Proxy from Users IDENTIFIED EXTERNALLY

Perhaps your organization already uses the Oracle Internet directory (OID) and/or enterprise user security, if so it is possible to create individual person users who do not have schemas on each instance of Oracle Database, and can be used in proxy sessions. Those users would be `IDENTIFIED GLOBALLY`.

Perhaps, you trust another directory service or the operating system to identify your users. In that case, you would still have a unique schema for each user, but the authentication (ID and password) would be retained externally. For instance, you might set an `OS AUTHENT PREFIX` like "OPS$" (typical) for your database, and create an Oracle user named `OPS$OSUSER`. The `OSUSER` user would not provide a password when connecting to Oracle database, but would gain access by virtue of being authenticated to the operating system as `OSUSER`. There are steps required to set this up, including gaining access to your directory or domain server from the Oracle database. This is a form of single sign-on.

However, what we are doing is intentionally different. We are creating individual users in Oracle Database that have no authentication. They have no password, and the IDENTIFIED EXTERNALLY modifier simply tells Oracle Database that the user is not authenticated by the database.

Another way to have done this would be to create the users identified by a random password that not even the administrator retains. If no one knows the password, no one can authenticate with it. The problem with that is that any password that exists needs to be managed and at least periodically changed (to another random password).

Establish a Proxy Session

To establish a proxy session, we do 2 things. First we create a `Properties` class (basically a hash table with keys and values, e.g., key=PROXY_USER_NAME and value=OSUSER). Then we pass this `Properties` class to the `openProxySession()` method of the `OracleConnection` class, as shown in Listing 8-9. This code is from the `doTest2()` method of the `OraSSOTests` class. Also see the `main()` method.

■ **Note** Find the test code in the file named *Chapter8/OraSSOTests.java.*

Note that we already have an existing connection at this point. We are connected to Oracle database as our application user. The goal here is to have our OS user account proxy through the application user.

Listing 8-9. *Open a Proxy Session, `doTest2()`*

```
userName = OracleJavaSecure.getOSUserID();
Properties prop = new Properties();
prop.setProperty( OracleConnection.PROXY_USER_NAME, userName );
conn.openProxySession(OracleConnection.PROXYTYPE_USER_NAME, prop);

String metrics[] =
    new String[OracleConnection.END TO END STATE INDEX MAX];
metrics[OracleConnection.END TO END CLIENTID INDEX] = userName;
conn.setEndToEndMetrics( metrics, ( short )0 );
```

Notice we get the OS user identity into `userName` from `NTSystem` or `UnixSystem` and set it as the `PROXY USER NAME`. When we open the session, we tell it we are basing the proxy on user name, `PROXYTYPE USER NAME`.

We also set the client identifier to the OS user name—that is a good handle for searching audit logs. At this point we have a proxy session, and we can validate it with this query:

```
SELECT USER
, SYS CONTEXT('USERENV','PROXY USER')
, SYS CONTEXT('USERENV','OS USER')
, SYS CONTEXT('USERENV','SESSION USER')
, SYS CONTEXT('USERENV','OS USER')
, SYS CONTEXT('USERENV','IP ADDRESS')
, SYS CONTEXT('USERENV','TERMINAL')
, SYS CONTEXT('USERENV','CLIENT IDENTIFIER')
FROM DUAL;
```

This will return a series of identity values like this:

```
user                 : OSUSER
userenv proxy user   : APPUSR
userenv current user : OSUSER
userenv session user : OSUSER
userenv os user      : OSUser (occasionally OrgDomain\OSUser)
userenv ip address   : 127.0.0.1
userenv terminal     : unknown
userenv client id    : OSUSER
```

In our Oracle proxy session, the `USER`, `CURRENT USER`, and `SESSION USER` also are set to `OSUSER`. Oracle database saw that our OS user from JDBC was `OSUser`, as shown for the `OS USER` session environment value.

Finally, we connected as `appusr`, which allows proxying through by `OSUSER`, so we see `APPUSR` as the `PROXY USER`. If you look at the code of `OraSSOTests`, you'll see that we connect as `appusr`:

```
private String appusrConnString =
    "jdbc:oracle:thin:appusr/password@localhost:1521:orcl";

conn = (OracleConnection) DriverManager.getConnection( appusrConnString );
```

So we connected as `appusr`, but after establishing our proxy session, you can see that our user is `OSUSER`.

We close our proxy connections with a bit different syntax:

```
conn.close( OracleConnection.PROXY SESSION );
```

In this context (the `doTest2()` method), the effect is the same as the standard `conn.close()`, but for cached connections/connection pools this new syntax only closes the current session but keeps the connection available for others.

PROXY USER VERSUS PROXY USER NAME

Unfortunately, Oracle has used the words "proxy user" for both sides of the "connect as user" and "proxy through user" relationship. Proxy user, `proxy user`, is the Oracle user that connects to the database as in our session environment, `SYS CONTEXT( 'USERENV','PROXY_USER')`. And `PROXY USER NAME` is the name of the user getting access through the proxy user as when we set up our proxy connection.

prop.setProperty(OracleConnection.**PROXY_USER_NAME**, userName). Perhaps it would have been better to delineate these players with the names "proxy host user" and "proxy client user," or "proxy connect user" and "proxy session user."Despite the confusion, we will need to keep them straight. One user initially connects to Oracle database with his password (called the proxy user), and another user proxies through (connects through) that user. That second user owns the session that will do all the work and we will see him in our audit logs.

Update p_check_hrview_access Procedure, Proxy Sessions

Our secure application role procedure, appsec.p check hrview access must be updated again to verify proxy sessions and grant the hrview role, as appropriate. We have added the code shown in Listing 8-10 to the body of the procedure for that purpose (find this in the file *AppSec.sql*).

Listing 8-10. *Verify Proxy Session*

```
IF( SYS_CONTEXT( 'USERENV', 'PROXY_USER' ) = 'APPUSR'
AND ( SYS CONTEXT( 'USERENV', 'IP ADDRESS' ) LIKE '192.168.%' OR
    SYS CONTEXT( 'USERENV', 'IP ADDRESS' ) = '127.0.0.1' )
AND TO CHAR( SYSDATE, 'HH24' ) BETWEEN 7 AND 18
AND SYS CONTEXT( 'USERENV', 'SESSION USER' ) =
    SYS CONTEXT( 'USERENV', 'CLIENT IDENTIFIER' )
AND SYS CONTEXT( 'USERENV', 'CLIENT IDENTIFIER' ) = just os user )
THEN
    EXECUTE IMMEDIATE 'SET ROLE hrview role';
END IF;
```

The first test within the if statement assures us that we are dealing with a proxy session and that the proxy user is appusr. If you recall, we initially only allowed appusr to execute this procedure, but now we have granted execute to PUBLIC. However, we are still permitting access only to appusr by assuring that either the SESSION USER is appusr (when only setting client identifier) or that the PROXY USER is appusr (for proxy sessions).

Next we have the standard tests for IP Address and SYSDATE time constraints. Then we have two more tests that basically assure that these three identity traits are identical:

```
SESSION USER = CLIENT IDENTIFIER = OS USER
```

The user proxying this session is the same as the OS user we got from NTSystem or UnixSystem and is the same as the OS user that is presented by JDBC to Oracle Database. If all that is true, then we set the secure application role, hrview role.

Audit Proxy Sessions

We want to audit activity specific to proxy sessions. We can configure that with these commands:

```
AUDIT UPDATE TABLE, INSERT TABLE BY appusr ON BEHALF OF ANY;
-- This would be nice, but every java class gets audited with this command
--AUDIT EXECUTE PROCEDURE BY appusr ON BEHALF OF ANY;
NOAUDIT EXECUTE PROCEDURE BY appusr ON BEHALF OF ANY;
```

Because appusr is the proxy user, we can audit whatever he does on behalf of others. Here we are auditing all update and insert queries. We decided against auditing all calls to execute procedures.

These queries will display audit trail entries generated by proxy sessions. The first query displays all proxy sessions—they have a PROXY_SESSIONID.

```
SELECT * FROM DBA AUDIT TRAIL WHERE PROXY SESSIONID IS NOT NULL;
```

This next query finds the specific proxy connection associated with the session and shows the proxy user for the connection. The results might appear as shown in Table 8-1. Our query actually returns more columns than those shown in the Table.

```
SELECT p.username proxy, u.os username, u.username, u.userhost, u.terminal,
u.timestamp, u.owner, u.obj name, u.action name, u.client id, u.proxy sessionid
FROM sys.dba audit trail u, sys.dba audit trail p
WHERE u.proxy sessionid = p.sessionid
ORDER BY u.timestamp DESC;
```

Table 8-1. *Example Audit Log Entries for Proxy Session*

PROXY	OS_USERNAME	USERNAME	ACTION_NAME	CLIENT_ID	PROXY_SESSIONID
APPUSR	OSUSER	OSUSER	LOGOFF	OSUSER	68325
APPUSR	OSUSER	OSUSER	LOGON		68325

Using Connection Pools

If you are strictly concerned with desktop client applications, then you can skip this section. Connection pools are generally only needed for multi-threaded, multi-user Server applications. A connection pool is a collection of connections that are available for use by clients or client threads. As needed, a client will acquire a connection from the pool, use it to query or update the Oracle database, and then return it to the pool.

A connection pool usually exists for the duration of the JVM. Consider this scenario: a web application server (for example, Tomcat) starts running and an application requests a connection from a pool. At that point, a pool of connections is established, and one of the connections is provided to the application. That specific application thread (typically tied to a user request—a browser seeking a dynamic web page) will return the connection to the pool when the thread is done with it. Our connection pool exists for use by all the web application threads (users browsing dynamic web pages) until the web application server (Tomcat) is shut down.

We are going to spend sufficient effort here to demonstrate that our single sign-on will work from any of the available connection pooling approaches. If you are using a Java Enterprise Edition (J2EE) container (like a web application server) and you use Enterprise Java Beans (EJBs), then chances are good that you are using connection pooling by way of a "container-managed connection pool."

