In order to put that role to work, we will categorize a certain set of the appver functions and procedures as being for use only by appver admin. We place those functions and procedures in a package, appsec admin pkg, and grant execute on the package only to appver admin. This command can be run as either SYS or appsec user.

```
GRANT EXECUTE ON appsec.appsec admin pkg TO appver admin;
```

We have one existing function, f set decrypt conns that we will move from appsec public pkg to this new package. No longer will just any application user be able to insert or update lists of connection strings for an application. Only users with the appver admin role will.

Delete Connection Strings

Deleting a connection string from the list is a task we accomplish in the local client application. We simply formulate the key of the connection string value that we plan to remove, and then remove the entry from connsHash. Use the removeAppConnString() method, shown in Listing 11-26.

Listing 11-26. *Remove Connection String from List, removeAppConnString()*

```
private static void removeAppConnString()( String instance, String user ) {
    instance = instance.trim();
    user = user.trim();
    String key = (instance + "/" + user).toUpperCase();
    connsHash.remove( key );
}
```

In order to make the change permanent for all future users of the application, we need to save the list to the Oracle Database. This is done by calling putAppConnections(), which we studied in Chapter 10 (see Listing 10-24).

Copy Connection Strings from Previous Version of Application

When the developer releases a new version of her application, she will want to use most of the same connection strings as the old version. If she kept her inner class version number/name the same, then she wouldn't need to worry about replicating connection strings; she would just continue using the existing list. However, it may be that she will want a new version of her inner class, and hence a new list of connection strings, for one of the following reasons:

- To modify the list of connection strings for the new version.
- To eventually disable the old version of her application by deleting the list of connection strings associated with it.

If her application only used a single Oracle connection or possibly two, it would be no problem rebuilding the list from scratch for the new version. However, if the application has many potential connections from which it may draw data, then the ability to replicate connection strings from one version of her application to the next will be a welcome feature.

Application Client Call to Copy List of Connection Strings

From the client application, we will call a new method in `OracleJavaSecure`, `copyAppConnections()`. It takes one parameter, the old inner class version name. When the current (new) inner class is evaluated on the Oracle database, we will be able to determine the new version right from the inner class.

We model `copyAppConnections()` on a method we have already examined, `putAppConnections()`. Our new method calls the Oracle procedure, `appsec admin pkg.p copy app conns`, passing the application inner class and the old version name. In this case, we do not pass the `connsHash` list of connection strings—it is obviously not needed.

It will be a rare event when we call our procedure, `p copy app conns`, to copy an application's connection strings list from one version of the inner class to the next version. We wouldn't create a new version of our inner class each time we update the application; only when we want to retire a previous version of the application, when our changes make a previous version of an application either dysfunctional or unacceptable.

We call `p copy app conns` with an instance of our new version of the application inner class. We also designate the old version name so we know from where to get our connection strings. This procedure, in Listing 11-27, bears strong resemblance to various procedures we've examined before. We assure SSO and 2-factor authentication, before moving on.

If this connection is acceptable, then we call the function, `f copy conns`, which accomplishes the duplication of the list of connection, strings to the new version.

Listing 11-27. *Copy List of Connection Strings to New Version, p_copy_app_conns*

```
PROCEDURE p copy app conns(
    m two factor cd      v two fact cd cache.two factor cd%TYPE,
    m class instance     v app conn registry.class instance%TYPE,
    m_prev_version       v app conn registry.class version%TYPE,
    m application id     v two fact cd cache.application id%TYPE,
    m err no         OUT NUMBER,
    m err txt        OUT VARCHAR2 )
IS
    return user VARCHAR2(40);
    m app user  v application registry.app user%TYPE := 'APPVER';
BEGIN
    m err no := 0;
    return user := f_is_sso( m app user );
    IF( return user IS NOT NULL )
    THEN
        IF( m two factor cd IS NULL )
        THEN
            m err txt := appsec only pkg.f_send_2_factor(return user, m application id);
        ELSIF( appsec only pkg.f_is_cur_cached_cd( return user, m application id,
            m two factor cd ) = 'Y' )
        THEN
            -- Reuse existing VARCHAR2, RETURN USER
            return user :=appsec only pkg.f_copy_conns(m class instance,m prev version);
        ELSE
            -- Wrong 2-Factor code entered
            RAISE NO DATA FOUND;
        END IF;
        app sec pkg.p log error( 0, 'Success copying App Conns, ' || return user );
```

```
        ELSE
            app sec pkg.p log error( 0, 'Problem copying App Conns, ' || return user );
        END IF;
    -- Raise Exceptions
    EXCEPTION
        WHEN OTHERS THEN
            m err no := SQLCODE;
            m err txt := SQLERRM;
            app sec pkg.p log error( m err no, m err txt,
                'p copy app conns' );
    END p copy app conns;
```

