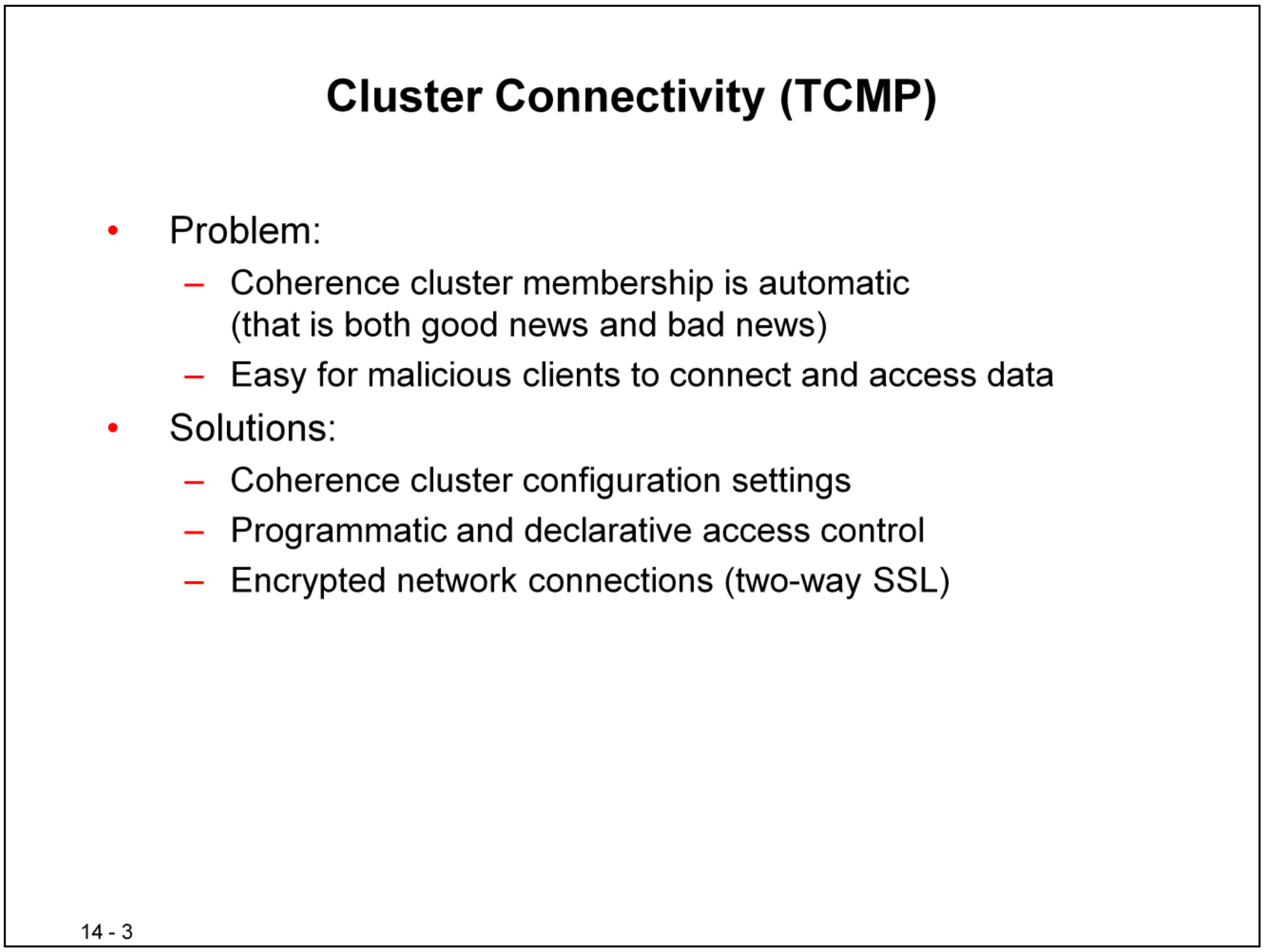
Understanding Coherence Security

Objectives

After completing this lesson, you should be able to describe the security options available with Coherence.

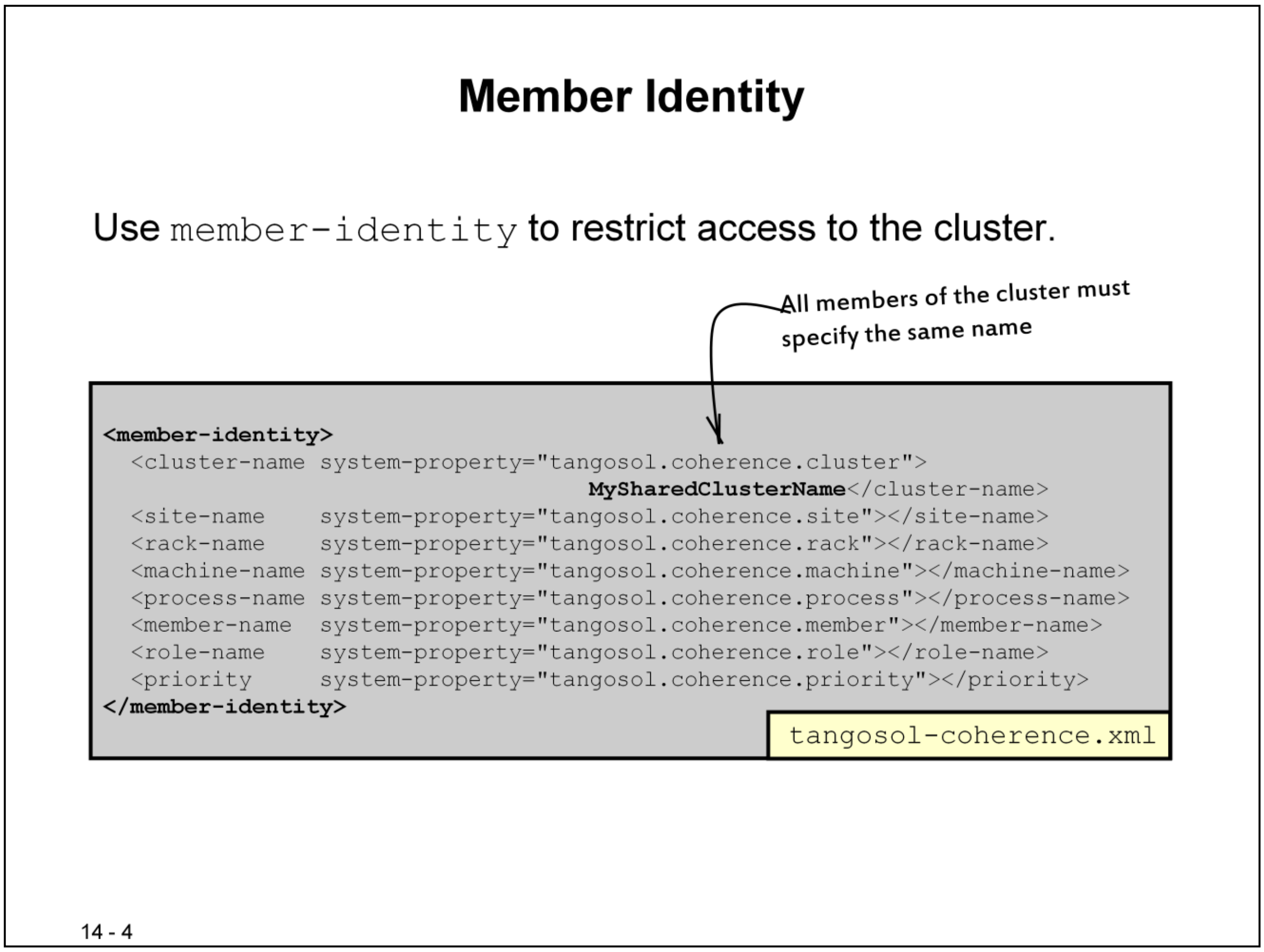


Cluster Connectivity (TCMP)

- Problem:
  - Coherence cluster membership is automatic (that is both good news and bad news)
  - Easy for malicious clients to connect and access data
- Solutions:
  - Coherence cluster configuration settings
  - Programmatic and declarative access control
  - Encrypted network connections (two-way SSL)

14 - 3

The default Coherence client connects to any running Coherence cluster automatically. At first, this appears to make Coherence an easy target for unauthorized users to exploit to access data that is stored in the cache. Coherence provides a number of security mechanisms that ensure that unauthorized users cannot access the cluster itself, or the data that is stored in the cache. This lesson covers the configuration settings, security frameworks, and network encryption capabilities offered by Coherence.



Member Identity

Use `member-identity` to restrict access to the cluster.

All members of the cluster must specify the same name

```
<member-identity>
  <cluster-name system-property="tangosol.coherence.cluster">
                                        MySharedClusterName</cluster-name>
  <site-name    system-property="tangosol.coherence.site"></site-name>
  <rack-name    system-property="tangosol.coherence.rack"></rack-name>
  <machine-name system-property="tangosol.coherence.machine"></machine-name>
  <process-name system-property="tangosol.coherence.process"></process-name>
  <member-name  system-property="tangosol.coherence.member"></member-name>
  <role-name    system-property="tangosol.coherence.role"></role-name>
  <priority     system-property="tangosol.coherence.priority"></priority>
</member-identity>
```

`tangosol-coherence.xml`

14 - 4

Coherence cache, proxy, management, and client processes all become “joined” with the cluster at startup. Coherence has the ability to autodiscover other cluster members. Due to this nature of Coherence, it is very important to ensure that rogue Coherence processes are not able to join the production cluster. Coherence provides features that control which processes can join the cluster, such as Member Identity. The `tangosol-coherence.xml` file has a `<member-identity>` element in the cluster configuration file. When set, only cache services running with the same `member-identity` set can join the cluster.

Authorized Hosts

Use `authorized-hosts` to restrict which hosts can join the cluster.

```
<authorized-hosts>
  <host-address>hostname1</host-address>
  <host-address>hostname2</host-address>
  <host-range>
    <from-address>10.1.1.1</from-address>
    <to-address>10.1.1.100</to-address>
  </host-range>
</authorized-hosts>
```

Only these hosts can connect to the cluster.
There can be multiple host-range entries

`tangosol-coherence.xml`

Coherence also provides a feature to allow only certain hosts to access the cluster. The `tangosol-coherence.xml` file has an `<authorized-hosts>` element in the cluster configuration. When set, only cache services running on these hosts can join the cluster.