Proxy Connections from an OCI Connection Pool

An Oracle call interface (OCI) connection pool is a traditional approach to connection pools, and our approach to single sign-on can succeed within it. OCI is a technology that supersedes Java and is not a "pure java" implementation. We would say that using ojdbc, Java is able to call OCI as an external resource.

Configure the Pool

The first step we must take is to establish the connection pool through an `OracleOCIConnectionPool` class. We set the URL (connection string), user, and password (`appusrConnURL`, `appusrConnUser`, `appusrConnPassword`) that all connections in the pool will have, as shown in Listing 8-11. They will all connect as `appusr`.

Be sure to edit the code in *OraSSOTests.java* to correctly identify your host, port, instance and network domain (in `SERVICE NAME`). Notice that the connection URL is specified as "jdbc:oracle:oci:". That is the designation of a heavyweight, OCI connection type. The connection string is specified in *TNSNames* (transparent network substrate) format, `appusrConnOCIURL`.

Listing 8-11. *Configure OCI Connection Pool*

```
private String appusrConnOCIURL =
    "jdbc:oracle:oci:@(description=(address=(host=" +
    "127.0.0.1)(protocol=tcp)(port=1521))(connect data=" +
    "(INSTANCE NAME=orcl)(SERVICE NAME=orcl)))";
    // Or
    //"(INSTANCE NAME=orcl)(SERVICE NAME=orcl.org.com)))";
private String appusrConnUser = "appusr";
private String appusrConnPassword = "password";

OracleOCIConnectionPool cpool = new OracleOCIConnectionPool();
cpool.setURL(appusrConnOCIURL);
cpool.setUser(appusrConnUser);
cpool.setPassword(appusrConnPassword);

Properties prop = new Properties();
prop.put (OracleOCIConnectionPool.CONNPOOL MIN LIMIT, "2");
prop.put (OracleOCIConnectionPool.CONNPOOL MAX LIMIT, "10");
prop.put (OracleOCIConnectionPool.CONNPOOL INCREMENT, "1");
cpool.setPoolConfig(prop);
```

Next, we configure the connection pool by building a `Properties` object with the basic parameters: the minimum pool size (also initial size), maximum pool size, and growth increment. We pass the `Properties` to our connection pool through the `setPoolConfig()` method. Note that these properties apply to the pool itself, not to any specific connection.

Get the Proxy Connection

We add one more parameter to the existing (for convenience) `Properties` object—the `PROXY USER NAME`. When we get a connection from the pool, we specifically want a proxy connection. All connections in the pool connect as `appusr`, but each proxy connection coming from the pool can be associated with a different user connecting through `appusr`. We set the `PROXY USER NAME` for this connection to `userName`, the OS user identity that we got from `NTSystem` or `UnixSystem`. When we request a proxy connection from the `cpool.getProxyConnection()` method, Listing 8-12, we pass the `Properties` with the `PROXY USER NAME` in our request.

Listing 8-12. *Get Proxy Connection from OCI Connection Pool*

```
prop.setProperty(OracleOCIConnectionPool.PROXY USER NAME, userName );
conn = (OracleConnection)cpool.getProxyConnection(
    OracleOCIConnectionPool.PROXYTYPE USER NAME, prop);
```

For this connection from an OCI pool, we set the client identifier the same way we did previously for standard connections.

View the Proxy Session

At this point, if you query the `OracleConnection.isProxySession()` method, you will find that this is not a proxy session. Don't let that disturb you. If you query Oracle Database you will find that the `PROXY USER` is `appusr`, and the `USER`, `CURRENT USER`, and `SESSION USER` are all set to our OS user identity. This is, by virtue of acquiring it through the `getProxyConnection()` method, a proxy connection and a proxy session.

It is unnecessary, but if you must have the `isProxySession()` method return `true`, you can generate another session through the `OracleConnection` class—you would have three sessions at that point. Using this code will also nullify your client identifier, which will interfere with the procedure that sets our secure application role. So if you do this, modify the SSO procedures (e.g., `appsec.p check hrview access`) to skip testing the client identifier.

```
//prop = new Properties();
//prop.setProperty(OracleConnection.PROXY USER NAME, userName );
//conn.openProxySession(OracleConnection.PROXYTYPE USER NAME, prop);
```

Note that this code instantiates a new `Properties` class. The reason for that is that our prior `Properties` had numeric keys based on the `OracleOCIConnectionPool` constants, and this `Properties` instance needs `String` keys based on the `OracleConnection` constants.

See the Proxy Connection

Here are the results from our test queries in the `OraSSOTests.doTest3()` method. These results are from an OCI connection pool proxy connection.

```
Is proxy session: false
user                 : OSUSER
userenv proxy user   : APPUSR
userenv current user : OSUSER
userenv session user : OSUSER
userenv os user      : ORGDOMAIN\OSUSER
userenv ip address   : 127.0.0.1
userenv terminal     : MYCOMPUTER
userenv client id    : OSUSER
Read HR view!!!!!!!!!!!!!!!!!!!
```

Notice the first line. As described in the previous section, the `OracleConnection.isProxySession()` method returns `false`; however, this is a proxy session through a proxy connection. You can see that in the `PROXY USER` identity trait, contrasted with the other user traits.

One of the nice things about heavy-weight OCI connections is that they can report the terminal name. Notice also that in the `OS USER` parameter the OCI connection reports the domain name. It may be that you can implement some additional security checks based on those identity traits.

Close the Proxy Session

Our proxy connection close method, in this context, closes the proxy session and returns the connection to the pool for reuse. Notice in Listing 8-13 that we pass the `PROXY SESSION` constant value to the `OracleConnection.close()` method. This is identical to how we close the proxy connection without connection pooling, except without pooling both the proxy session and the Oracle connection get closed.

Listing 8-13. *Close Proxy Connection*

```
conn.close( OracleConnection.PROXY SESSION );
```

Review All the Pool Properties

We can examine all the properties that are used by the connection pool, even those properties that we did not explicitly set. The code in Listing 8-14 gets the complete list of configuration properties and prints each key/value pair.

Listing 8-14. *Display all OCI Connection Pool Properties*

```
prop = cpool.getPoolConfig();
Enumeration enumer = prop.propertyNames();
String key;
while( enumer.hasMoreElements() ) {
    key = (String)enumer.nextElement();
    System.out.println( key + ", " + prop.getProperty( key ) );
}
```

We see the following list as the result from that code:

```
connpool active size, 0
connpool pool size, 2
connpool max limit, 10
connpool min limit, 2
connpool timeout, 0
connpool is poolcreated, true
connpool increment, 1
connpool nowait, false
```

Summary of the OCI Connection Pool

The OCI connection pool handles our single sign-on approach very well. We are able to verify the user, pass our secure application role procedure tests and access HR data.

Our `OracleConnection` class from an OCI connection pool does not recognize the proxy session, but a proxy session exists nonetheless.

OCI connection pool connections recognize the terminal name and can return the Domain name along with the OS user name.

Here is the catch. An OCI connection pool *is very slow to be set up*, the connections are heavyweight and demand more resources. Once set up, and using existing connections in the pool, performance is not an issue. For this reason, an OCI connection pool is suitable for server-based applications with a long runtime duration (e.g., within a web application server).

Proxy Sessions from a Thin Client Connection Pool

While connection pools are generally only needed for Server applications supporting multiple, concurrent users, you might have some call for a multi-threaded, stand-alone or client application that will perhaps need to reuse multiple connections from a pool—be creative! A thin-client connection pool or cache is a great way to accomplish this. Our approach to single sign-on can succeed within it.

Configuring the Pool/Cache

We establish the lightweight (thin) connection pool through an `oracle.jdbc.pool.OracleDataSource` class. Set the URL (connection string), user, and password that all connections in the pool will have. They will all connect as `appusr`.

Edit the connection string (URL), `appusrConnURL`, setting the appropriate host, port, and instance; and set the correct password for `appusr` in `appusrConnPassword`. The URL is specified in TNSNames format. Notice in Listing 8-15 that the URL specifies a thin (light-weight) connection as "jdbc:oracle:thin". Thin connections require fewer resources and are set up faster than OCI connections. Thin connections use only java to communicate by SQL*Net protocol with Oracle Database, as opposed to using OCI as an external, non-java resource to run SQL*Net. We name our connection cache (pool), "APP_CACHE."

Listing 8-15. *Configure Thin-Client Connection Pool*

```
private String appusrConnThinURL =
    "jdbc:oracle:thin:@(description=(address=(host=" +
    "127.0.0.1)(protocol=tcp)(port=1521))(connect data=" +
    "(INSTANCE NAME=orcl)(SERVICE NAME=orcl.org.com)))";

OracleDataSource cpool = new OracleDataSource();
cpool.setURL(appusrConnThinURL);
cpool.setUser(appusrConnUser);
cpool.setPassword(appusrConnPassword);

// Enable Connection Caching
cpool.setConnectionCachingEnabled(true);
cpool.setConnectionCacheName("APP_CACHE");

Properties prop = new Properties();
prop.setProperty("InitialLimit", "3");
prop.setProperty("MinLimit", "2");
prop.setProperty("MaxLimit", "10");
cpool.setConnectionCacheProperties(prop);
```

We will also set up some initial properties for the connection pool and pass them to the `setConnectionCacheProperties()` method. These are similar to the pool properties we saw earlier for the OCI connection pool, but the key names are quite different.

Use Statement Caching

You will see some confusion when doing research on the web regarding lightweight connection pooling. Much of this confusion stems from the availability of statement caching in the `OracleDataSource` class. It is particularly muddled by the reference to connection pooling as connection caching. For good measure, let's enable statement caching as in Listing 8-16. This has nothing to do with connection pooling.

Listing 8-16. *Enable Statement Caching*

```
cpool.setImplicitCachingEnabled(true);
```

With statement caching, when you call a prepared statement, the local connection caches it. Should you call the statement again, it executes faster than it would if not cached.

Implicit caching, which we enabled by calling `setImplicitCachingEnabled(true)`, happens automatically. You can also enable *explicit* statement caching which requires you to designate a key string (name) for the statement and to recall it by that name. For more information, do an Internet search on Oracle explicit statement caching.