Java Stored Procedure to Copy Connection Strings

Once again, we call an Oracle stored function, which is really just a wrapper for a method written in Java that will accomplish the task. This function, `f copy conns` in Listing 11-28 calls the `copyPreviousConns()` method.

Listing 11-28. *Java Stored Procedure to Copy Connection Strings, f_copy_conns*

```
    FUNCTION f copy conns( class instance RAW, class version VARCHAR2 )
    RETURN VARCHAR2
    AS LANGUAGE JAVA
    NAME 'orajavsec.OracleJavaSecure.copyPreviousConns( oracle.sql.RAW, java.lang.String )
return java.lang.String';
```

Database Java Method to Copy Connection Strings

`copyPreviousConns()` is perhaps the most complex method we will study; however, it is only complex in managing multiple versions of our application at once, not in the process. We are going to be performing steps that we have accomplished in other methods that we have examined. Here, though, we will go back and forth between the new and old versions of the inner class.

Our first step is to get the current inner class name and version number (shown in Listing 11-29) directly from the inner class, `classInstance RAW` parameter. We do this by getting the bytes of the `RAW` and pushing them through a `ByteArrayInputStream`, then through an `ObjectInputStream`, from which we read an `Object` as the `providedClass` member. From that `Object`, we can read the name into the `className` member. We can also get the `getRevLvl()` method and read the revision level into the `classVersion` member.

Listing 11-29. *Get New Class Version and Name*

```
    byte[] appClassBytes = classInstance.getBytes();
    ByteArrayInputStream bAIS = new ByteArrayInputStream( appClassBytes );
    ObjectInputStream oins =
        new ObjectInputStream( bAIS );
    Object classObject = oins.readObject();
    oins.close();
    Class providedClass = classObject.getClass();
```

```
String className = providedClass.getName();
Method classMethod = providedClass.getMethod( "getRevLvl" );
String classVersion = ( String )classMethod.invoke( classObject );
```

Next, we want to get an idea of how this is going to go. Do we really have anything to copy? We get the list of connection strings using the current inner class name, which we just got from the `Object`, and using the previous version name that was passed here in the `prevVersion` parameter. (See Listing 11-30.) By passing those arguments (1 and 2) to the stored procedure `p get class conns`, we can get back the list of connection strings (OUT parameter 4) for the previous version of our inner class (OUT parameter 3). If we get a null back for our previous inner class, then there is nothing to copy, so we return.

Listing 11-30. *Select Connection Strings for Previous Version*

```
stmt = ( OracleCallableStatement )conn.prepareCall(
    "CALL appsec.appsec only pkg.p_get_class_conns(?,?,?,?)" );
stmt.registerOutParameter( 3, OracleTypes.RAW );
stmt.registerOutParameter( 4, OracleTypes.BLOB );
stmt.setString( 1, className );
stmt.setString( 2, prevVersion );
stmt.setNull(   3, OracleTypes.RAW );
stmt.setNull(   4, OracleTypes.BLOB );
stmt.executeUpdate();
if( null == stmt.getBytes( 3 ) ) return "Nothing to copy";
```

Listing 11-31 shows another application of our new `f unmask` Oracle stored function. Now that we are encrypting the connection strings as we store them on disk, we need to also decrypt them as we read them from storage.

Listing 11-31. *Decrypt Connection Strings from Previous Version*

```
byte[] prevConnsBytes = stmt.getBytes(4);
stmt = ( OracleCallableStatement )conn.prepareCall(
    "{? = call appsec.f_unmask(?,?,?)}" );
stmt.registerOutParameter( 1, OracleTypes.RAW );
stmt.setBytes(  2, prevConnsBytes );
stmt.setString( 3, className );
stmt.setString( 4, prevVersion );
stmt.executeUpdate();

prevConnsBytes = stmt.getBytes(1);
```

We read the decrypted bytes for our list of application connection strings, but we do not need to assemble a `HashMap` object to represent the list; rather, we will be storing the byte array as-is for the new version.