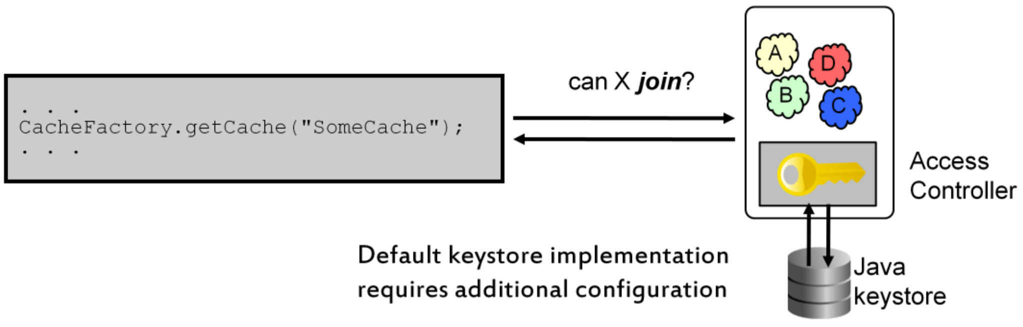
Access Control

Access control

- Accepts or denies requests to obtain as cache
- Is disabled by default, but can be enabled using
  `<security-config/>` or
  `-Dtangosol.coherence.security...`

Often requires additional configuration

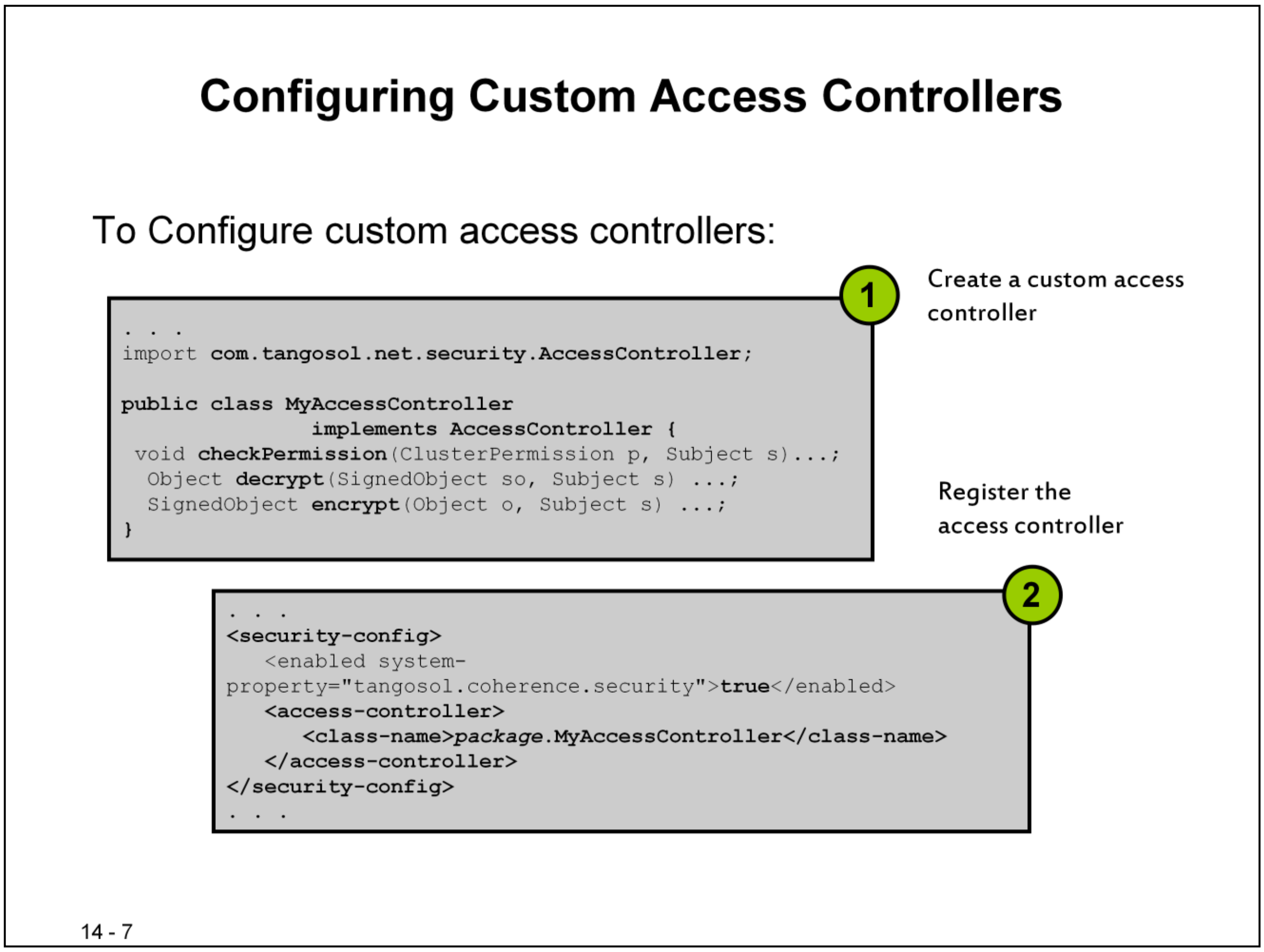
- Default implementation uses keystore to validate requests



```
. . .
CacheFactory.getCache("SomeCache");
. . .
```

can X *join*?
A
D
B
C
Access Controller
Java keystore
Default keystore implementation requires additional configuration

Access control is the process of limiting `getCache()` or `ensureCache()` operations. Coherence provides a default access controller which uses a keystore to perform authentication. The default implementation uses `keytool`, or a similar tool, to create a keystore which is referenced using `-Dtangosol.coherence.secutity.keystore={path to keystore file}` and confirms subjects and principles. The default implementation also requires a login module, refer to the Coherence 12*c* security documentation and/or lesson 11 for more information on login modules.



Configuring Custom Access Controllers

To Configure custom access controllers:

1 Create a custom access controller

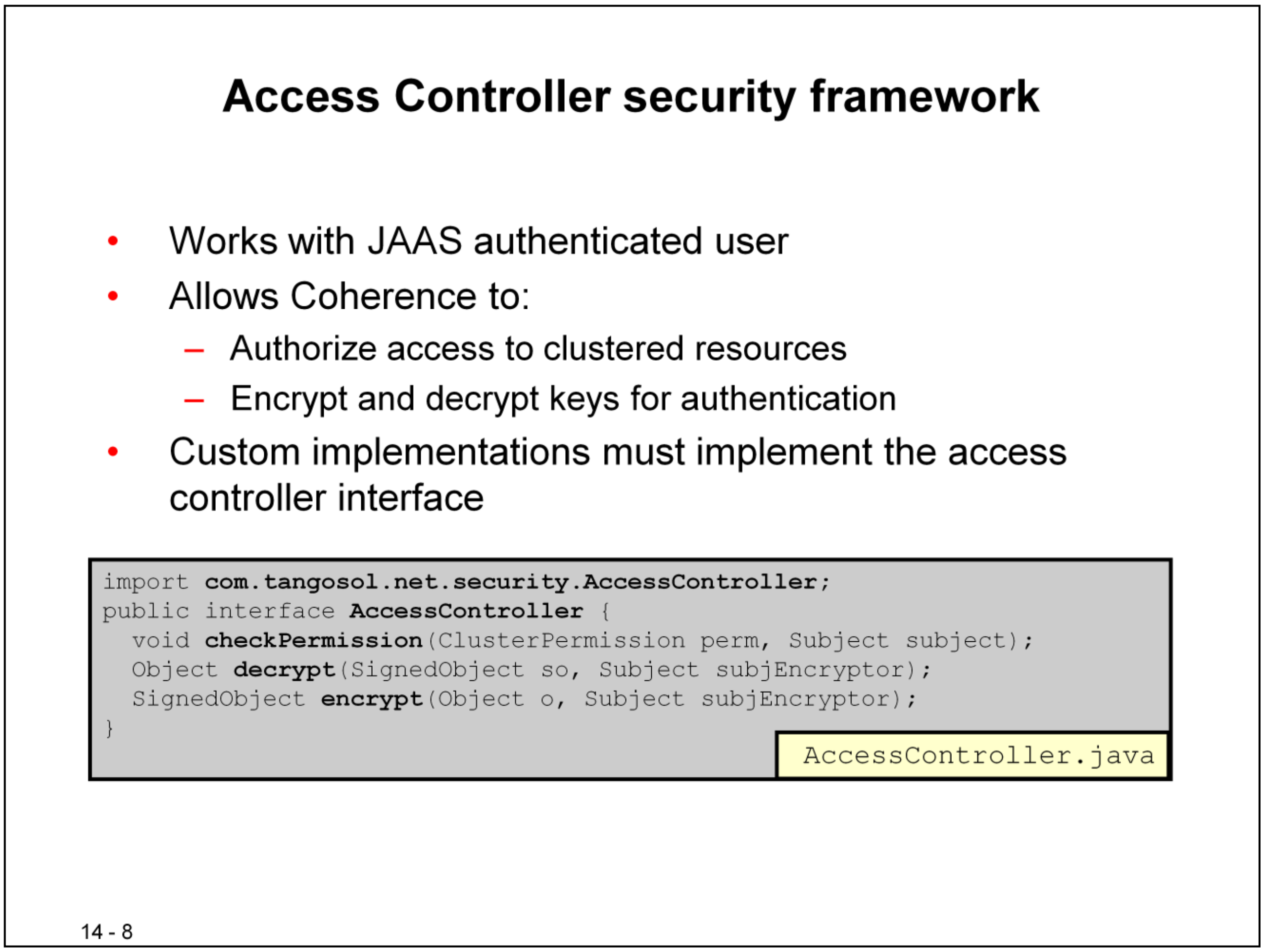
```
. . .
import com.tangosol.net.security.AccessController;

public class MyAccessController
               implements AccessController {
 void checkPermission(ClusterPermission p, Subject s)...;
  Object decrypt(SignedObject so, Subject s) ...;
  SignedObject encrypt(Object o, Subject s) ...;
}
```

2 Register the access controller

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

14 - 7

Configuring an access controller is a two step process. First you must provide a implementation of a JAAS login module. Coherence provides a basic implementation of a access controller. Covered in more detail in later slides. Second you must configure Coherence to use the provided implementation by specifying the implementation using a Coherence operational override which includes an element which references the access controller class.