Get the Proxy Session

We request a thin connection from the pool through the `getConnection()` method, as shown in Listing 8-17. After getting the pooled connection, we request a proxy session. Pass the user identity from `NTSystem` or `UnixSystem` as a property value to the `openProxySession()` method of the `OracleConnection`. Again, for convenience, we just add another value, `PROXY USER NAME` to the *existing* `Property` class.

Listing 8-17. *Get Proxy Connection from Thin-Client Connection Pool*

```
conn = (OracleConnection)cpool.getConnection();
prop.setProperty(OracleConnection.PROXY USER NAME, userName );
conn.openProxySession(OracleConnection.PROXYTYPE USER NAME, prop);
```

When we close the proxy session, the connection will return to the pool for reuse:

```
conn.close( OracleConnection.PROXY SESSION );
```

See the Proxy Session

The results from our queries on the thin client proxy session are as we would expect. We have set both our client identifier and our proxying user, `osuser` through the proxy user, `appusr`. The connection passes the tests for our secure application role and is able to read data from the HR schema.

```
Is proxy session: true
user                 : OSUSER
userenv proxy user   : APPUSR
userenv current user : OSUSER
```

```
userenv session user : OSUSER
userenv os user      : OSUSER
userenv ip address   : 127.0.0.1
userenv terminal     : unknown
userenv client id    : OSUSER
Read HR view!!!!!!!!!!!!!!!!!!!
```

Review All the Pool Properties

We can also retrieve and review all the connection pool properties:

```
prop = cpool.getConnectionCacheProperties();
```

```
MaxStatementsLimit, 0
AbandonedConnectionTimeout, 0
MinLimit, 2
TimeToLiveTimeout, 0
LowerThresholdLimit, 20
InitialLimit, 3
ValidateConnection, false
ConnectionWaitTimeout, 0
PropertyCheckInterval, 900
InactivityTimeout, 0
LocalTransactionCommitOnClose, false
MaxLimit, 10
ClosestConnectionMatch, false
AttributeWeights, NULL
```

COMPILING, DEPRECATED METHODS AND ANNOTATIONS

When you compile OraSSOTests.java you will receive two warnings. Two of the methods we are calling here are deprecated; that is, they are no longer promoted as current programming practices. You can see what methods those are by providing an argument on the command line, like this:

```
javac -Xlint:deprecation OraSSOTests.java

OraSSOTests.java:265: warning: [deprecation] setConnectionCachingEnabled(boolean
) in oracle.jdbc.pool.OracleDataSource has been deprecated
                        cpool.setConnectionCachingEnabled(true);
                              ^
OraSSOTests.java:274: warning: [deprecation] setConnectionCacheProperties(java.u
til.Properties) in oracle.jdbc.pool.OracleDataSource has been deprecated
            cpool.setConnectionCacheProperties(prop);
                  ^
2 warnings
```

Wouldn't you know it, these are the two methods we are calling to enable and configure our connection pool (cache)!?! Why are they deprecated? Even with those warnings, the code will still compile correctly and will run just fine. It is rare, in my experience, that a deprecated method actually disappears—sometimes they are actually revived. Fear not!

These methods are only recently deprecated, and I suspect the reason is that Oracle has developed a new package to do thin-client connection pooling, universal connection pool (UCP). We will implement UCP later in this chapter.

We can keep these deprecation warnings from bothering us when we compile by placing a @SuppressWarnings *annotation* on the method with our deprecated calls as shown below. Notice that there is no semicolon at the end of the annotation; it applies to the following method, `doTest4()`.

```
@SuppressWarnings("deprecation")
void doTest4() {
```

Annotations are not part of the Java syntax, but they are included in the Java code and the byte code (after compilation) in order to instruct the Java utilities and tools (like *java.exe* and *javac.exe*). The variety of uses of annotations are quite numerous, and you can extend them by defining your own annotation types and by writing Java code for use by the Java tools in responding to annotations.

Many annotations like this are not acceptable to the Oracle JVM, so we will comment them in our code before creating the Java structures in Oracle Database.

Summary of the Thin Client Connection Pool

Thin client connection pooling is quick and convenient. It supports our approach to SSO by proxy and by client identifier.

It is a little disturbing to see the key methods used for connection pooling being labeled as "deprecated." But that is not a roadblock.

Universal Connection Pool

UCP is the newest kid on the block for connection pooling. Because it is so new, it would behoove you to watch for and implement any updates to UCP.

■ **Note** Find the test code for UCP in the file named *Chapter8/OraSSOTests2.java.*

Compile/Run with UCP

As yet, the UCP package has not been incorporated into the Oracle drivers jar (ojdbc6.jar). You will need to download a separate *ucp.jar* file from Oracle at *www.oracle.com/technetwork/indexes/downloads*. Scroll down and find the file in the [**Drivers**] section.

Our test code for UCP is in a separate file, *OraSSOTests2.java*. When you compile and run this code, you will need to include the *ucp.jar* file in your `CLASSPATH`, as shown in Listing 8-18.

Listing 8-18. *Command to Compile and Run with UCP.jar*

```
javac -classpath "%CLASSPATH%";ucp.jar OraSSOTests2.java
java -classpath "%CLASSPATH%";ucp.jar OraSSOTests2
```

A better approach, if you are going to use UCP, would be to add it to your `CLASSPATH` in the OS environment, as described in Chapter 3.

Use a Connection Pool Factory

UCP uses a `PoolDataSourceFactory` class to instantiate the pool connections. We provide the `PoolDataSource` with the fully-qualified name of the class to be used for each connection, "oracle.jdbc.pool.OracleDataSource," and we set the URL (connection string) to be used for the connections. See Listing 8-19.

Listing 8-19. *Configure UCP Connection Pool*

```
private String appusrConnString =
    "jdbc:oracle:thin:appusr/password@localhost:1521:orcl";

PoolDataSource cpool = PoolDataSourceFactory.getPoolDataSource();
cpool.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
cpool.setURL( appusrConnString );

cpool.setInitialPoolSize(5);
cpool.setMinPoolSize(2);
cpool.setMaxPoolSize(10);
```

We call methods of the `PoolDataSource` class, `cpool` to set some properties of the connection pool.

Note That UCP Is a Thin Client Implementation

Notice that the URL we specify (above) is a lightweight, thin connection. We are also able to specify the URL in the straightforward connection string syntax, as shown in Listing 8-20. If desired, we could specify the URL in TNSNames format and provide the user and password in separate method calls. We can also specify the straightforward connection string syntax, leaving out the user and password, and provide user and password in separate method calls.

Listing 8-20. *Alternative UCP Connection Pool Specification*

```
cpool.setURL(appusrConnURL);
cpool.setUser(appusrConnUser);
cpool.setPassword(appusrConnPassword);
// Or
cpool.setURL("jdbc:oracle:thin:@localhost:1521:Orcl" );
cpool.setUser(appusrConnUser);
cpool.setPassword(appusrConnPassword);
```

One reason I very much prefer connection string syntax, including user name and password, is that I can provide them all the connection details, including password, from a secure external source as a single string. We will plumb the depths of this topic in Chapter 10 and Chapter 11. Providing connection strings from a secure external source means that we can do the following:

- Avoid embedding passwords in our application code.
- Centralize the storage, maintenance, and distribution of our connection strings (thereby allowing us to change the parameters of a connection string in one place and have the change applied wherever we use the connection).

Get and Use the UCP Connection

This will look very familiar. We get a connection from the pool, get a proxy session and set the client identifier exactly as we did for our non-UCP thin connection pool. The results are identical as well.

Summary of Universal Connection Pool

If you don't mind living a bit on the cutting edge, then UCP is the way to go. It will require you to watch for updates to the UCP package, and eventual inclusion of UCP in the Oracle drivers, *ojdbc6.jar*. You will also need to incorporate *ucp.jar* into your `CLASSPATH` (refer to Chapter 3). As far as our goal of Oracle SSO is concerned, UCP works without a hitch.

Application Use of Oracle SSO

The presumptive requirement we are making in this chapter is that the user has authenticated on the client. The goal we have in our code is to leverage that existing authentication and make it available to application developers so that they can tap into Oracle SSO without burdening them with the inner workings.

I'm going to implement only one of the five options for Oracle SSO that we've examined. We will select a non-pooled connection with proxy session (refer to the `doTest2()` method in *OraSSOTests.java*). If you want to implement non-proxy connections or pooled connections, you should be able to do that easily enough based on the *OraSSOTests.java* code.

Our Example Application Oracle SSO

We will examine this from the outside in; that is, from the application developers' point of view first. After we explore what the developer needs to do, we'll discuss what changes we need to make to the `OracleJavaSecure` class in order to support the developers.

Use the Application Oracle Connection

Each application will make connections to different Oracle instances as different application user accounts. That logic has to exist within the application. Our example application gets data from the `HR` schema as the `appusr` user, so we connect as that user in Listing 8-21.

Listing 8-21. *Application Oracle Connection Specification*

```
String urlString = "jdbc:oracle:thin:appusr/password@localhost:1521:orcl";
Class.forName( "oracle.jdbc.driver.OracleDriver" );
OracleConnection conn =
    (OracleConnection)DriverManager.getConnection( urlString );
```

■ **Note** You can find this code in the file named *Chapter8/AppOraSSO.java.*

Another approach, and one you would use if you were implementing a connection pool in the `OracleJavaSecure` class rather than in the client application, would *not* instantiate a connection in the application; rather, you would simply pass the application-specific URL, `urlString` to `OracleJavaSecure` in order to configure the connection pool. And you would get back an `OracleConnection` from the pool for use in your application. At least, that's how I would do it.

Get a Proxy Connection for SSO

Preferably, the developers can make a single method call to acquire a proxy connection that will successfully pass the tests in our secure application role procedure. Let's call that method `setConnection()`. The application developer would call it like this:

```
conn = OracleJavaSecure.setConnection( conn );
```

This would overwrite the existing `conn` with the `OracleConnection` returned by that method. In reality, remember that these are just references (pointers) to objects in memory, and there has been no new instance of this object created, so the object pointer has not changed. We passed a reference to the `OracleConnection` from the application to `OracleJavaSecure` (everything resides in a single JVM.) Then `OracleJavaSecure` set the proxy session and the client identifier on that `OracleConnection`. When we go to use it back in the application, those features are now part of our original `OracleConnection`. We can and will just use the syntax shown in Listing 8-22.