However, before we proceed to copy our list of connection strings to the new version, we will assure we are not overwriting an existing list. We will call the `p count class conns` procedure for the current inner class name and version to see if an entry exists in the database (see Listing 11-32). If none exists, then we are okay to insert; otherwise, we need to examine the list of connection strings that exists for the new version of the application.

Listing 11-32. *Count Connection Strings Stored for Current Version*

```
stmt = ( OracleCallableStatement )conn.prepareCall(
    "CALL appsec.appsec only pkg.p_count_class_conns(?,?,?)" );
stmt.registerOutParameter( 3, OracleTypes.NUMBER );
stmt.setString( 1, className );
stmt.setString( 2, classVersion );
stmt.setInt(    3, 0 );
stmt.executeUpdate();
boolean okToOverwrite = false;
if( stmt.getInt( 3 ) == 0 ) {
    // Do insert!
    okToOverwrite = true;
} else {
```

If an entry exists in the database, we will assure that the inner class instance that we provided as an argument to this method is the same as what is stored. We do this by simply getting an object from the stored bytes based on the current class name and version (see Listing 11-33). Recall that the Java class loader cannot load two substantially different classes by the same name; a runtime exception would be thrown.

Listing 11-33. *Get Connection Strings and Class Stored for Current Version*

```
stmt = ( OracleCallableStatement )conn.prepareCall(
    "CALL appsec.appsec only pkg.p_get_class_conns(?,?,?,?)" );
stmt.registerOutParameter( 3, OracleTypes.RAW );
stmt.registerOutParameter( 4, OracleTypes.BLOB );
stmt.setString( 1, className );
stmt.setString( 2, classVersion );
stmt.setNull(   3, OracleTypes.RAW );
stmt.setNull(   4, OracleTypes.BLOB );
stmt.executeUpdate();

byte[] cachedBytes = stmt.getBytes(3);
oins = new ObjectInputStream( new ByteArrayInputStream(
    cachedBytes ) );
classObject = oins.readObject();
oins.close();
```

When we read the object from the `ObjectInputStream`, it had better be identical to the object we passed as the `classInstance` argument to this method; otherwise an `InvalidClassException` will be thrown. For good measure, we get a class instance from the object and test to see if it is equal to the class we got earlier in Listing 11-34. If it's not, we would have already failed (unless the inner class name is different, in which case it shouldn't be stored in the database with this name).

Listing 11-34. *Test Stored Class and Connection Strings*

```
Class testClass = classObject.getClass();

if( testClass != providedClass )

    return "Failed to setDecryptConns";
```

```
if( null == stmt.getBytes(4) ) okToOverwrite = true;
else {
```

It could be that we stored an entry for this inner class, but stored a `null` for the associated list of connection strings. Listing 11-34 tests this. If the connection strings are `null`, we can overwrite this entry.

It is more likely that an empty `HashMap` was stored for this new inner class version as a placeholder. We can test this by getting the list of connection strings and reading it to see if there are any entries in the list. But first, we revisit our new `f unmask` function to decrypt the connection strings list that we read from storage for the current class name and version, as shown in Listing 11-35.

Listing 11-35. *Decrypt Stored Connection Strings List for Current Version*

```
byte[] connsBytes = stmt.getBytes(4);
stmt = ( OracleCallableStatement )conn.prepareCall(
    "{? = call appsec.f_unmask(?,?,?)}" );
stmt.registerOutParameter( 1, OracleTypes.RAW );
stmt.setBytes(  2, connsBytes );
stmt.setString( 3, className );
stmt.setString( 4, classVersion );
stmt.executeUpdate();
```

Create an object from the decrypted list of connection strings and cast it as a `HashMap`. Next, test the size of the `HashMap`. If the size is zero, we can overwrite this entry; however, if it is not empty, we return without copying the old version connection strings to the new version. See Listing 11-36.