Access Controller security framework

- Works with JAAS authenticated user
- Allows Coherence to:
  - Authorize access to clustered resources
  - Encrypt and decrypt keys for authentication
- Custom implementations must implement the access controller interface

```
import com.tangosol.net.security.AccessController;
public interface AccessController {
  void checkPermission(ClusterPermission perm, Subject subject);
  Object decrypt(SignedObject so, Subject subjEncryptor);
  SignedObject encrypt(Object o, Subject subjEncryptor);
}
```

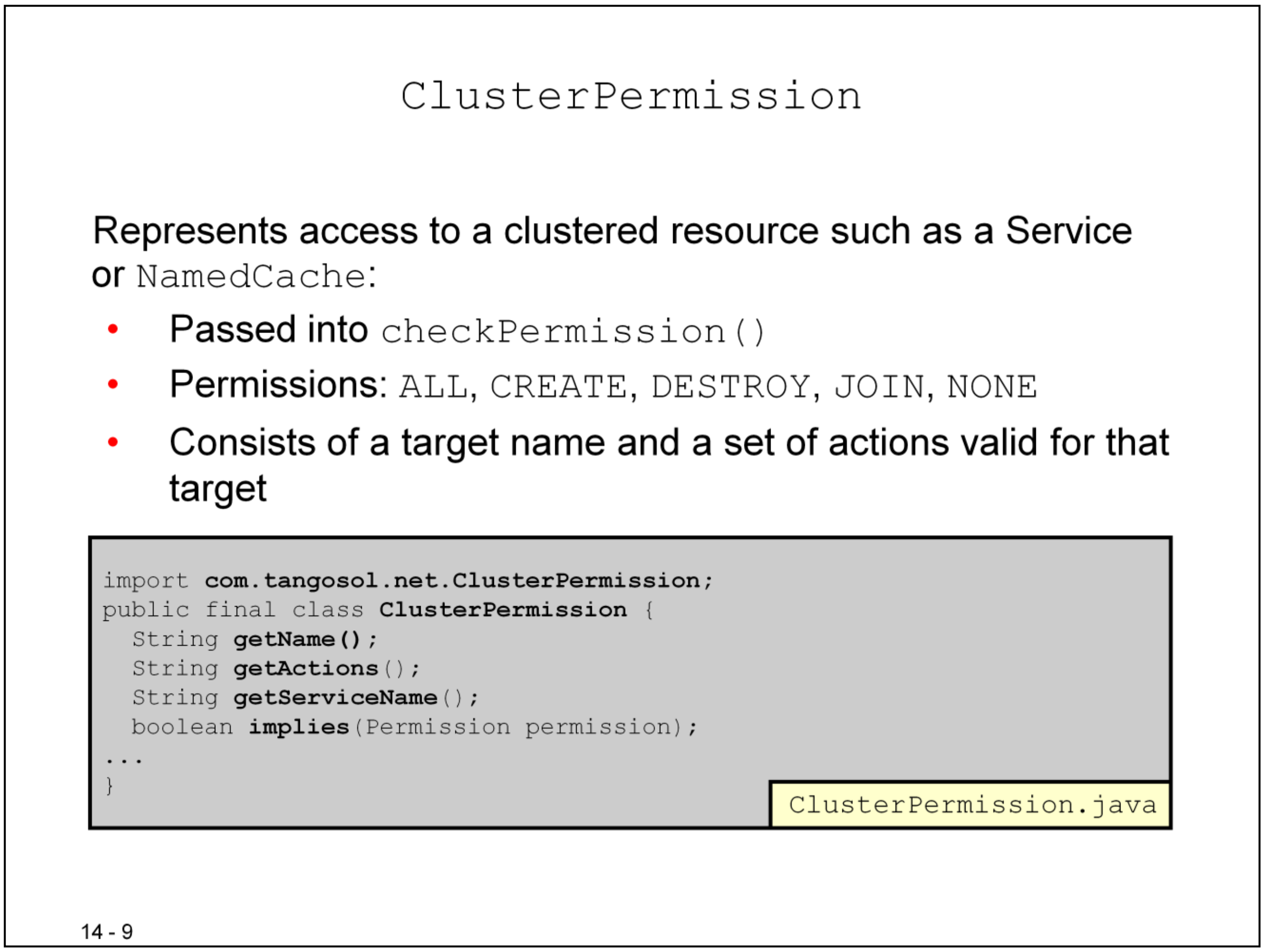
AccessController.java

Coherence provides an Access Controller framework that allows you to implement your own security mechanisms. Coherence provides a `DefaultController` implementation that uses a Java keystore-based `LoginModule` and `permission.xml` file to provide authentication and authorization services. Other `LoginModule` types may be implemented to integrate with existing authentication and authorization systems, such as an LDAP `LoginModule`, and Oracle Entitlements Server (OES), a centralized fine-grained entitlement engine.

The Access Controller framework provides hooks for authentication and authorization. This includes methods for encrypting and decrypting identity keys for authentication purposes. For example, if the public key is used to successfully decrypt the identity, then it must have been encrypted with the associated private key.

Authorization is provided for operations on clustered resources, such as:

- Creating a new clustered cache or service
- Joining an existing clustered cache or service
- Destroying an existing clustered cache



ClusterPermission

Represents access to a clustered resource such as a Service or NamedCache:

- Passed into checkPermission()
- Permissions: ALL, CREATE, DESTROY, JOIN, NONE
- Consists of a target name and a set of actions valid for that target

```
import com.tangosol.net.ClusterPermission;
public final class ClusterPermission {
  String getName();
  String getActions();
  String getServiceName();
  boolean implies(Permission permission);
...
}
```

ClusterPermission.java

14 - 9

The AccessController's checkPermission() method uses a ClusterPermission class as the first parameter to pass the criteria used to make the authorization decision for the request. The ClusterPermission class specifies which resource is being requested (the target), and the privileged actions that are valid for that resource.

- getName(): Contains the name of the cache or service that needs to be protected
- getActions(): Returns the actions (privileges) as a String in a canonical form. This represents the actions the current request is trying to attempt.
- getServiceName(): Returns the service name for this permission object or null if the permission applies to any service
- implies(): Checks if the specified permission's actions are "implied by" this object's actions

Registering an Access Controller

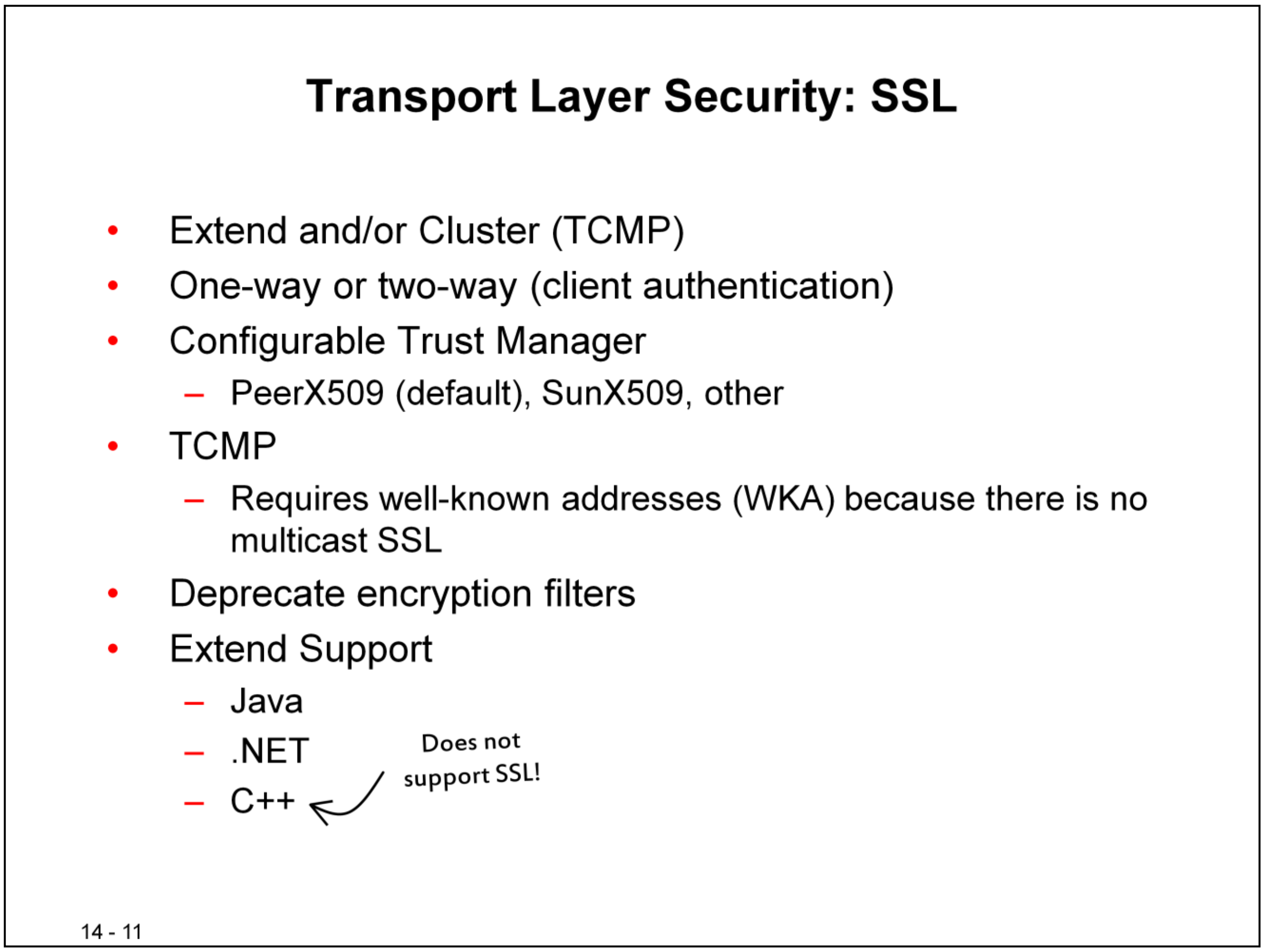
- Custom Access Controllers
- Are registered using a `security-config` element
- Within an operational configuration
- Specifying the fully qualified path to a class which implements `com.tangosol.net.security.AccessController`

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

tangosol-coherence.xml

14 - 10

See the Oracle Coherence Operational Configuration element documentation for a complete list of all the sub elements of `security-config` and their use.