Listing 8-22. *Add Proxy Features to Existing Connection, setProxyConnection()*

```
OracleJavaSecure.setProxyConnection( conn );
```

The results will be the same—the original `OracleConnection`, `conn` now has the proxy features.

If we were passing the URL only, we would call this method, which returns an `OracleConnection` with proxy session and client identifier:

```
OracleConnection conn = OracleJavaSecure.setConnection( urlString );
```

Close the Proxy Connection

We want to allow for the possibility that these connections may come from a connection pool, and we want to be sure to close the proxy session in that case, so we will instruct the developers to call a method to close the connection, like this:

```
OracleJavaSecure.closeConnection();
```

Updates to OracleJavaSecure

The generic `Connection` class does not support proxy connections nor setting the client identifier. We will be using instances of `OracleConnection` from here out. Our static class member, `conn` is now an `OracleConnection`, see Listing 8-23. Also our static initializer for `conn` casts the `Connection` as an `OracleConnection`.

Listing 8-23. *OracleJavaSecure Static OracleConnection*

```
private static OracleConnection conn;

static {
    try {
        // The following throws an exception when not running within an Oracle Database
        conn = (OracleConnection)(new OracleDriver().defaultConnection());
    } catch( Exception x ) {}
}
```

Update setConnection() Method

We are going to overload the `setConnection()` method (see Listing 8-24), retaining one that takes a `Connection` parameter and adding one that takes an `OracleConnection`. The first will call the second so that both configure a proxy connection with the client identifier set. The OS user identity from `NTSystem` or `UnixSystem` is used for proxying and in the client identifier:

Listing 8-24. *Set Internal Connection in OracleJavaSecure and Configure, setConnection()*

```
public static final OracleConnection setConnection( Connection c ) {
    return setConnection( (OracleConnection)c );
}
public static final OracleConnection setConnection( OracleConnection c ) {
```

```
        conn = null;
        // We are going to require that only we will set up initial proxy connections
        if( c == null || c.isProxySession() ) return null;
        else try {
            // Set up a non-pooled proxy connection with Client Identifier
            // To use an alternate solution, refer to code in OraSSOTests.java
            String userName = getOSUserID();
            if ( ( userName != null ) && ( !userName.equals( "" ) ) ) {
                Properties prop = new Properties();
                prop.setProperty( OracleConnection.PROXY_USER_NAME, userName );
                c.openProxySession(OracleConnection.PROXYTYPE USER NAME, prop);

                String metrics[] =
                    new String[OracleConnection.END TO END STATE INDEX MAX];
                metrics[OracleConnection.END TO END CLIENTID INDEX] = userName;
                c.setEndToEndMetrics( metrics, ( short )0 );

                // If we don't get here, no Connection will be available
                conn = c;
            } else {
                // This is not a valid user
            }
        } catch ( Exception x ) {
            x.printStackTrace();
        }
        return conn;
    }
```

This code is exactly what we have been discussing for Oracle SSO. Notice that if the connection that is handed to us is already a proxy connection, we discard it and nullify our `Connection`. We will not recognize any proxy connections that are configured elsewhere—for security sake, we are jealous and protective of our role in this.

Add an Overloaded setConnection() Method

We are returning the resultant, configured `OracleConnection` from the `setConnection()` methods listed in 8-24, and these will support an additional `setConnection()` method. The additional method (Listing 8-25) takes the URL as a `String`, instantiates a `Connection` and calls the core setConnection() method, which configures the connection with a proxy session and client identifier. It then returns the configured `OracleConnection`. It is made relatively easy with a call to the other `setConnection()` methods, and returning what they return.

Listing 8-25. *Alternate Set Internal Connection, setConnection()*

```
    public static final OracleConnection setConnection( String URL ) {
        Connection c = null;
        try {
            Class.forName( "oracle.jdbc.driver.OracleDriver" );
            c = DriverManager.getConnection( URL );
        } catch ( Exception x ) {
            x.printStackTrace();
```

```
        }
        return setConnection( c );
    }
```

■ **Note** If you were going to implement a connection pool inside the `OracleJavaSecure` class, this is where you would do it. You would want to assure that the URL string had not changed with each subsequent call.

Close the Proxy Connection

Have the developers call a method on `OracleJavaSecure` to close the configured proxy connection, in order to assure the call is appropriate (Listing 8-26). We must assure that we close the proxy session, which may be an additional session on a single connection.

Listing 8-26. *Close the Internal Connection, closeConnection ()*

```
    public static final void closeConnection() {
        try {
            conn.close( OracleConnection.PROXY_SESSION );
        } catch( Exception x ) {}
    }
```

A Code Template to Give Developers

The lines shown in Listing 8-27 are all the Java that application developers will need in order to do Oracle SSO, by using the `OracleJavaSecure` class.

Listing 8-27. *Method Calls for Application Developers*

```
    OracleConnection conn = OracleJavaSecure.setConnection( connectionString );
    // Do Oracle queries here
    OracleJavaSecure.closeConnection();
```

Refer to the instructions at the end of Chapter 7 in order to create a jar file containing `OracleJavaSecure` to give the application developers.

Note that there will also need to be a secure application role, protected by a procedure like the `p check hrview access` in order to complete the SSO protection of our data.

In the case where we are doing proxy connections, there will need to be an Oracle user for each OS user identity that we want to have accessing our applications. And those Oracle individual person users will need to be granted "proxy through" on our application user.

Chapter Review

In this chapter, we discussed how to identify the Windows or UNIX user using the JAAS classes, `NTSystem`, and `UnixSystem`. Because those classes are not provided cross-platform, we delved into using reflection to both instantiate and call methods in those classes.

After establishing the OS identity, we explored the code needed to use that identity when authenticating to Oracle Database. The end goal is that our Oracle application users will be able to use our applications without entering their passwords. In fact, they won't need passwords at all on Oracle database, but we will be able to track each user's actions by one of two methods:

1. We set the connection client identifier equal to the user ID and then assure it exists in the connection before granting access. We can then find that client ID in the audit trail logs.
2. We create an Oracle user for each Windows/UNIX user, although the Oracle user does not need a password or any privileges besides `CONNECT`. We then proxy those individual users through our application user. We can query the audit logs for the proxy user identity.

We explored several varieties of Oracle connection pools we might establish. There are three connection pool technologies we may use:

1. Lightweight, thin client connection pools
2. Heavyweight, OCI connection pools
3. Universal connection pooling (UCP)

Finally, we built some `setConnection()` methods in `OracleJavaSecure` that make it easy for an Oracle application developer to tap into this technology.

Figure 8-1 illustrates the basic SSO process. The client application calls `OracleJavaSecure.setConnection()` method with an appropriate connection string. That method further calls `getOSUserID()` which gets the user ID (username) from the operating system using `NTSystem` or `UnixSystem`, as appropriate. With that OS user ID, we open a proxy session for our `Connection` through an Oracle user by the same name.

Using our proxy session, we call the `p check hrview access` procedure, which assures our SSO credentials are in order, then sets the secure application role, `hrview role`. At that point, we have done SSO to Oracle database—notice we didn't enter a password for the specific Oracle user. We can then select sensitive data from the `EMPLOYEES` table using the privileges granted to `hrview role`.

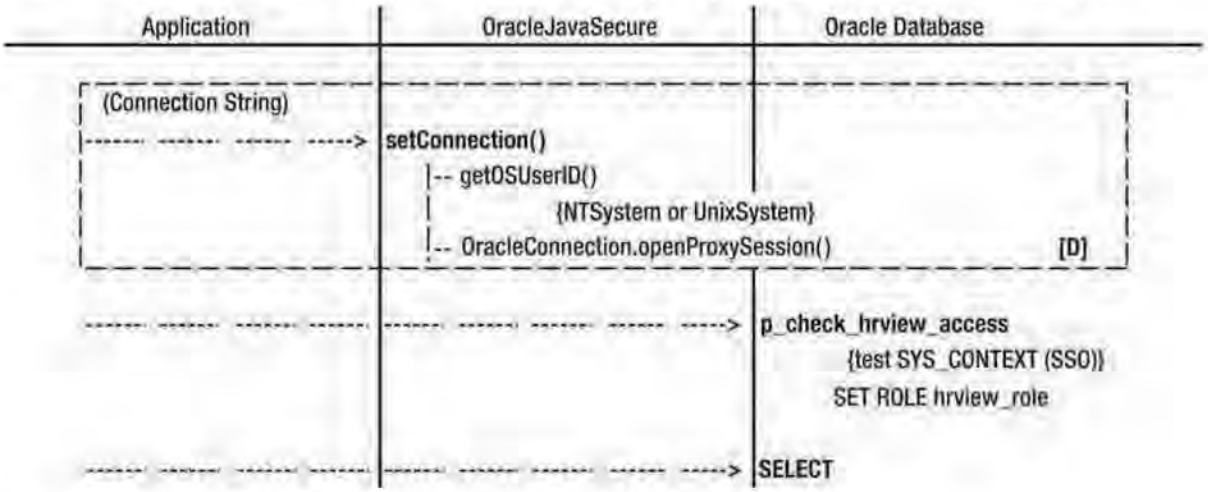


Figure 8-1. SSO procedure

CHAPTER 9

Two-Factor Authentication

What would life be like if there were no imposters, no charlatans, and no thieves? Sorry, that rhetorical question does not provide any security. Neither does putting on a set of rose-colored glasses and believing that we are secure just because we've implemented substantial security measures. We will always be susceptible to trickery and carelessness. The weakest link, even with upright associates, is always a person taking shortcuts. Social engineering and lack of attention to our computer security code of conduct (like, don't write down your password, don't share your password, use complex passwords, and change your password periodically) give thieves entrance to our most secure systems.