Listing 11-36. *Test if Stored Connection Strings List is Empty*

```
            oins = new ObjectInputStream( new ByteArrayInputStream(
                stmt.getBytes(1) ) );
            Object currentConns = oins.readObject();
            oins.close();
            HashMap<String, String> currConnsHash =
                (HashMap<String, String>)currentConns;
            if( 0 == currConnsHash.size() ) okToOverwrite = true;
        }
    }
    if( ! okToOverwrite ) return "Current connsHash is not empty!";
```

If we have gotten this far, then either there was no entry in `v app conn registry` for the current (new) version of the application inner class, or the associated list of connection strings was null or empty. So we are free to copy the old connection strings to the new version. But first, we will encrypt them as in Listing 11-37, specifically for the new version, by calling our new `f mask` function.

Listing 11-37. *Encrypt Old Connection Strings List for New Version Before Storing*

```
stmt = ( OracleCallableStatement )conn.prepareCall(
    "{? = call appsec.f_mask(?,?,?)}" );
stmt.registerOutParameter( 1, OracleTypes.RAW );
stmt.setBytes(  2, prevConnsBytes );
stmt.setString( 3, className );
stmt.setString( 4, classVersion );
stmt.executeUpdate();
```

```
prevConnsBytes = stmt.getBytes(1);
```

Now we store the encrypted list of connection strings for the new inner class version by calling the `p set class conns` procedure. Listing 11-38 shows that call.

Listing 11-38. *Store Encrypted Connection Strings List for New Version*

```
stmt = ( OracleCallableStatement )conn.prepareCall(
    "CALL appsec.appsec only pkg.p_set_class_conns(?,?,?,?)" );
stmt.setString( 1, className );
stmt.setString( 2, classVersion );
stmt.setBytes(  3, appClassBytes );
stmt.setBytes(  4, prevConnsBytes );
stmt.executeUpdate();
```

Figure 11-1 illustrates the process of copying connection strings from a previous version of an application to the new, current version. The only item in the diagram that I want to mention is the first call to Oracle database, the `p copy app conns` procedure. That procedure is in the `appsec admin pkg` package and can only be execute by users who have been granted the `appver admin` role.

