication between extend clients and extend proxies can be secured using SSL. 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.

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 store must include the signing CA's digital certificate. Self-Signed certificates may be used during development as needed.

The following topics are included in this section:

- Configuring a Cluster-Side SSL Socket Provider
- Configure a Java Client-Side SSL Socket Provider
- Configure a .NET Client-Side Stream Provider

Configuring a Cluster-Side SSL Socket Provider

SSL is configured in the cluster-side cache configuration file by defining a SSL socket provider for a proxy service. There are two options for configuring a SSL socket provider depending on the level of granularity that is required:

- Per Proxy Service – Each proxy service defines a SSL socket provider configuration or references a pre-defined configuration that is included in the operational configuration file.
- All Proxy Services – All proxy services use the same global SSL socket provider configuration. Proxy services that provide their own configuration override the global configuration. The global configuration can also reference a predefined configuration that is included in the operational configuration file.

Configure a SSL Socket Provider Per Proxy Service

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

See "socket-provider" in *Oracle Coherence Developer's Guide* for a detailed reference of the `<socket-provider>` element.

Example 5–2 demonstrates a proxy scheme that configures a 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 5–2 configures both an identity key store (`server.jks`) and a trust key store (`trust.jks`). This is typical of two-way SSL authentication where both the client and proxy must exchange their digital certificate and confirm each other's identity. For one-way SSL authentication, the proxy server configuration must include an identity key store but need not include a trust key store.

Example 5–2 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>
            </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 a SSL socket provider configuration that is defined in the `<socket-providers>` node of the operational deployment descriptor by specifying the configuration's `id` attribute (`ssl`). See "socket-providers" in *Oracle Coherence Developer's Guide* for a detailed reference of the `<socket-providers>` element.

> **Note:** Out-of-box, a pre-defined SSL socket provider is included in the operational deployment descriptor and is named `ssl`. The pre-defined SSL socket provider is configured for two-way SSL connections and is based on peer trust where every trusted peer resides within a single JKS key store. See "Using the Pre-Defined SSL Socket Provider" on page 5-6 for details. To configure a different SSL socket provider, use an operational override file to modify the pre-defined 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 a 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 "defaults" in *Oracle Coherence Developer's Guide* for a detailed reference of the `<default>` element.

The following example uses the same SSL socket provider configuration from Example 5–2 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>
          </ssl>
       </socket-provider>
   </defaults>
   ...
```

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

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

Configure a Java Client-Side SSL Socket Provider

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

- Per Remote Service – Each remote service defines a SSL socket provider configuration or references a pre-defined configuration that is included in the operational configuration file.
- All Remote Services – All remote services use the same global SSL socket provider configuration. Remote services that provide their own configuration override the global configuration. The global configuration can also reference a predefined configuration that is included in the operational configuration file.

Configure a SSL Socket Provider Per Remote Service

To configure a SSL socket provider for a remote service, add a `<socket-provider>` element within the `<tcp-initiator>` element of a remote scheme definition. See "socket-provider" in *Oracle Coherence Developer's Guide* for a detailed reference of the `<socket-provider>` element.

Example 5–3 demonstrates a remote cache scheme that configures a socket provider that uses SSL. The example 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 5–3 configures both an identity key store (`server.jks`) and a trust key store (`trust.jks`). This is typical of two-way SSL authentication where both the client and proxy must exchange their digital certificate and confirm each other's identity. For one-way SSL authentication, the client configuration must include a trust key store but need not include an identity key store, which indicates the client does not exchange its digital certificate to the proxy and remains anonymous. The client's trust key store must includes the CA's digital certificate that was used to sign the proxy's digital certificate.

Example 5–3 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>
                  </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 a SSL socket provider configuration that is defined in the `<socket-providers>` node of the operational deployment descriptor by specifying the configuration's `id` attribute (`ssl`). See "socket-providers" in *Oracle*

Coherence Developer's Guide for a detailed reference of the `<socket-providers>` element.

> **Note:** Out-of-box, a pre-defined SSL socket provider is included in the operational deployment descriptor and is named `ssl`. The pre-defined SSL socket provider is configured for two-way SSL connections and is based on peer trust where every trusted peer resides within a single JKS key store. See for "Using the Pre-Defined SSL Socket Provider" on page 5-6 for details. To configure a different SSL socket provider, use an operational override file to modify the pre-defined 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 a 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 "defaults" in *Oracle Coherence Developer's Guide* for a detailed reference of the `<default>` element.

The following example uses the same SSL socket provider configuration from Example 5–3 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>
         </ssl>
      </socket-provider>
   </defaults>
   ...
```

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

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

Configure a .NET Client-Side Stream Provider

SSL is configured in the .NET client-side cache configuration file by defining an SSL stream provider for remote services. The SSL stream provider is 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 extend proxy's certificate is included in the Certificate Manager and that the CA's certificate that was used to sign the proxy's certificate is included as a trusted certificate authority. See "Generating Windows SSL Artifacts" on page 5-2 for a generic example. For more information on managing certificates, see

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

---

Example 5–4 demonstrates a remote cache scheme that configures a SSL stream provider. Refer to the cache configuration XML schema (*INSTALL_DIR*\config\cache-config.xsd) for details on the elements used to configure a SSL stream provider.

Example 5–4 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>
```