So, we are looking for further constraints on identity that can assure us that the person sitting at the keyboard is who they claim to be. There are many things that are being done in computer security to attempt to achieve this, such as the following:

- Requiring a second password or PIN code.
- Assuring there is a person at the computer instead of an automated program by having the person enter a non-computer-readable graphic representation of a word, called a CAPTCHA, which stands for completely automated public turing test to tell computers and humans apart (named in part after Alan Turing, the father of computer science and artificial intelligence).
- Requiring the user to answer personal questions, like giving the name of their first pet.
- Having a biometric scanner, like fingerprint, retina, or facial recognition.
- Having a secure ID token that synchronizes a code with the server to provide a one-time password, along with a PIN code.
- Out-of-band communication to a separate account or device, e.g., pass codes sent to your e-mail, pager, or cell phone.

Several of these efforts can be considered two-factor authentication. Combining them, you can even achieve three-factor authentication. For example:

1. What you know (passwords and PIN)
2. Who you are (human and biometric)
3. What you have (a Secure ID token or cell phone)

Perhaps a second password or additional PIN could also be considered two-factor authentication, but not so much. It is still just (1) what you know.

We are going to implement two-factor authentication that uses the single sign-on from Chapter 8, and a code that we will send to a separate account, preferably on a separate device. We will send pass codes to pagers, cell phones, and, as a last resort if neither of the first two is available, to an e-mail account.

Get Oracle Database to Send E-Mail

Our two-factor authentication will primarily contact pagers and cell phones and not send the pass code to a user's e-mail account. However, we are going to have the e-mail option. Also, the easiest way to send messages to commercial cell phones is by using the cell phone providers' short message service (SMS), texting hosts that often accept e-mail messages bound for the cell phones. So we are going to implement e-mail from Oracle database.

Oracle database provides a package called `UTL_MAIL` that enables us to send e-mail from the database. We will implement that; although, we could also load a Java class to send e-mail, and configure it to execute from a Java stored procedure. Perhaps we would use the JavaMail API or we might open a plain Java `Socket` and write simple mail transport protocol (SMTP) commands to it (so we don't have to load the JavaMail *mail.jar* file into Oracle database). Later in this chapter we will call a Java-stored procedure to read a web page, and sending e-mail could be done similarly.

Installing UTL_MAIL

The `UTL_MAIL` package is not installed in Oracle database by default. We will have to install it manually. Actually, we will ask the `SYS` user to install it for us. The package resides in two files that are located in a server directory resembling this path: *oracle\product\11.2.0\dbhome_1\RDBMS\ADMIN*. The files are called *utlmail.sql* and *prvtmail.plb*.

A *.plb* file is a wrapped PL/SQL file. Wrapped files can be considered obfuscated (not easily read), but not encrypted. The wrapped format lends severe difficulty to reverse engineering, but doesn't prevent it; although, you'd need to be a hacker to obtain the resources to unwrap the file. To create a wrapped procedure, function, package, or type, you pass a *.sql* file containing the `CREATE` statement for those structures to the wrap utility (comes with the Oracle Database software). This creates a non-reversible *.plb* file. Make sure you save an archive of the original *.sql* file in a secure location in case you ever need to edit it. Wrapped package files are used to hide the PL/SQL code, perhaps to protect intellectual property or to otherwise add security.

As `SYS`, you or your DBA friend need to open each of these files in order: *utlmail.sql* then *prvtmail.plb*; and execute them within the database. This can be done through TOAD, for example, by loading the files into the SQL Editor and executing them as a script. From SQL*Plus, `SYS` can run:

```
@C:\app\oracle\product\11.2.0\dbhome 1\RDBMS\ADMIN\utlmail.sql

@C:\app\oracle\product\11.2.0\dbhome 1\RDBMS\ADMIN\prvtmail.plb
```

■ **Note** You can find these commands listed in the file named *Chapter9/Sys.sql*.

Granting Access to UTL_MAIL

We are only going to allow one Oracle user to access the UTL_MAIL package: the appsec user. We will include code in the app_sec_pkg package to send out 2-factor pass codes using UTL_MAIL. First, let's enable our Security Administrator, secadm user to make entries in the database Access Control Lists (ACLs). Have SYS make the grants shown in Listing 9-1, including grant execute on the UTL_MAIL package to the appsec_role.

Listing 9-1. *Grant Access to UTL_MAIL, as SYS user*

```
GRANT EXECUTE ON sys.dbms network acl admin TO secadm role;
GRANT EXECUTE ON sys.utl mail TO appsec role;
```

Then as secadm user, execute the commands in Listing 9-2 to establish access control list (ACL) entries that will permit appsec to open port 25 and send e-mail. Be sure to edit the second command in the listing, inserting the name of your corporate SMTP server into the "host" field.

Listing 9-2. *ACL Entries to Send E-Mail, as Secadm User*

```
BEGIN
  DBMS NETWORK ACL ADMIN.CREATE ACL (

    acl          => 'smtp acl file.xml',
    description  => 'Using SMTP server',
    principal    => 'APPSEC',
    is grant     => TRUE,
    privilege    => 'connect',
    start date   => SYSTIMESTAMP,
    end date     => NULL);

  COMMIT;

END;
/

BEGIN
  DBMS NETWORK ACL ADMIN.ASSIGN ACL (

    acl         => 'smtp acl file.xml',
    host        => 'smtp.org.com',
    lower port  => 25,
    upper port  => NULL);
  COMMIT;

END;
/
```

■ **Note** You can find a script of the commands in Listing 9-2 in the file named *Chapter9/SecAdm.sql.*

You will notice that this syntax is quite different from what we're used to when calling procedures. These are, after all, simply calls to the `CREATE ACL` and `ASSIGN ACL` procedures in the `DBMS NETWORK ACL ADMIN` package that is installed by default in the Oracle Database. In Listing 9-2, we are specifying the parameters for these procedures using *named notation*, as opposed to our usual *positional notation*. (I've given named notation here because every example I've seen has used that notation; we probably all just copy the example given in the Oracle documentation).

In named notation, the value being assigned to each parameter is given after the defined parameter name and the assignment operator `=>`, e.g., `acl => 'smtp acl file.xml'`. When we call a procedure using positional notation, we list the parameters in the order they were defined in the procedure (hence the name *positional*); but in named notation, the order is not important.

Another difference between positional and named notation is in how optional parameters are handled. You will recall that our error logging table, `t appsec errors`, has two optional parameters, `msg txt` and `update ts`. They are optional because we configured them with default values. The default value for `update ts` is `SYSDATE`. In our `p log error` procedure, we never provide a value for `update ts`. We want the default SYSDATE.

Back to the topic at hand: in positional notation, only optional parameters that are defined after all the required parameters may be ignored. With named notation, we can skip providing optional parameters no matter where they occur in the parameters definition, as long as we supply values for all the required parameters by name.

In the two procedures being called in Listing 9-2, `CREATE ACL` and `ASSIGN ACL`, the optional parameters are the last two defined for each procedure, so there is no compelling reason to call these procedures using named notation. In addition, the two calls that I have listed provide a value (or NULL) for each parameter, including the optional ones.

Again as `SYS` user or `DBA`, you can list the ACLs to ensure that your entries were made. Execute these queries:

```
SELECT * FROM sys.dba network acls;
SELECT * FROM sys.dba network acl privileges;
```

The results of the first command will resemble this:

```
"HOST"         "LOWER PORT" "UPPER PORT" "ACL"                          "ACLID"
"smtp.org.com" "25"         "25"         "/sys/acls/smtp acl file.xml" "004B...
```

The results of the second command will resemble this:

```
"ACL"                          "ACLID"  "PRINCIPAL" "PRIVILEGE" "IS GRANT" ...
"/sys/acls/smtp acl file.xml" "004B... "APPSEC"    "connect"   "true"     ...
```

Testing Sending E-Mail

While we are here, we should execute a test to be sure we can send e-mail. We are going to execute the test as our Application Security, `appsec` user. I apologize for bouncing around between these accounts, but we are delegating tasks and checking our work at each step.

■ **Note** You can find a script of the commands to follow in this section in the file named *Chapter9/AppSec.sql*.

We have already configured the Network ACL for appsec to be able to open port 25 on your SMTP mail host, but for each session we also need to tell the UTL MAIL package to use that server as our SMTP host. As appsec, we add a property to our Oracle session that the UTL MAIL package will read:

```
ALTER SESSION SET SMTP OUT SERVER = 'smtp.org.com';
```

Then we send an e-mail message. The arguments we provide are our e-mail address, address of the recipient, title of the message, and text of the message.

```
CALL UTL MAIL.SEND( 'myname@org.com', 'myname@org.com', '', '',
    'Response','2FactorCode' );
```

Getting Oracle Database to Browse Web Pages

Besides sending two-factor authentication pass codes by e-mail/SMS to cell phones (and possibly e-mail accounts), we are going to send the pass codes to pagers. At our company, we send text messages to corporate pagers from a web page interface. This may or may not be the approach you will need to take to distribute two-factor authentication pass codes to pagers; however, it is relevant to any message distribution, because e-mail and web services are the primary modes of text messaging distribution from a user application.

Delegating Java Policy to Security Administrator

We already saw how we can add an ACL to allow a user to open a port. Now, in order to open a port as a Java stored procedure, we will need to grant Java security permissions. We are, in effect, allowing Java to perform an activity that is normally denied by the Oracle JVM security sandbox.

First, we will have SYS or a DBA delegate the policy permission to manage specific Java sandbox privileges to our security administrator, the secadm user. As SYS, do this with the code in Listing 9-3.

Listing 9-3. *Grant Policy for Security Administrator to Grant Socket Permissions*

```
CALL DBMS JAVA.GRANT POLICY PERMISSION(
    'SECADM ROLE', 'SYS',
    'java.net.SocketPermission',
    '*');

COMMIT;
```

The DBMS JAVA.GRANT POLICY PERMISSION command specifies secadm role as the recipient of the permission. SYS is the schema in which the grant is effective. The kind of permission being granted is a SocketPermission. And secadm role can thereby manage any socket (*).