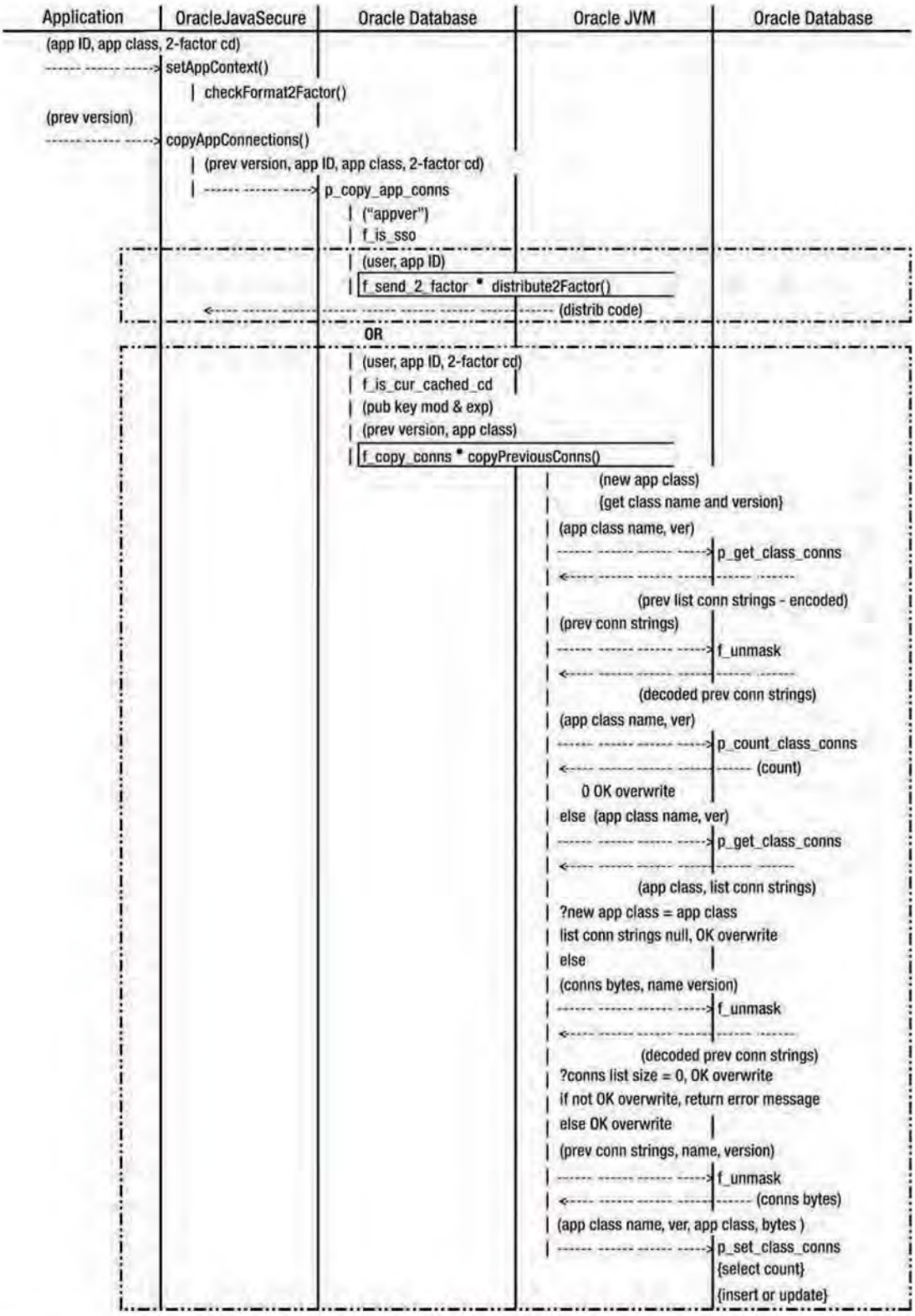


Figure 11-1. *Copying application connection strings to a new version*

Add Other Authentication Credentials

We are not limited to storing only Oracle connection strings in `v app conn registry`. Recall that the `HashMap` is simply a list of string keys and associated string values. Once you get the `HashMap` back to your application, you can request a specific value based on any key you please.

Of course you could store connection strings or at least passwords for connections to non-Oracle databases. You might also store such things as passwords for secure FTP connections. Our current methods in `OracleJavaSecure` are tailored for storing Oracle connection strings, but you could add a method for storing, for example, secure FTP passwords. Listing 11-39 shows a sample method you might employ.

Listing 11-39. *Example Method for Storing Other (FTP) Credentials*

```
public static void putAppFTPString( String key, String password ) {
    appAuthCipherDES.init( Cipher.ENCRYPT MODE,
        appAuthSessionSecretDESKey, appAuthParamSpec );
    byte[] bA = appAuthCipherDES.doFinal(password.getBytes() );
    connsHash.put( "FTP" + key, new RAW( bA ) );
}
```

For security purposes, you would want to devise your method (as shown) to prepend the key with the string "FTP" or something. We would use this as a filter to keep this method from decrypting non-FTP entries in the `connsHash` list. Listing 11-40 provides an example method for getting FTP passwords from `connsHash`.

Listing 11-40. *Example Method for Retrieving Other (FTP) Credentials*

```
private static String getAppFTPString( String key ) {
    return new String(
        appAuthCipherDES.doFinal( connsHash.get( "FTP" + key ).getBytes() ) );
}
```

Notice that this method is designated as `private`—you would want another method in `OracleJavaSecure` that establishes the FTP connection and returns the connection to the client application, rather than returning the clear text FTP password to the application. We do not want to give our passwords to applications.

Update Application Security Structures

Before moving on to new topics, please run all the commands and scripts that we have described thus far. At a SQL*Plus prompt or other SQL client, as `SYS` user, run the commands in *Chapter11/Sys.sql*. Substitute the name of an OS user who will be doing administrative tasks (you?) in the `GRANT appver admin` command.

Then, as the application security, `appsec` user, run the commands in *Chapter11/AppSec.sql*. That should be easy. Additionally, execute the code from *Chapter11/F_MASK.plb* and *Chapter11/F_UNMASK.plb* (the masked versions).

Still as `appsec` user, remove the comment from the first line, `CREATE OR REPLACE AND RESOLVE JAVA` in *Chapter11/orajavsec/OJSC.java* (the obfuscated version) and execute that as SQL code. And finally, uncomment the first line of *Chapter11/orajavsec/OracleJavaSecure.java* and edit the `expectedDomain` and URL strings at the top of the code. Remove the passwords from the `main()` method, at the bottom. Then

execute that as SQL code. Recall, you may have to `SET DEFINE OFF` to keep from being prompted for variable substitution at each & symbol in the code.