Transport Layer Security: SSL

- Extend and/or Cluster (TCMP)
- One-way or two-way (client authentication)
- Configurable Trust Manager
  - PeerX509 (default), SunX509, other
- TCMP
  - Requires well-known addresses (WKA) because there is no multicast SSL
- Deprecate encryption filters
- Extend Support
  - Java
  - .NET
  - C++

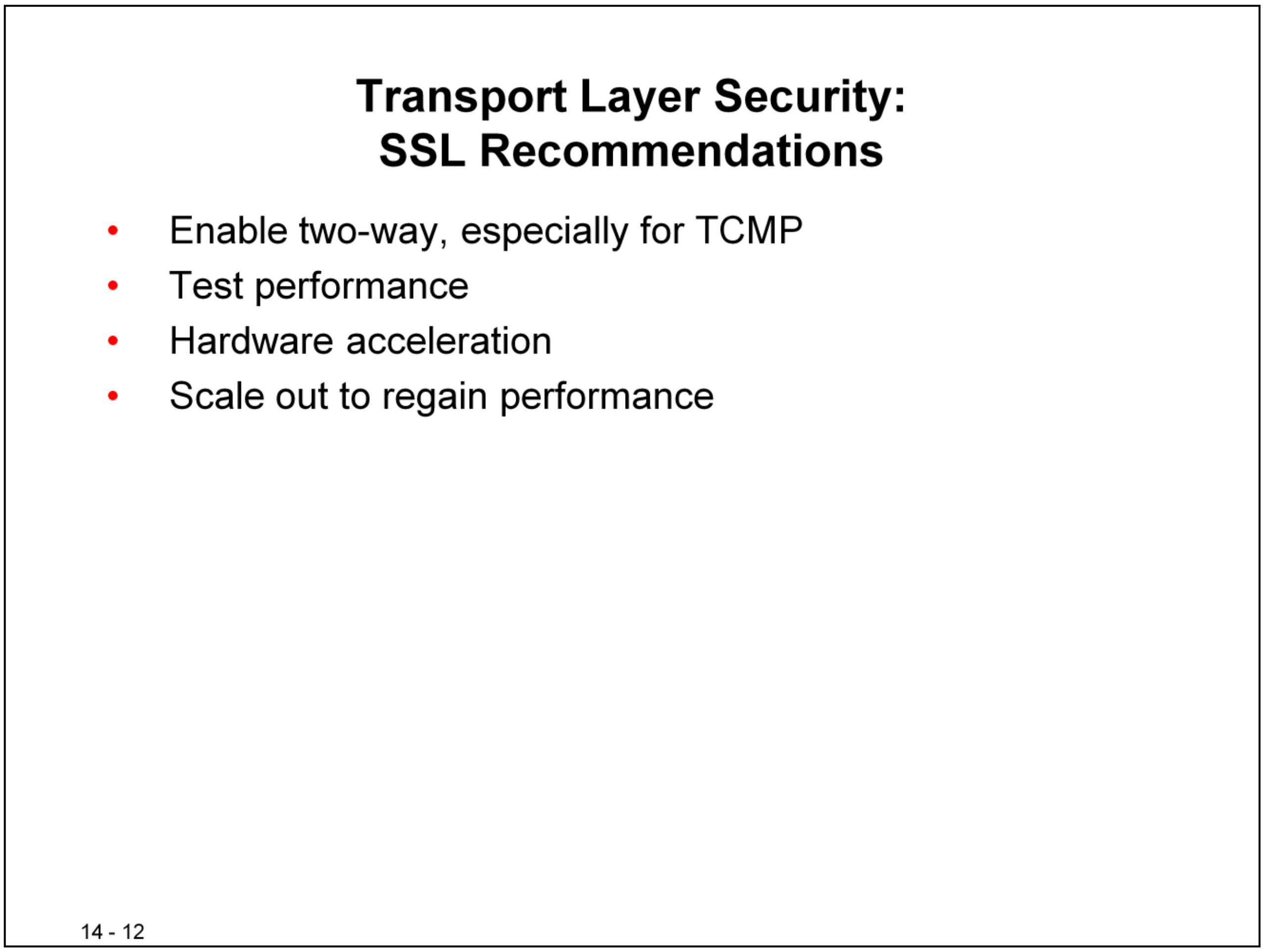
Does not support SSL!

14 - 11

Coherence provides SSL transport layer security for TCP/IP protocols for Coherence clustered nodes and Coherence*Extend nodes. Currently, there are two included trust managers: PeerX509 which is the default selection, and SunX509. The Coherence framework allows for the integration of any trust manager. When using SSL for clustered (TCMP) nodes, the use of well-known addresses (WKA) TCP/IP connections is required because there is no SSL support for the multicast networking protocol.

The encryption filters that were previously part of the Coherence product are deprecated as of the 3.6 release, and replaced by the SSL feature.

SSL support is available for Coherence*Extend nodes, unfortunately C++ is not currently supported.

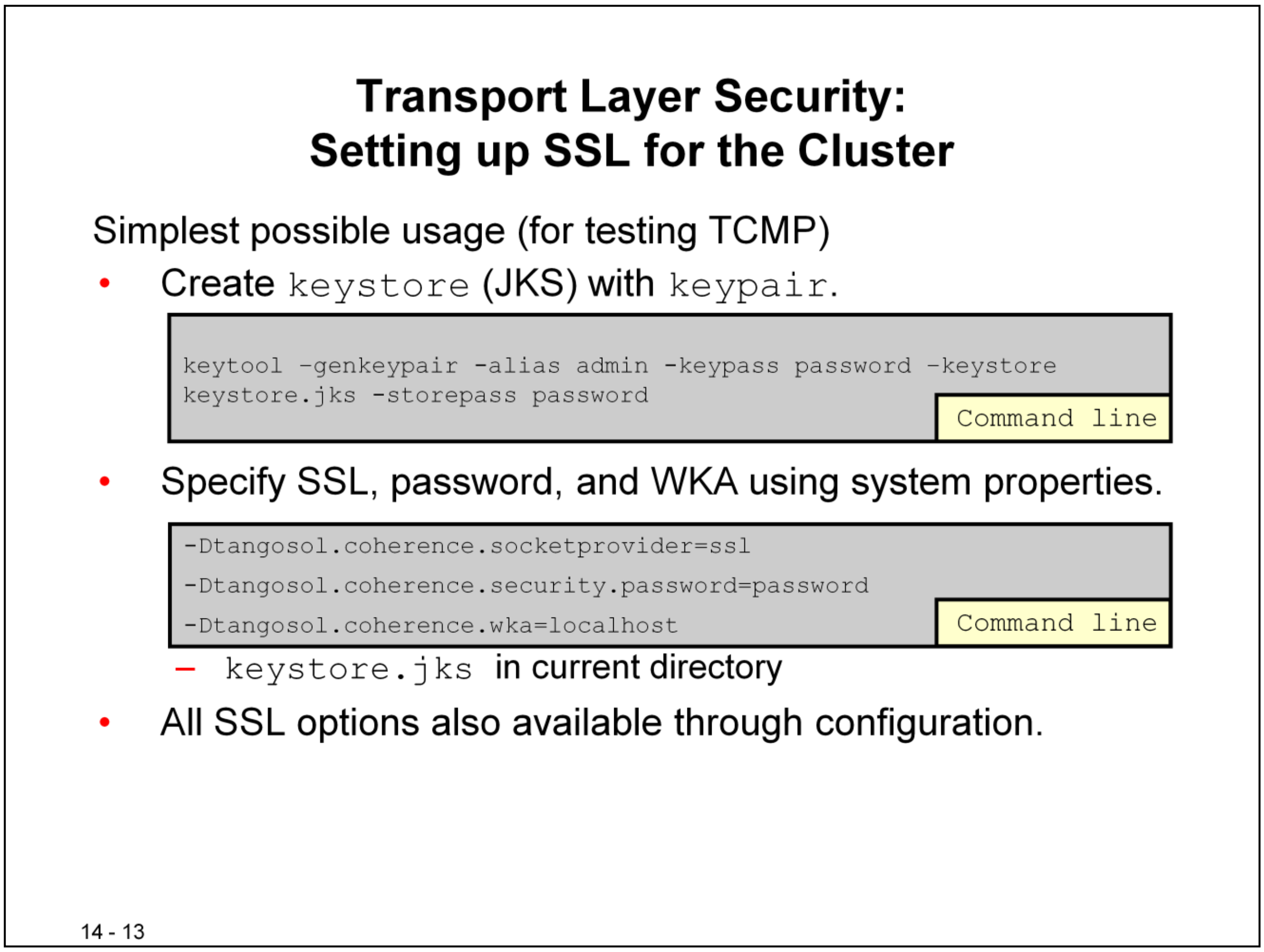


Transport Layer Security: SSL Recommendations

- Enable two-way, especially for TCMP
- Test performance
- Hardware acceleration
- Scale out to regain performance

14 - 12

Oracle recommends using two-way SSL, especially for TCMP clustered nodes, because there is no notion of a client in the cluster, and it does not make sense for one clustered node to provide a trust certificate to another node so it can be trusted, while not receiving some form of trusted artifact in return from the other cluster member. Naturally, with any SSL or encryption-intensive processing, the application should be tested to see if it performs well, and standard performance improving techniques should be employed when needed, such as the use of hardware encryption/decryption accelerators, which can nearly eliminate processing latency, or scaling out the cluster to provide more CPU processing power to handle the load.