Assure that the Java policy permission has been granted with the following command. The policies are granted to a grantee number, GRANTEE#. We look up the name of that user in the USER$ table where user number, USER# matches the grantee number.

```
SELECT u.user#, u.name, p.name, p.type name, p.action
FROM sys.user$ u, sys.java$policy$ p
WHERE p.name LIKE '%java.net.SocketPermission%'
AND p.grantee# = u.user#;
```

The results of this query will resemble the following. `JAVA ADMIN` has a grant to `SocketPermission` on installation.

```
"USER#" "NAME"        "NAME 1"                         "TYPE NAME"
"40"    "JAVA ADMIN"  "0:java.net.SocketPermission#*" "oracle.aurora.rdbms...

"93"    "SECADM ROLE" "0:java.net.SocketPermission#*" "oracle.aurora.rdbms...
```

We are delegating a limited permission to manage the policy regarding opening sockets (network ports) to the `secadm role`. There are other policies we might grant, like the policy regarding opening files on the Oracle server file system, but we don't need that here.

Permitting Application Security User to Read Web Pages

Now, as the `secadm` user, let's grant permission for our application security, the `appsec` user, to actually open a port to the web server that sends text messages to our corporate pagers. Change the name and port number of your web server, as required, then execute the code in Listing 9-4 as `secadm` user.

Listing 9-4. *Grant Socket Permission to Application Security User*

```
CALL DBMS JAVA.GRANT PERMISSION(
    'APPSEC',
    'java.net.SocketPermission',

    'www.org.com:80',
    'connect, resolve'
);
```

By this Java permission grant, we are heavily restricting what can actually be done.

- We will only permit connections to a specific server at a specific port, e.g. *www.org.com:80*.
- We will only permit one user to open the connection, `appsec`.
- And we will only allow the "connect" and "resolve" actions, which are sufficient to read a web page (and via the `GET` method, submit data in the URL). We need `resolve` action so that we (in the Oracle database) can do a DNS lookup/name resolution on (for example) `www.org.com`, to find the IP Address. We need the `connect` action so that we can actually establish a connection on a network port. Those are actions that are not permitted by default from the Oracle JVM security sandbox.

This call to `GRANT PERMISSION` will likely throw an "uncaught Java exception" error. We cannot fix that, and it's nothing to worry about. Perhaps Oracle doesn't expect us to call procedures in the `DBMS JAVA` package from the SQL command line.

Testing our ability to read web pages from Java on the Oracle database will require that we configure a Java stored procedure and update our Java code. Let's wait and test the fully functional code after we write it.

The Two-Factor Authentication Process

We are going to have Oracle applications attempting to read sensitive data that we are protecting. And we are going to require that these applications do database authentication first; they will need to pass our single sign-on tests, which were described in Chapter 8. Then they will need to request and receive a two-factor pass code that we will send to them from Oracle database. It is that process of requesting and receiving the two-factor pass code that we will be implementing in this chapter.

Once the user receives the two-factor pass code, they will submit that along with their request for data (and do the encryption key exchanges that we finished discussing in Chapter 7). At that point, if the user clicks his heels together and says "There's no place like home", he will be back in Kansas with access to read and update the data, as authorized.

Here's the catch: we can't assume that this two-factor code exchange will be instantaneous; otherwise, we would just require that the two-factor code exchange and the data requests occur in the same session. On the contrary, we will need to cache the two-factor codes in Oracle database for a set time limit to assure that the user and code go together. Certain other facts about the user need to correlate as well, such as the address of the computer from which the requests are being made.

Can you picture what we need to make all this happen? First of all, we need some data about our application users. We need their corporate pager numbers, cell phone numbers, and corporate e-mail addresses. The e-mail address is already stored in the `HR.EMPLOYEES` table. We will build another table in the `HR` schema to hold cell phone and pager numbers.

Along with the cell phone number, we will want to designate a carrier. Each carrier (e.g., AT&T) has a different address to which we will send messages for the cell phone. In the case of AT&T cell phones, we will send SMS messages to `10-digit-phone-number@txt.att.net`. There are SMS aggregators who will deliver SMS messages to phones handled by a number of carriers. If you have paid access to an aggregator, then you can use that as the carrier for all phones.

Security Considerations for Two-Factor Distribution Avenues

Two-factor authentication messages can be delivered to our application users via a variety of devices. We are going to consider delivering the messages by cell phone, pager, and e-mail. With each of those devices we have to consider the security implications. Our preference will be to send the two-factor code to a cell phone or a pager. Only if those fail will we want to send the code to an e-mail address.

Security Issues with Two-Factor Delivery to E-Mail

E-mail in and of itself is a fairly secure application. It is password protected and it is usually well managed. However, the data is usually not protected in transit by encryption, and it is extremely easy to impersonate an e-mail sender and send mail as them. There is no inherent problem with exchanging non-sensitive data using e-mail as long as you are willing to filter out the junk and verify the sender by an alternate means, as needed.

■ **Note** I love e-mail because it lets us communicate in a non-concurrent, non-collocated, and person-to-person or broadcast fashion. It also can serve as a tactical record of a dialog and as a communication archive.

However, the whole idea with two-factor authentication is that we require a user to have two different and separate identifying traits. Yes, e-mail and our Oracle application are different pieces of code, but they may be running on the same computer, so they may not be separate.

If a hacker breaks into my computer and my e-mail is running, or she also breaks into my e-mail on my computer, and then if I send two-factor codes to e-mail, she can also run the Oracle application as me; even though the Oracle application is supposed to be protected by two-factor authentication.

Security Issues with Two-Factor Delivery to Pagers

I cannot speak regarding all pagers because some may operate differently, more like cell phones; however, the pagers I'm familiar with are quite simple devices, like AM radios. Two-way pagers generally use paging broadcast signals to deliver pages, but responses are returned through cellular telephone signals. Do not assume that page messages are individually delivered to your specific pager. That is not how the system works.

Page messages are delivered to a radio tower and broadcast, sometimes in data blocks of many messages. Each message is prefixed with a code. Your pager and all pagers in your system, listen to everything being broadcast from the radio antenna. The pager is programmed with a code or a list of codes to listen for. If the pager "sees" a code that it recognizes, it displays the associated message on the screen. If your pager is turned off, or the physics of the radio waves are not quite right (for instance, you're in a basement full of pipes), then you miss the broadcast and there's no way for your pager to retrieve the message later.

Now, picture the eaves-dropper sitting in a nearby hotel room with a scanner and a printer. He is listening to the radio frequency that the pager antenna is emitting, and he is printing out messages that are going to target codes, or he is printing out (or saving to a file) all messages. Also picture the hacker with his own radio and antenna, around the corner, sending his own devious page messages to target pager codes.

Now that you know the security issues, I will hasten to say that pagers are a great way to send immediate, simple, non-sensitive messages to people who may be traveling or away from a computer and phone. Paging is also a great way to broadcast messages to a number of people (via a paging group) simultaneously. Just don't send anything you don't want the world to see. And if you have doubts about the authenticity of a message, check with the sender. Also if you are a sender, don't assume the recipient got the page.

Pagers are probably a technology that will quickly disappear now that cell phone texting has become so popular. Pagers have the advantage of being inexpensive to buy and have inexpensive service plans. They are also allowed into some secure facilities where cell phones are not permitted, so pagers may be required in order to contact people in those facilities who are not at a landline phone or computer.

Security Issues with Two-Factor Delivery to Cell Phones

Unlike pagers, cell phones messages are delivered from a specific antenna to a specific cell phone. As you pass from the area covered by one antenna (cell tower) to another, your communication is handed off to the other cell.

Generally, this communication is fairly secure, with what passes as encryption. The key to reading and sending data is contained in the subscriber identity module (SIM) card in your phone. We have all seen television shows and movies depicting how a SIM card can be cloned, and an imposter cell phone can eaves drop on a call to the original phone. I don't think that happens much in real life, but be aware of the possibility.

Once my son broke the keypad on his cell phone and got another one with the same phone number. I suppose that they cloned his SIM card in the new phone. For a while, both phones received all the texting messages that were sent to him, so I can assure you that cloning (cell phones) works.

I like to remind my kids that the text messages they send, even if addressed only to a single other cell phone, are going into a system that can be read by many people that they don't even know: the service provider technical staff and law enforcement who may request access. That's not even mentioning the parents, siblings, friends, and other snoops or thieves who might browse the phone at some time.

We need to remember that any data (including voice, photograph, and video) which is made into a signal should be considered publicly accessible. That includes signals sent by plain old telephone systems (POTS), cell phones, the Internet, and so on. That includes electrical, radio wave, light, and sound transmissions.

Preferred Two-Factor Delivery

Cell phones are the most secure avenue for delivering our two-factor pass codes. However, there is nothing sensitive about the pass codes, so pretty much any delivery route is fine. If a code is read by an eavesdropper, to use the code that person would have to be sitting at the computer that initiated the Oracle application, logged in as the initiating user and within the code cache timeout period (10 minutes). If someone sends a fraudulent code, that is not likely to have an effect other than frustrating the user who is trying to use it for two-factor authentication.

The only preference we have is that if cell phone or pager delivery is available, we do not want to send the code to e-mail. Because e-mail may be running on the same computer as the Oracle application, sending the code to e-mail does not meet our goals for two-factor authentication, which is separate and distinct identity traits. Specifically, the two factors we are counting on are what you know (SSO initial login password), and What you have (a separate cell phone or pager).

Oracle Structures Supporting Two-Factor Authentication

In order to accomplish two-factor authentication, we will create some new tables and incorporate them into our security processes. We will create a table to hold cell phone and pager numbers for users. Additionally, we will create a table to hold the SMS gateway address for each cell phone carrier whose phones we need to support. During our 10-minute cache timeout period, while we wait for the 2-factor codes to be delivered to the user's cell phone, pager or e-mail, we will store the two-factor codes in the Oracle database, in another new table that we will create.

In addition to new tables, we will also create functions to send and to test the two-factor codes. We will also modify the `p check hrview access` procedure to accept and test the two-factor code before setting the secure application role, `hrview role`.