Authenticate on a Separate Oracle Instance

I'm going to describe something now that reveals the extent to which we might want to pursue security. What if we segregated the application verification tasks from the actual application data? What security would that buy us? The primary benefits we will achieve are the following:

- Fewer accounts with passwords, so fewer accounts to attack
- Reduced ancillary functions (fewer optional database programs), so reduced vulnerabilities
- Ability to revoke some `PUBLIC` access to particularly revealing data dictionary views without hampering database development
- A first database hurdle for hackers to overcome before sensitive data in the second database can be pursued by attack

In this section, we will create a new database instance, perhaps on the same server. This instance will have sufficient privileges to accomplish application verification, but no more. In order to ensure that, we will not create the privileged roles that we have discussed, `secadm role` and `app sec role`; rather, we will do all our configuration steps as the `SYS` user.

■ **Caution** If you do not intend to create an additional database instance for application verification, then do not issue any of the commands in this section. You can skip down to *Test Enhanced Security*; however, to learn about creating and hardening an Oracle database instance and about database links, read through this section in any case.

What about our application verification, `appver` user password? Is it still susceptible to snooping? Well, it isn't encrypted, and it could be recovered by anyone diligent enough to reconstruct our obfuscated `OJSCode` class. So the question we need to ask is what is our security exposure from having that password revealed.

We might pat ourselves on the back and say that we've got it covered with our login trigger, `t screen appver access`, on the `appver` schema that calls the `p appver logon` stored procedure. And we might smirk and scoff at those who think they can get by the SSO proxy requirements, and down the road our two-factor authentication, encryption key exchange, and application verification process.

However, in the back of our minds, we realize that a hacker having a password to an account on a production Oracle database must be a bad thing. And the back of our minds would be correct. Even if there is nothing else that the account has access to, there is still `PUBLIC` data, and that data can be revealing. From standard `PUBLIC` access, a hacker can learn all the users of the database, setting him up to conduct a social engineering attack. The hacker can also see all the code of any procedures and Java that is granted to `PUBLIC`, and he can see the logic that is contained in triggers and views. `PUBLIC` grants provide an entrée into the database for every user account.

I believe that the Oracle Corporation could make a sweeping change to the database that would improve security. The sweeping change would be to make PUBLIC a regular role. Perhaps the PUBLIC role would be granted by default and additionally not lost when a user does a SET ROLE. However, PUBLIC would not be whatever it is now. Currently, when access is granted to PUBLIC, it is like saying that no grants are required. We cannot revoke PUBLIC from users. Every Oracle user always has PUBLIC access.

I can see how the use of PUBLIC is like making certain things a part of an Oracle user's identity. Maybe what I'm really shooting for is an almost-PUBLIC role that Oracle database could grant as a default and that would not get removed when a user does SET ROLE. At database install, some or most of the things that are normally granted to PUBLIC (anything that is not strictly required for logon and select) could be granted to almost-PUBLIC. Then for restricted users, we could revoke the almost-PUBLIC role.

Have I gone off the deep end? Well let's take a look. First, select from a public view to see all the users in the database:

```
SELECT * FROM ALL USERS;
```

By listing all the database users, an attacker has multiple opportunities to try password guessing, or perhaps finds multiple individuals to contact in an attempt to do a social engineering attack. I'd like to turn off access to that PUBLIC view.

What if we'd like to know the names of all the Oracle procedures that are used by the application security, appsec user? We could query the ALL PROCEDURES PUBLIC view:

```
SELECT * FROM ALL PROCEDURES WHERE OWNER = 'APPSEC';
```

Now let's view the code for one of those procedures, p check role access by querying the ALL SOURCE PUBLIC view:

```
SELECT * FROM ALL SOURCE WHERE OWNER = 'APPSEC' AND NAME = 'P CHECK ROLE ACCESS';
```

Granted, the user can only see the source of procedures that have been granted to PUBLIC, but is that source really something the user needs to see? We shall see that it is not.