Transport Layer Security: Setting up SSL for the Cluster

Simplest possible usage (for testing TCMP)

- Create `keystore` (JKS) with `keypair`.

```
keytool -genkeypair -alias admin -keypass password -keystore
keystore.jks -storepass password
```

Command line

- Specify SSL, password, and WKA using system properties.

```
-Dtangosol.coherence.socketprovider=ssl
-Dtangosol.coherence.security.password=password
-Dtangosol.coherence.wka=localhost
```

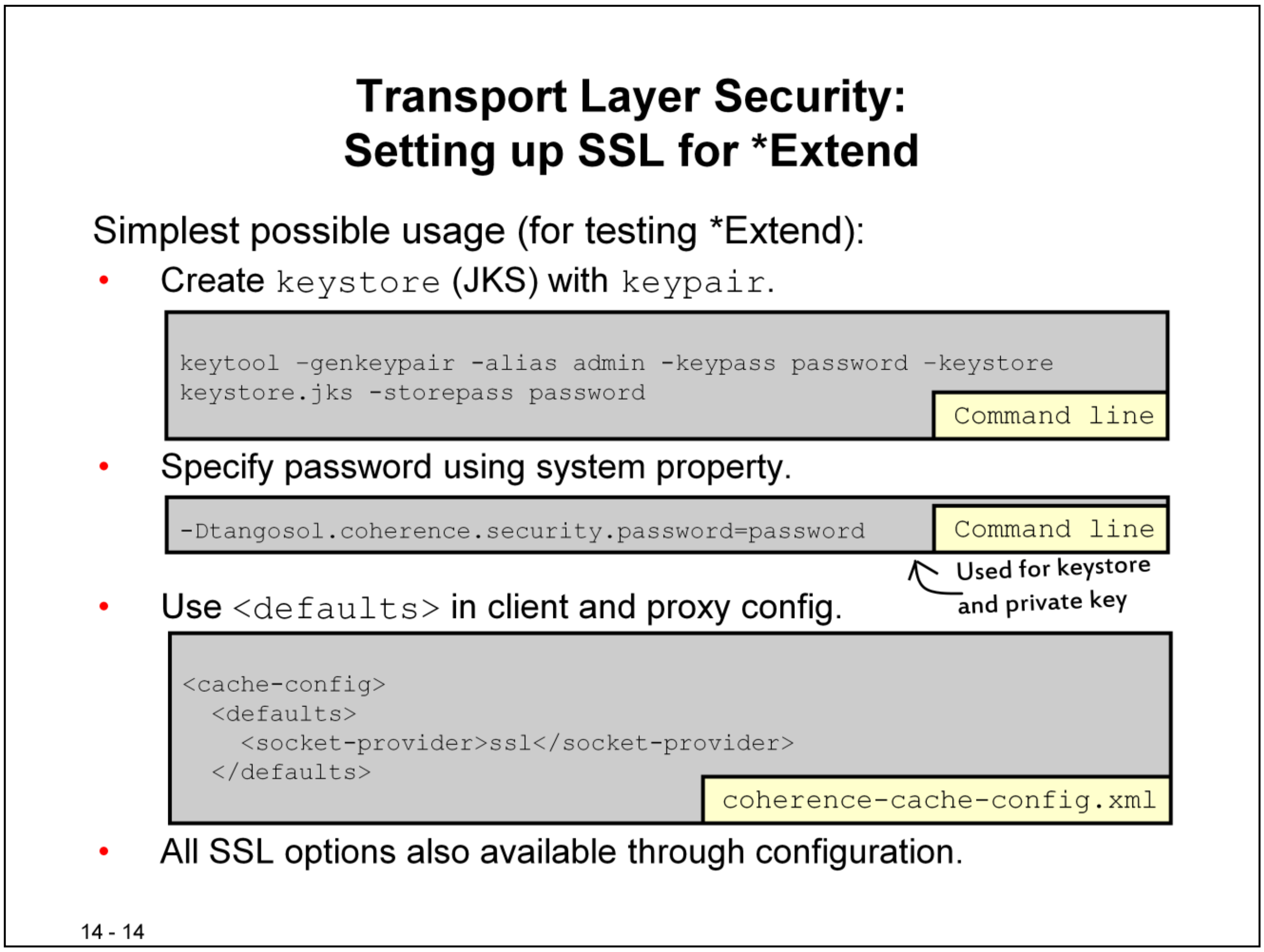
Command line

  - `keystore.jks` in current directory
- All SSL options also available through configuration.

14 - 13

This slide shows the simplest way to set up SSL for clustered nodes:

- A Java Key Store (JKS) is created or used to store a certificate public and private keypair.
- SSL, password, and WKA settings are configured using system properties from the command line. Note in this case that the password is then in clear text, and is used for both the keystore password and the private key password. The `keystore.jks` file, which holds the keypair, must be in the current directory where the JVM is executed. The alias does not need to be specified because internally, the keystore API being used by Coherence assumes that only a single private key exists in the keystore, but perhaps exists in multiple public certs.
- All the SSL settings are also available through the operational `tangosol-coherence.xml` configuration override file.



Transport Layer Security: Setting up SSL for *Extend

Simplest possible usage (for testing *Extend):

- Create `keystore` (JKS) with `keypair`.

```
keytool -genkeypair -alias admin -keypass password -keystore
keystore.jks -storepass password
```

Command line

- Specify password using system property.

```
-Dtangosol.coherence.security.password=password
```

Command line

Used for keystore and private key

- Use `<defaults>` in client and proxy config.

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

coherence-cache-config.xml

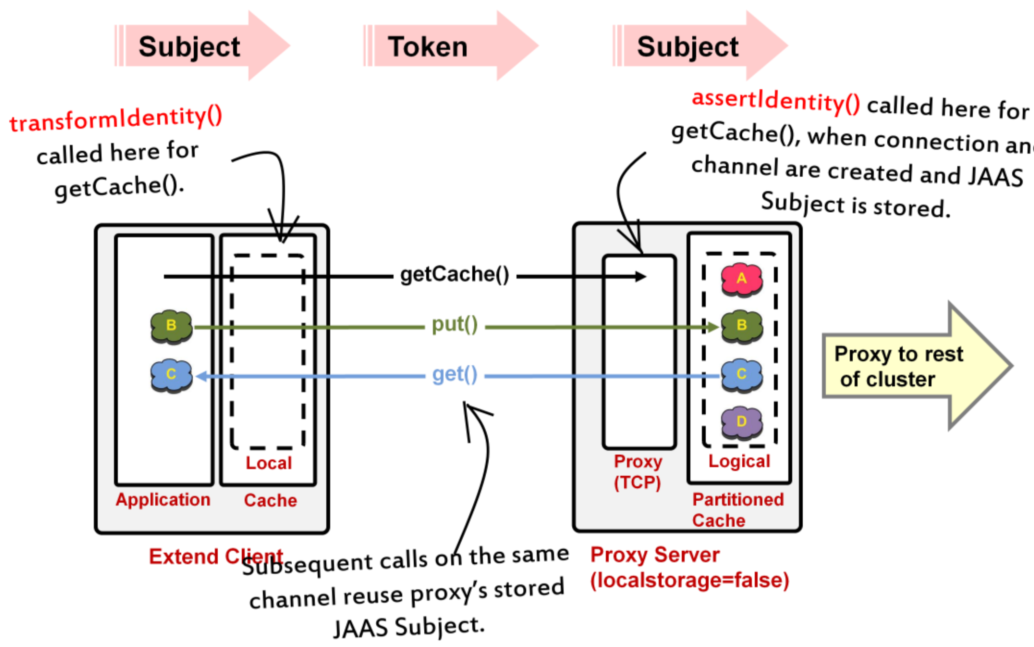
- All SSL options also available through configuration.

14 - 14

This slide shows the simplest way to set up SSL for Coherence*Extend nodes:

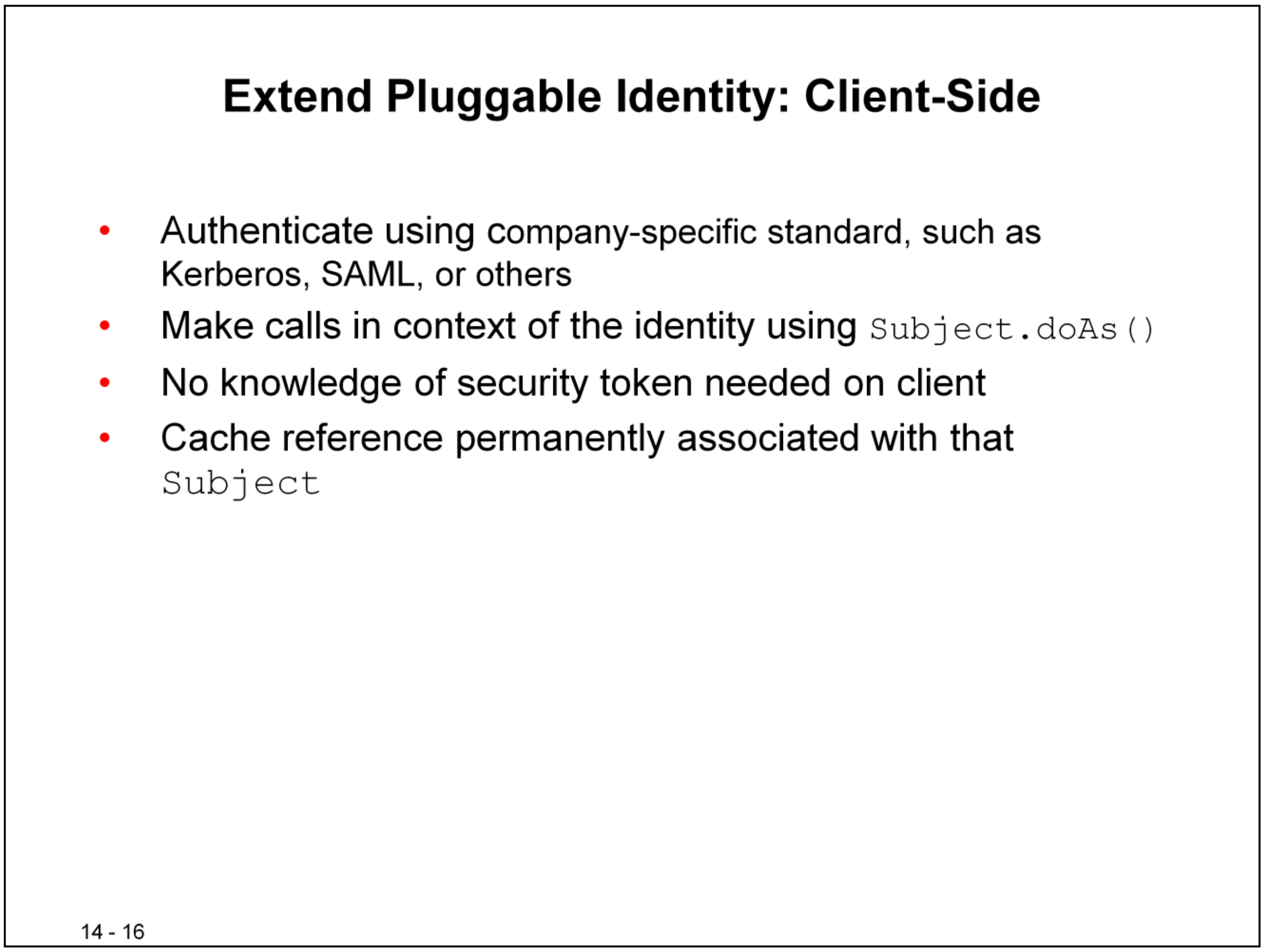
- A Java Key Store (JKS) keystore is created or used to store a certificate public and private keypair.
- Configure the password using the system property from the command line. Note in this case that the password is then in clear text, and is used for both the keystore password and the private key password.
- The `<defaults>` element in the client cache configuration file and the `<proxy-scheme>` configuration in the `coherence-cache-config.xml` file are used to specify SSL as the transport protocol.
- All the SSL settings are also available through the operational `tangosol-coherence.xml` configuration override file.