Creating the SMS Carrier Host Table

Most, if not all, cell phone service providers (carriers) offer access to text their phones through an e-mail server (SMTP.) For example, if you have a personal cell phone that is provided by AT&T, you can send an e-mail message to your phone number at the AT&T SMTP server (e.g., 8005551212@txt.att.net), and they will route your text message to the phone as an SMS message.

Each carrier has its own SMTP to SMS gateway and address; for example, AT&T has txt.att.net. We need a table to store those addresses so that we can send our two-factor authentication codes to cell phones provided by a variety of carriers. We will create a table, `sms carrier host`, to hold those addresses. See Listing 9-5. We also create a unique index based on `sms carrier cd` that we designate as our primary key. And we create a view (`v sms carrier host`) of the table. Create this table in the HR schema.

■ **Note** You can find the commands in Listing 9-5 in the file named *Chapter9/HR.sql*.

Listing 9-5. *Create an SMS Carrier Host Table*

```
CREATE TABLE hr.sms carrier host
(
    sms carrier cd  VARCHAR2(32 BYTE) NOT NULL,
    sms carrier url VARCHAR2(256 BYTE)
);

CREATE UNIQUE INDEX sms carrier host cd pk ON hr.sms carrier host
    (sms carrier cd);


ALTER TABLE hr.sms carrier host ADD (
    CONSTRAINT sms carrier host cd pk
    PRIMARY KEY
    (sms carrier cd)
    USING INDEX sms carrier host cd pk
);

CREATE OR REPLACE VIEW hr.v sms carrier host AS SELECT * FROM hr.sms carrier host;
```

After creating the table, you will want to pre-populate it with the addresses of the major carriers, and any other carriers that are popular with your workforce. Be aware that some of the smaller cell-phone providers piggyback on the major carrier's systems, especially phones from providers that are offered strictly as pre-pay phones.

You can find comprehensive lists of provider's SMS gateways on the Internet. One fairly comprehensive list can be found at `http://en.wikipedia.org/wiki/List of SMS gateways`. I do not want to promote any specific carrier, but offer these example addresses (URLs) in Listing 9-6 that you may want to insert for your use.

Listing 9-6. *Insert Sample SMS Carrier Host Entries*

```
INSERT INTO hr.sms carrier host
    ( sms carrier cd, sms carrier url ) VALUES
    ( 'Alltel', 'message.alltel.com' );

INSERT INTO hr.sms carrier host
    ( sms carrier cd, sms carrier url ) VALUES
    ( 'AT T', 'txt.att.net' );
```

```
INSERT INTO hr.sms carrier host
    ( sms carrier cd, sms carrier url ) VALUES
    ( 'Sprint', 'messaging.sprintpcs.com' );

INSERT INTO hr.sms carrier host
    ( sms carrier cd, sms carrier url ) VALUES
    ( 'Verizon', 'vtext.com' );
```

Creating a Table of Employee Mobile Numbers

We already have a table in the HR schema that includes the e-mail address for personnel: the EMPLOYEES table. However, we also need to store pager numbers and cell-phone numbers for employees, so we will create a table to hold those data values, the emp mobile nos table. We need one more data value to tie all this together: when a user log's in, and we identify her through our SSO process, we need a way to associate the HR EMPLOYEES and emp mobile nos data with the user. We will add a column named user id to the emp mobile nos table to make that association. See Listing 9-7 for the creation command.

The HR.EMPLOYEES table includes a primary key index on EMPLOYEE ID, which is an independent numeric value: each employee that is hired is assigned the next numeric value, sequentially. Whatever that value is, we will use to assign mobile numbers to that same employee. We are further associating that numeric EMPLOYEE ID with an SSO logged-in user, the user id column.

Listing 9-7. *Create a Table for Employee Mobile Phone Numbers*

```
-- Adjust length of Pager No and Phone No as needed
CREATE TABLE hr.emp mobile nos
(
    employee id     NUMBER (6) NOT NULL,
    user id         VARCHAR2(20 BYTE) NOT NULL,
    com pager no    VARCHAR2(32 BYTE),
    sms phone no    VARCHAR2(32 BYTE),
    sms carrier cd  VARCHAR2(32 BYTE)
);
```

At this point, I will point out an important fact that you need to consider. As we go forward in our security structures and procedures, we become dependent on the security that we established previously. Our two-factor authentication depends on SSO; we need to identify the user who is requesting two-factor authentication in order to send the two-factor code to the correct devices.

Additionally, we need to know what EMPLOYEE ID is associated with a user id. Two-factor authentication can only succeed for employees who have an entry in the emp mobile nos *table*, or an alternative. Consider the possibility that we might have added these columns directly to the HR.EMPLOYEES table. It would have been correct to do so as far as data normalization standards dictate; unless it can be proved that not all employees need access to our applications.

It may seem a bit backward to have the computer user id field in a table of mobile phone numbers instead of in the primary EMPLOYEES table, but we are adding this functionality onto the existing HR structures, and we decided to do it all in one place, the emp mobile nos table.

It is a great idea to enforce some integrity constraints on the data by way of unique indexes. We do this with the commands shown in Listing 9-8. We only permit one record in this table per `employee id`. We also only allow a `user id` to be associated with one record, and hence with one `employee id`. Notice in the table definition that neither `employee id` nor `user id` are permitted to be null.

Listing 9-8. *Create Indexes for Employee Mobile Phone Numbers Table*

```
CREATE UNIQUE INDEX emp mob nos emp id pk ON hr.emp mobile nos
    (employee_id);


CREATE UNIQUE INDEX emp mob nos usr id ui ON hr.emp mobile nos
    (user id);


ALTER TABLE hr.emp mobile nos ADD (
    CONSTRAINT emp mob nos emp id pk
    PRIMARY KEY
    (employee id)
    USING INDEX emp mob nos emp id pk,
    CONSTRAINT emp mob nos usr id ui
    UNIQUE (user id)
    USING INDEX emp mob nos usr id ui
);

ALTER TABLE hr.emp mobile nos ADD (
  CONSTRAINT employee id fk
  FOREIGN KEY (employee_id)
  REFERENCES employees (employee_id),
  CONSTRAINT sms carrier cd fk
  FOREIGN KEY (sms carrier cd)
  REFERENCES sms carrier host (sms carrier cd));
```

The unique index on `employee id` is both our primary key on this table, and is a foreign key. Refer to the commands in Listing 9-8. By the foreign key relationship, we are establishing the constraint that in order to create a record in the `emp mobile nos` table for a certain `employee id`, there must already exist a record in the `EMPLOYEES` table with that `EMPLOYEE ID`.

Similarly, we have a foreign key constraint, `sms carrier cd fk` on the table. (Again, see Listing 9-8) That constraint restricts us to writing (insert or update) `sms carrier cds` in the `emp mobile nos` table only if they already exist in the `sms carrier host` table.

You need to be aware that foreign keys are binding on both tables. Once we insert a record with a specific `employee id` in the `emp mobile nos` table, we cannot delete the record for that `EMPLOYEE ID` from the `EMPLOYEES` table until we've already deleted the associated record from `emp mobile nos`. We've encoded the rule that we can't have a record in `emp mobile nos` for an `employee id` unless an associated record exists in `EMPLOYEES`; if we deleted the associated record from `EMPLOYEES`, we would no longer meet that requirement. Oracle database will not let us do that.

We always want to have a view of our table that is our primary reference to the table, so we create one with the code in Listing 9-9. If we are granting and using a view, then we can change the table (or tables) it refers to while maintaining the view, and keep from breaking any code.

Listing 9-9. *Create a View of Employee Mobile Phone Numbers Table*

```
CREATE OR REPLACE VIEW v emp mobile nos AS SELECT * FROM hr.emp mobile nos;

INSERT INTO hr.v emp mobile nos
    ( employee id, user id, com pager no, sms phone no, sms carrier cd )
    VALUES ( 300, 'OSUSER', '12345', '8005551212', 'Verizon' );

COMMIT;
```

Insert mobile numbers for our example user ID, OSUSR, into the table. Note that `employee id` 300 is what we forced when we updated the sequence and inserted our example user in Chapter 7. While we're here, let's also insert a record for you. Use an insert statement like that just given, substituting the `user id` you use to log in (perhaps to Windows) and your pager number, cell phone number, and carrier code (whatever name you will use for all cell phones from the same provider you use). You will also need to insert a record in the `HR.EMPLOYEES` table for you (first name, last name, and e-mail address.) Use the commands in Listing 9-10 as templates.

Listing 9-10. *Template Commands to Create Employees and Add Their Mobile Numbers*

```
INSERT INTO hr.employees
    (employee id, first name, last name, email, phone number, hire date,
    job id, salary, commission pct, manager id, department id)
VALUES
    (hr.employees seq.NEXTVAL, 'First', 'Last', 'EMAddress',
    '800.555.1212', SYSDATE, 'SA REP', 5000, 0.20, 147, 80);


INSERT INTO hr.v emp mobile nos
    ( employee id, user id, com pager no, sms phone no, sms carrier cd )
    VALUES ( (
        select employee id from hr.employees where
        first name = 'First' and last name = 'Last'
    ), 'UserID', '12345', '8005551212', 'Verizon' );

COMMIT;
```

Be sure to `COMMIT` the inserts and updates you've made in order to make them visible to other sessions and other users.

Accessing HR Tables from Application Security Procedures

We are going to continue to have our application security user, appsec, run all the security procedures. In order to accomplish our two-factor authentication, she will need to read the EMPLOYEES table from HR, as well as the tables that we just created. We grant her access as shown in Listing 9-11.

Listing 9-11. *Grant Application Security User Access to See Our Views*

```
GRANT SELECT ON hr.v employees public TO appsec;
GRANT SELECT ON hr.v sms carrier host TO appsec;

GRANT SELECT ON hr.v emp mobile nos TO appsec;
```

Create the Two-Factor Codes Cache Table

Now that we have defined the tables in HR that hold the addresses and numbers where we are going to send our two-factor authentication codes, we need to consider how we will abide the interim between sending the code and having the user enter it in our application. It may require several minutes for the two-factor authentication code to traverse the Internet and the cell phone system to the user's phone. We do not want to keep our connections open to Oracle database for the whole duration of this out-of-band communication, so we need to think about how to store the two-factor codes for later comparison, and how to assure that the same user to whom we issued and sent the two-factor code is entering it.

With our SSO process and the mapping between login users (user id) and employee ids (in the emp mobile nos table), we can save a two-factor authentication code for the specific user who logged in. At this point, we will allow one two-factor code per user (although we will modify that in the next Chapter), so we index by employee id.

We create a table to cache (retain) two-factor authentication codes for a time. This is shown in Listing 9-12. We are creating this in the appsec schema, so connect to Oracle database as appsec, if not already, and execute SET ROLE appsec role to get the Application Security role in order to accomplish this step.

Listing 9-12. *Create a Table to Cache Two-Factor Authentication Codes*

```
CREATE TABLE appsec.t two fact cd cache
(
    employee id   NUMBER(6) NOT NULL,

    two factor cd VARCHAR2(24 BYTE),

    ip address    VARCHAR2(45 BYTE) DEFAULT SYS CONTEXT( 'USERENV', 'IP ADDRESS' ),

    distrib cd    NUMBER(2),
    cache ts      DATE DEFAULT SYSDATE

);
```

Now this is pure conjecture and creative imagination, but I believe that codes in the following format will be sufficiently complex for security and easy to enter. A series of 12 random numeric characters in groups of 4four separated by dashes, e.g., 1234-5678-9012. Perhaps you disagree and believe another format is better, and that is okay—you have the PL/SQL code and can change it as desired. Just note that this table is being created with a maximum two factor cd length of 24 characters.

Notice also that we are sizing a column for `ip address` to be 45 characters. I may have over-specified it a bit, but while an IPv4 address is limited to 15 characters, an IPv6 address can be up to 39. Furthermore, an IPv4 address, mapped to IPv6 can be represented by up to 45 characters.

We also have a field in the `t two factor cd cache` table for the timestamp, `cache ts`. It is set to default to `SYSDATE`, but that only has effect when we insert. When we update, we will "manually" set it to `SYSDATE`. Why do we need a cache timestamp? Here is some more creative imagination: you will find in the code that we consider a two-factor authentication code to be good for 10 minutes. We will not send another two-factor authentication code to the same user within that time period, and after 10 minutes, the code we sent is no longer valid.

The last column that we need to explore in this table, `distrib cd`, is a numeric value that indicates how the two-factor code was distributed. Table 9-1 shows the potential values.

Table 9-1. *Two-Factor Code Distribution Values*

Value	Distribution
0	Not Distributed
1	To Pager
2	To Cell Phone
3	To Pager and Cell Phone
4	To E-Mail
5	To Pager and E-Mail
6	To Cell Phone and E-Mail
7	To Pager, Cell Phone and E-Mail

I won't show it here, but on this table the `employee id` is both a unique index and the primary key. We also create the view, `v two fact cd cache`. Code for creating the indexes can be found in the file, *Chapter9/AppSec.sql*.

Testing Cache Aging

Let's insert a record and play with the aging algorithm we are going to use.

```
INSERT INTO appsec.v two fact cd cache
( employee id ,two factor cd )

VALUES
(300,'FAKE');
```

Now, select the record to see that the timestamp is set.

```
SELECT * FROM appsec.v two fact cd cache;
```

Execute the following command a couple times. It will show the time elapsed since the timestamp was set, and it should count up as you run it.

```
SELECT (SYSDATE-cache ts)*24*60
FROM appsec.v two fact cd cache WHERE employee id=300;
```

That command uses date arithmetic like what we've seen before. We subtract the `cache ts` from `SYSDATE`. Normally what you see when adding and subtracting from `SYSDATE` is some number of days. Always behind the scene is more precision, which is expressed in fractions of days. In a 10-minute span, there is only a small fraction of a day. We multiply it by 24 to get the fraction of an hour. And we multiply it by 60 to get the number of minutes.

Verifying Current Cached Two-Factor Pass Code

A specific user (determined by SSO) will hand a two-factor authentication code to this procedure. What questions do we need to ask about the code to determine if it is acceptable? First we ask if there is an existing code for the user. Second we ask if the existing code was requested from the same address that the current user is using. And third we ask if the existing code is less than 10 minutes old. We ask all those questions in the `SELECT` query that we have in the function `f is cur cached cd`, shown in Listing 9-13. If a code that meets those requirements exists, it is returned in the `cached two factor cd` variable.

Listing 9-13. *Test User-Entered Two-Factor Code Against Cached Version, `f is cur cached cd`*

```
CREATE OR REPLACE FUNCTION appsec.f is cur cached cd( just os user VARCHAR2,
    two factor cd t two fact cd cache.two factor cd%TYPE )

RETURN VARCHAR2

AS
    cache timeout mins NUMBER := 10;
    return_char VARCHAR2(1) := 'N';

    cached two factor cd v two fact cd cache.two factor cd%TYPE;

BEGIN
    SELECT c.two factor cd INTO cached_two_factor_cd

    FROM v two fact cd cache c, hr.v emp mobile nos m

    WHERE m.employee id = c.employee id

    AND m.user id = just os user

    AND c.ip_address = SYS CONTEXT( 'USERENV', 'IP ADDRESS' )

    AND ( SYSDATE - c.cache ts )*24*60 < cache_timeout_mins;

    IF cached_two_factor_cd = two_factor_cd
    THEN

        return_char := 'Y';
    END IF;
    RETURN return char;
END f is cur cached cd;

/
```

The last thing we do is compare the `cached two factor cd` we found to the `two factor cd` that the user handed us. If they are equal, we return a "Y" (equivalent to a `boolean true`, but easier.) Otherwise, we return the default "N".

Notice that this is one of two places where you need to set a different cache duration, if 10 minutes is not correct for your application. The other place is in the `OracleJavaSecure.distribute2Factor()` method.

As with other procedures and functions defined in this chapter, we are not placing them in an Oracle package. Because of the different grants (none or `PUBLIC`) to these structures, we would need different packages with only one or two procedures or functions each. In the next chapter, we will add many similar procedures and functions and we will organize them into packages at that time.

Sending Two-Factor Pass Codes

If we have determined that we need to send two-factor codes to a user, we will call the function `f send 2 factor` on Oracle database. It is a Java stored procedure that calls Java code on the Oracle Database to do what we cannot (easily) do in PL/SQL.

In this case, we call the `distribute2Factor()` method (see Listing 9-14), which generates the two-factor code, attempts to send it to pagers and cell-phones, and stores the code in the cache table.

Listing 9-14. *Send the Two-Factor Code to the User, f_send_2_factor*

```
CREATE OR REPLACE FUNCTION appsec.f send 2 factor( just os user VARCHAR2 )
RETURN VARCHAR2

AS LANGUAGE JAVA

NAME 'orajavsec.OracleJavaSecure.distribute2Factor( java.lang.String ) return
java.lang.String';

/
```

Updating the Secure Application Role, HRVIEW_ROLE Procedure

Recall our secure application role, `hrview role`. Without it we cannot read the data that we've deemed sensitive in the `HR` schema. We originally created the procedure, `p check hrview access`, in Chapter 2 to provide access to the secure application role. At that time, we just checked IP Address and time of day.

In the last chapter, we updated the procedure to do SSO. If the user connection passed our SSO requirements, we set the role; otherwise, not.

Now in this chapter, we are adding another test – has the user passed a valid two-factor authentication code? See Listing 9-15. In order to accommodate that test, we have to modify the procedure to take an argument: the two-factor code that the user has entered. We also return error codes, as we've done in other procedures. In this case, we will return our distribution code (remember 1 if by pager?) in the `err txt` field if there is no actual error (no `err no`).

Listing 9-15. *Secure Application Role Procedure Header, p_check_hrview_access*

```
CREATE OR REPLACE PROCEDURE appsec.p check hrview access(
    two_factor_cd t two fact cd cache.two factor cd%TYPE,

    m err no  OUT NUMBER,
    m err txt OUT VARCHAR2 )
```

```
AUTHID CURRENT USER
AS
    just os user    VARCHAR2(40);
    backslash place NUMBER;
BEGIN
```

I'm leaving out the bulk of the body of this procedure here, but I wanted to point out the section that is specifically used for two-factor authentication, shown in Listing 9-16. If we have passed SSO and other connection tests, then we enter this code. If the user did not pass us a two-factor authentication code, then we call the `f send 2 factor` function. Otherwise, we test the two-factor code to see if it passes muster by calling the `f is cur cached cd` function. If the two-factor code is good, we set the secure application role; however, if not we let them know by raising a `NO DATA FOUND` exception: they entered the wrong code, or perhaps just an old (older than 10 minutes) code.

Listing 9-16. *Secure Application Role Procedure Body, p_check_hrview_access*

```
    THEN
        IF( two factor cd IS NULL OR two factor cd = '' )
        THEN
            m_err_txt := f_send_2_factor( just os user );
        ELSIF( f_is_cur_cached_cd( just os user, two factor cd ) = 'Y' )
        THEN
            EXECUTE IMMEDIATE 'SET ROLE hrview role';
        ELSE
            -- Wrong or Old 2 factor code. Could return message in M ERR TXT,
            -- or this will get their attention.
            RAISE NO_DATA_FOUND;
        END IF;
    END IF;
```

Notice that when we call `f send 2 factor`, we set `m err txt` to the return value. That is so we can pass the distribution code (summary of what devices the two-factor code is delivered to) coming out of `f send 2 factor` back to the client.

Update OracleJavaSecurity.java for Two-Factor Authentication

There are several updates and additions we will make to *OracleJavaSecure.java* to support two-factor authentication. The biggest addition will be a new method to distribute the two-factor authentication codes, `distribute2Factor()`. We will discuss that method at length.

We see how to set some static member variables to hold the specific addressing data we intend to use. We also explore the individual methods that send two-factor codes to SMS devices, pagers, and e-mail.

Setting Some Company-Specific Addresses

Several settings in *OracleJavaSecure.java* are specific to your corporate implementation of two-factor authentication: the DNS domain name within your organization, the host name that handles mail routing for your company, and perhaps a web application URL from which text pager messages can be