Extend Pluggable Identity: Architecture



Subject
Token
Subject
transformIdentity() called here for getCache().
assertIdentity() called here for getCache(), when connection and channel are created and JAAS Subject is stored.
getCache()
put()
get()
B
C
A
B
C
D
Local
Application
Cache
Proxy (TCP)
Logical
Partitioned Cache
Proxy to rest of cluster
Extend Client
Subsequent calls on the same channel reuse proxy's stored JAAS Subject.
Proxy Server (localstorage=false)

Extend Pluggable Identity answers the question, "How do I get an identity from the Extend client to the Coherence proxy server?" Coherence*Extend provides the ability to convert a JAAS `Subject` to a corporate standard security token type, such as SAML, or a custom token type, to support single-sign-on and authorization checks for clients connecting to a Coherence proxy server. The client side calls a `transformIdentity()` hook that can be custom implemented, which is responsible for converting a JAAS `Subject` into the token used for identity assertion on the proxy side. The proxy server calls the `assertIdentity()` hook when a new user is connecting to the server, and is responsible for converting the token into a usable JAAS `Subject`, which is stored and associated with the connection, and multiplexed channel for that user. Subsequent calls on a `NamedCache` or Service that are requested on that channel, for that user, reuse the stored `Subject` to avoid the expensive operation of asserting the identity repeatedly. There is no notion of a "session" in Coherence. However, when the `CacheService`'s `releaseCache()` method is called, the associated channel is closed and any subsequent requests by that subject will trigger the identity assertion process again. It is up to the `identityAsserter()` code to check if the subject's token is still valid.

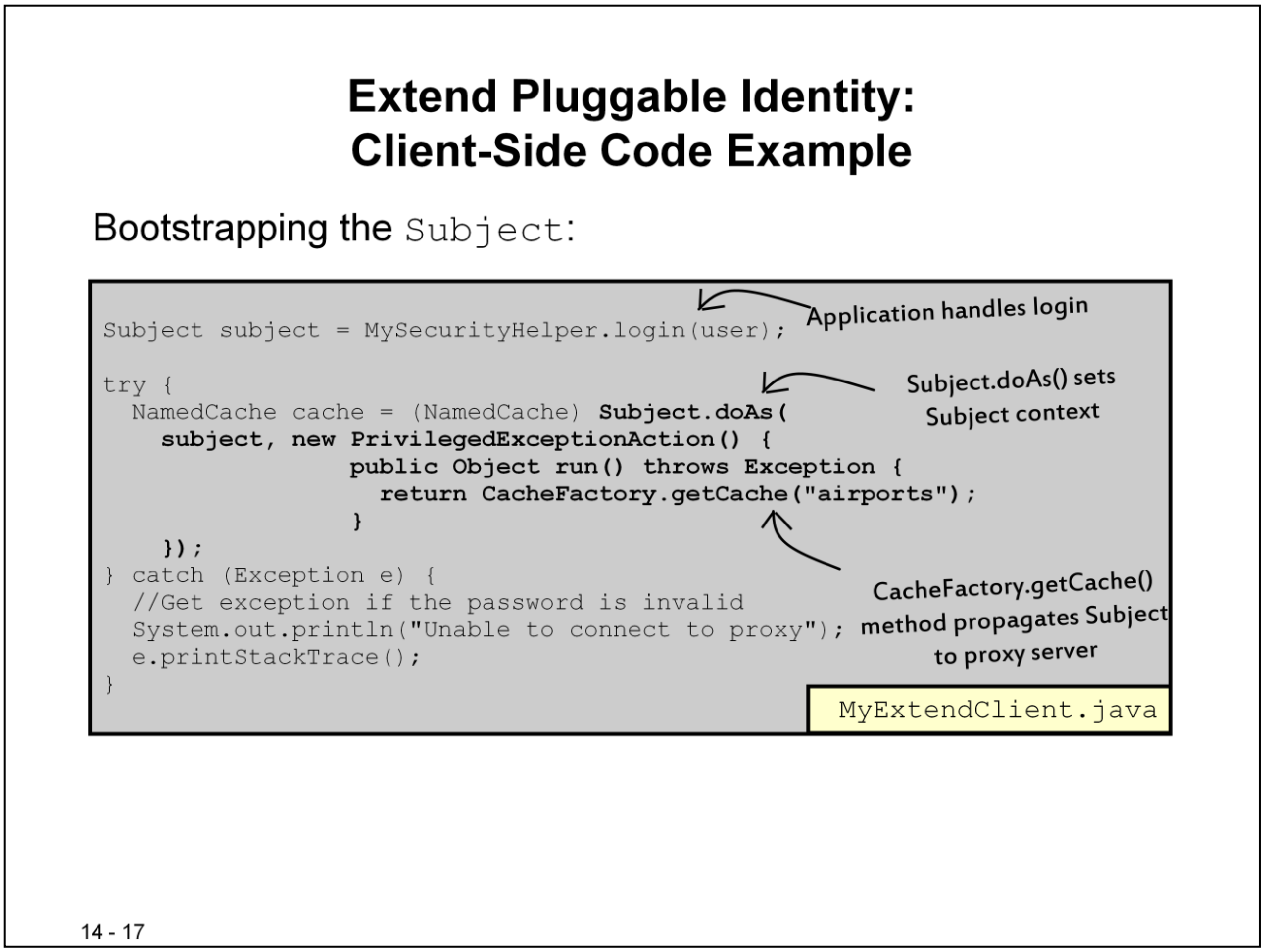


Extend Pluggable Identity: Client-Side

- Authenticate using company-specific standard, such as Kerberos, SAML, or others
- Make calls in context of the identity using `Subject.doAs()`
- No knowledge of security token needed on client
- Cache reference permanently associated with that `Subject`

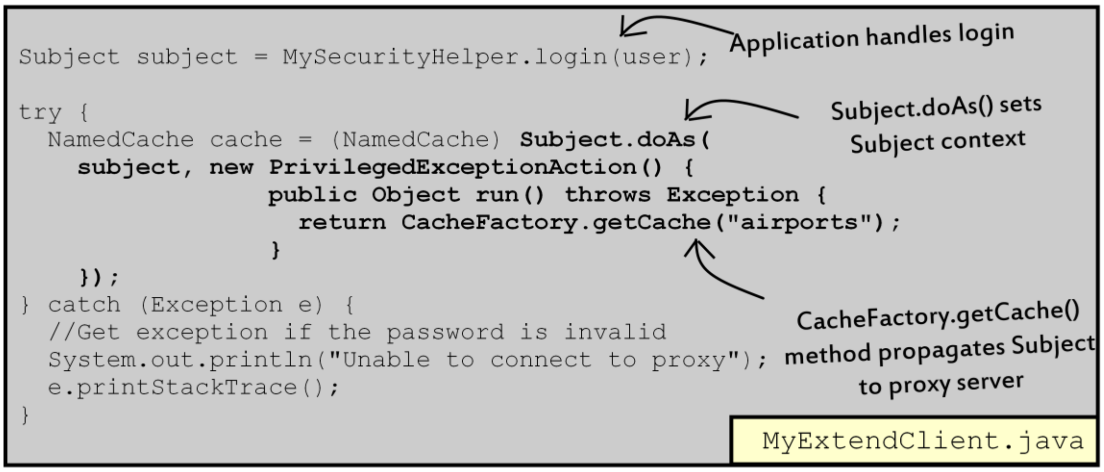
14 - 16

Authentication occurs either within an application that has embedded Coherence*Extend as part of its architecture to work with a Coherence clustered cache, or is a stand-alone Coherence*Extend client. In either case, the authentication mechanism used by the client application is entirely up to the application. This could include standard JAAS authentication provided by Coherence, a Kerberos solution, a SAML solution, or some other solution. The end result is that whatever mechanism is used, it must be represented as a JAAS `Subject` in order to perform a `Subject.doAs()` operation that wraps a call to either `CacheFactory.getCache()` or `getService()`, which will get the `Subject` associated with a channel for all user requests for that cache or service.



Extend Pluggable Identity: Client-Side Code Example

Bootstrapping the `Subject`:



```
Subject subject = MySecurityHelper.login(user);

try {
  NamedCache cache = (NamedCache) Subject.doAs(
    subject, new PrivilegedExceptionAction() {
                 public Object run() throws Exception {
                   return CacheFactory.getCache("airports");
                 }
    });
} catch (Exception e) {
  //Get exception if the password is invalid
  System.out.println("Unable to connect to proxy");
  e.printStackTrace();
}
```

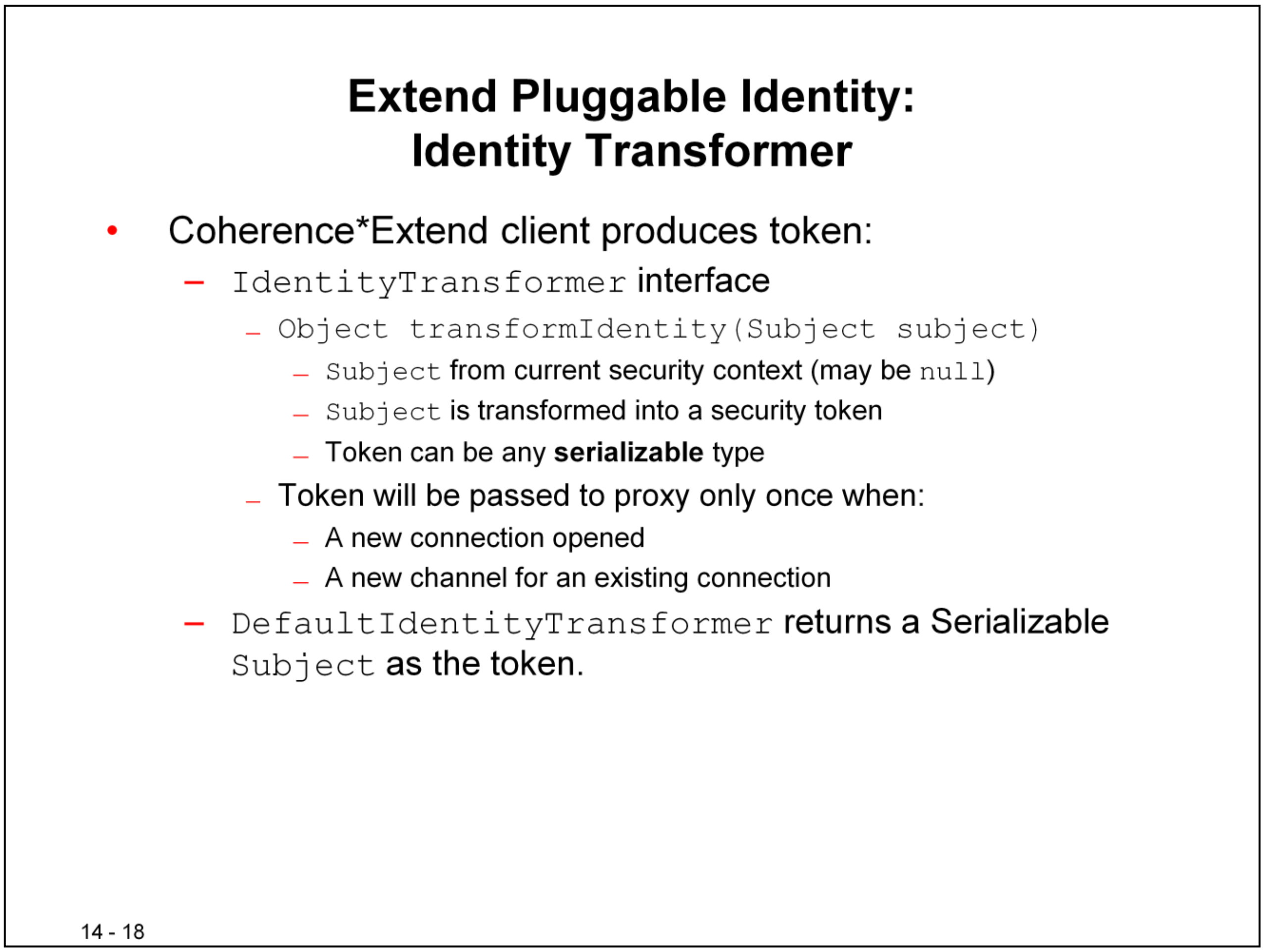
Application handles login
Subject.doAs() sets Subject context
CacheFactory.getCache() method propagates Subject to proxy server
MyExtendClient.java

14 - 17

Here is a code example of a client invoking `Subject.doAs()` with a `PriviledgedExceptionAction` that calls `CacheFactory.getCache()`. This initiates communication with the proxy server over TCP/IP. Here is what happens behind the code:

- If this is the first connection, the connection is established, then a channel is created.
- The Extend client invokes the `transformIdentity()` method to convert the `Subject` into whatever token is required for identity assertion on the proxy side.
- The token form of the identity is transferred to the proxy server.
- The Subject is associated with the channel for subsequent calls.

The proxy side of the equation is discussed in the identity assertion slides.



Extend Pluggable Identity: Identity Transformer

- Coherence*Extend client produces token:
  - `IdentityTransformer` interface
    - `Object transformIdentity(Subject subject)`
      - `Subject` from current security context (may be `null`)
      - `Subject` is transformed into a security token
      - Token can be any **serializable** type
    - Token will be passed to proxy only once when:
      - A new connection opened
      - A new channel for an existing connection
  - `DefaultIdentityTransformer` returns a Serializable `Subject` as the token.

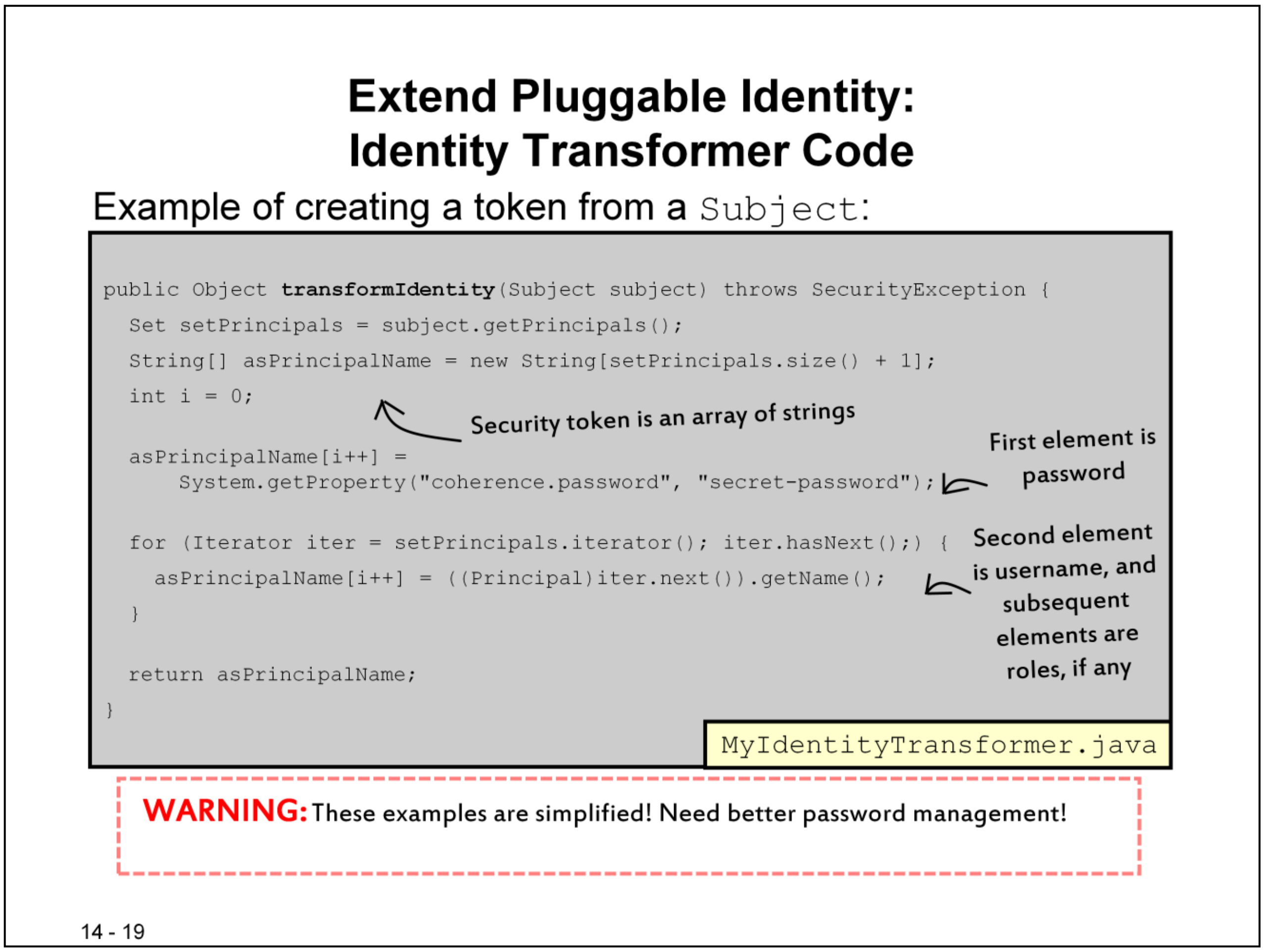
14 - 18

The `transformIdentity()` method can be implemented with custom code that converts the `Subject` to a token to work with any corporate security standard. The `transformIdentity()` method is called, and the security token is passed to the proxy server up to three times for an authenticated user:

- If it is the first TCP connection, the `transformIdentity()` method is called and the token passed to the proxy server when the connection is created.
- The connection is multiplexed, using a Coherence mechanism called channels. Each new `Subject` that connects to Coherence gets its own channel and:
  - When the channel is *created*, the `transformIdentity()` method is called, and the token is passed.
  - When the channel is *accepted*, the `transformIdentity()` method is called, and the token is passed.

This is required to avoid security vulnerabilities.

The out of the box `DefaultIdentityTransformer` returns a Serializable version of a `Subject`.



Extend Pluggable Identity: Identity Transformer Code

Example of creating a token from a `Subject`:

```
public Object transformIdentity(Subject subject) throws SecurityException {
  Set setPrincipals = subject.getPrincipals();
  String[] asPrincipalName = new String[setPrincipals.size() + 1];
  int i = 0;

  asPrincipalName[i++] =
      System.getProperty("coherence.password", "secret-password");

  for (Iterator iter = setPrincipals.iterator(); iter.hasNext();) {
    asPrincipalName[i++] = ((Principal)iter.next()).getName();
  }

  return asPrincipalName;
}
```

Security token is an array of strings

First element is password

Second element is username, and subsequent elements are roles, if any

MyIdentityTransformer.java

WARNING: These examples are simplified! Need better password management!

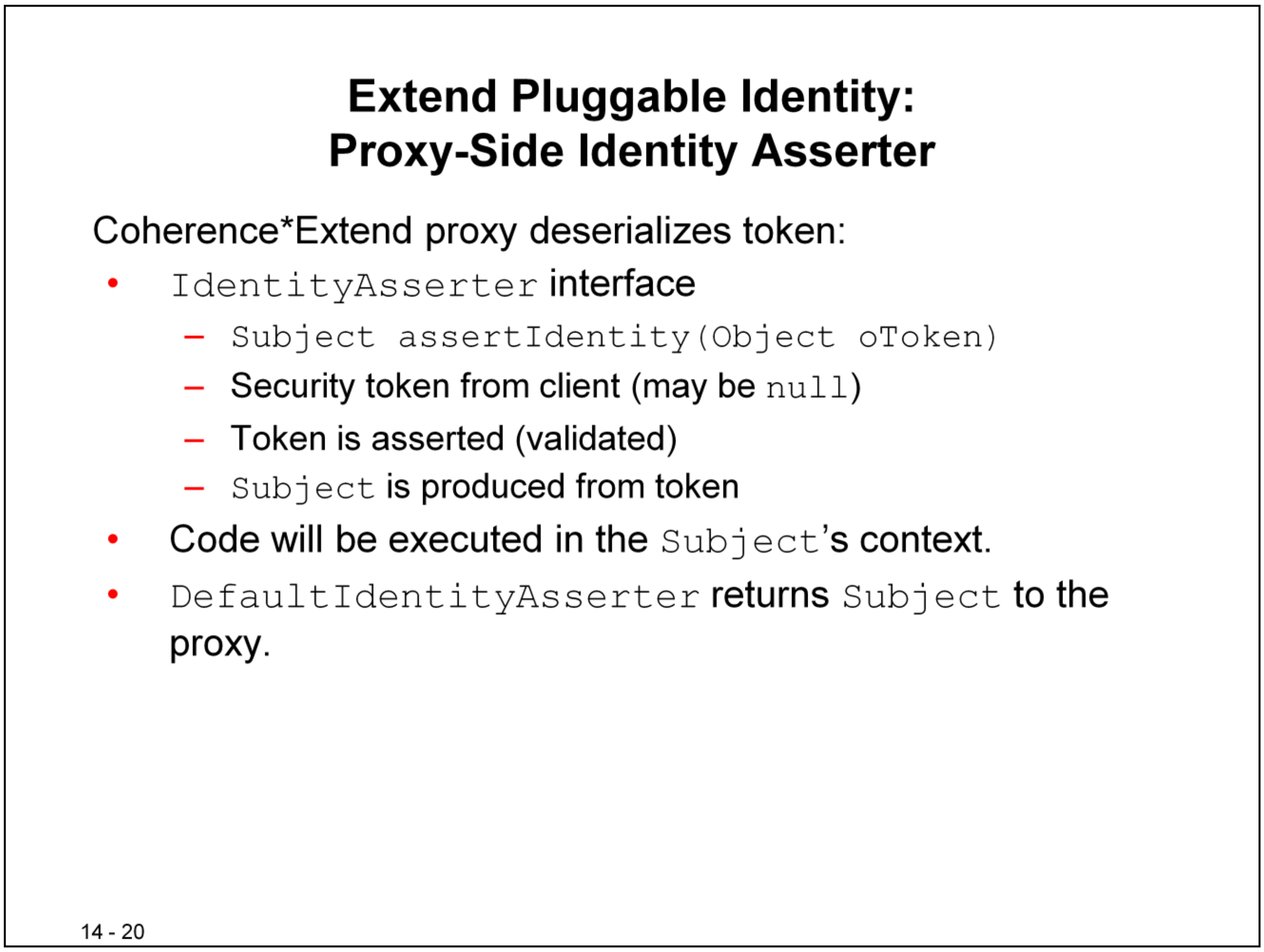
14 - 19

This is a code example of the `transformIdentity()` method that converts a Subject into an array of Strings, which is returned as the security token:

- The first element of an array is the password.
- The second element is the username.
- Subsequent elements are role names.

This is a simplified example! A real world version would require:

- A better mechanism for getting the password other than the clear text system property shown here
- Signing and/or encryption of token



Extend Pluggable Identity: Proxy-Side Identity Asserter

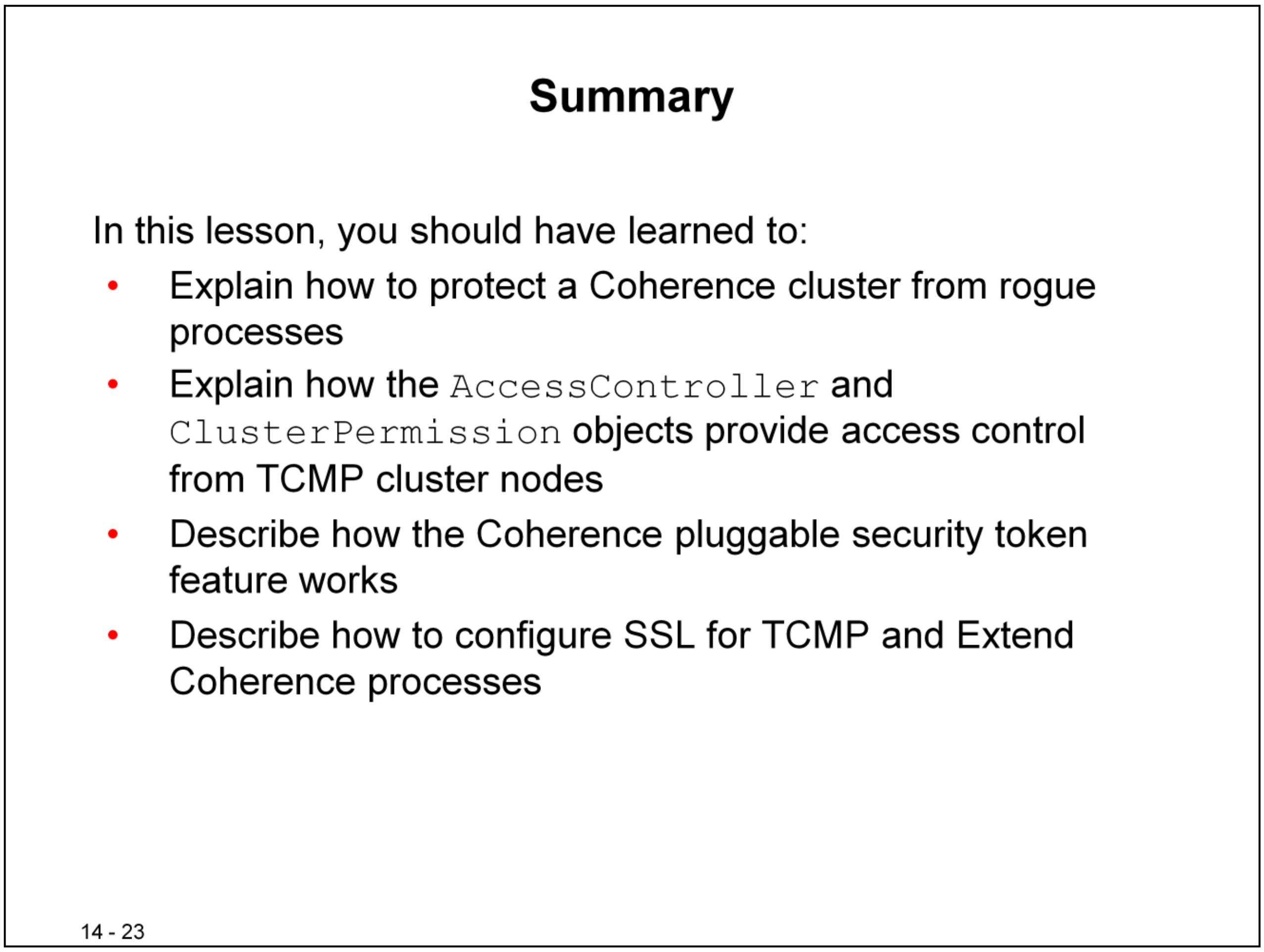
Coherence*Extend proxy deserializes token:

- `IdentityAsserter` interface
  - `Subject assertIdentity(Object oToken)`
  - Security token from client (may be `null`)
  - Token is asserted (validated)
  - `Subject` is produced from token
- Code will be executed in the `Subject`'s context.
- `DefaultIdentityAsserter` returns `Subject` to the proxy.

14 - 20

The `assertIdentity()` method can be implemented with custom code that converts the token created by `transformIdentity()` back into a `Subject` that the proxy server can use. The `assertIdentity()` method is called for an authenticated user as the connections and channels are created per each `Subject.` The `assertIdentity()` code is responsible for validating the token. When the proxy server has validated and associated the `Subject`, all subsequent calls on that channel are executed in the context of that `Subject`.

The `DefaultIdentityAsserter` supplied with Coherence returns a `Subject` to the proxy.



Summary

In this lesson, you should have learned to:

- Explain how to protect a Coherence cluster from rogue processes
- Explain how the `AccessController` and `ClusterPermission` objects provide access control from TCMP cluster nodes
- Describe how the Coherence pluggable security token feature works
- Describe how to configure SSL for TCMP and Extend Coherence processes

In this lesson, you learned about the various security features available in Coherence that protect application data from external attacks. You should now be able to explain these features, and how they work to others.