Create a New Oracle Database Instance

We need about 2 GB of space on the hard drive to create a sufficiently large database to hold just what we need to do application verification. "Why that much?" you might ask. Remember that to do application verification we need a basic Oracle database, and we need the data dictionary views, and we need to run PL/SQL, and we need to run Java. On top of that, for two-factor authentication, we need to send e-mail, and we need configure system privileges to read data across the network (URLs). All of these functions require space.

We will call our new database instance apver (notice it is like the appver user name, except with only one "P"). We need an initialization/parameter configuration file in order to build our new database instance. If you are creating the apver instance on a server that already has an installed instance, e.g., ORCL, then you can copy some files that will be helpful. One of those files is called *init.ora*. Change directory to where these files reside, outside your server Oracle home directory:

```
D:
cd \app\oracle\admin
```

Copy the entire *orcl* directory to a new directory named *apver*. This command will copy the directory and all the contents.

```
xcopy orcl apver /ei
```

Now change directory to the new parameters file, *pfile* directory and rename the existing *init.ora* file template to *init.ora*. Then edit the *init.ora* file.

```
cd \app\oracle\admin\apver\pfile
ren init.ora.* init.ora
edit init.ora
```

Search and replace the following strings:

```
Replace             With
=======             ====
=orcl               =apver
\orcl               \apver
```

Your final file should have parameters like those in Listing 11-41. Your `db domain` and directory names may be different. The local_listener will the same for apver as for the primary database.

Listing 11-41. *Initialization File for the apver Instance*

```
db block size=8192
open cursors=300
db domain=org.com
db name=apver
control files=("D:\app\oracle\oradata\apver\control01.ctl",
"D:\app\oracle\flash recovery area\apver\control02.ctl")
db recovery file dest=D:\app\oracle\flash recovery area
db recovery file dest size=4039114752
compatible=11.2.0.0.0
diagnostic dest=D:\app\oracle
memory target=1288699904
local listener=LISTENER Orcl
processes=150
audit file dest=D:\app\oracle\admin\apver\adump
audit trail=db
remote login passwordfile=EXCLUSIVE
dispatchers="(PROTOCOL=TCP) (SERVICE=apverXDB)"
undo tablespace=UNDOTBS1
```

We want to copy this *init.ora* file into its default location. This will come in handy later when we import the parameter settings into a system parameter file. Execute the copy command:

```
copy D:\app\oracle\admin\apver\pfile\init.ora %ORACLE HOME%\DATABASE\INITAPVER.ORA
```

Create a directory for the secondary control file:

```
mkdir D:\app\oracle\flash recovery area\apver
```

Also, let's create a directory for our new instance database files:

```
mkdir D:\app\oracle\oradata\apver
```

Create a New Oracle Service

Each Oracle Database instance is started, usually on system reboot, by a Service. You can see these services in Windows by going to your **Start** menu and running the Computer Management application.

You will need to use system administrator privileges. Go to Services and Applications, then to Services and scroll down to the Oracle services. They are all normally named with the prefix "Oracle," and are sorted alphabetically. We are not going to explore creating, starting or stopping processes on Unix or Linux; the steps are the same, but the commands (Run Command level files) are different.

Because we are not using any Oracle web administrative services for the work covered in this book, all the Oracle services can be set to manual; however, do not do that on a production Oracle database server. Then we can manually start the standard `OracleServiceORCL` service to start the ORCL instance. We also start the standard `OracleOraDb11g homeTNSListener` service to start the listener. Those 2 Oracle services are all we need. Typically, clients connect over the network to a listener service that then connects them to the database instance.

For the following discussions, we will assume you have set your `ORACLE HOME` to something like:

```
SET ORACLE HOME=D:\app\oracle\product\11.2.0\dbhome 1
```

Our new Oracle instance will be named `apver`, so we can use a command like the following to add a service to start the instance. You need to be working in a Windows command prompt window with administrative privileges, so right click on Command Prompt in your Start menu and select Run as administrator.

```
%ORACLE HOME%\BIN\oradim -NEW -SID apver -STARTMODE manual
    -PFILE "D:\app\oracle\admin\apver\pfile\init.ora"
```

We also want to set the new service to automatically start the Oracle database process. This is in addition to starting the Service automatically. It is telling the service to issue a `STARTUP` command to the database. We can set the service to `MANUAL` startup later, and when we manually start the service it will automatically start the database.

```
oradim -EDIT -SID apver -STARTMODE AUTO -SRVCSTART SYSTEM
    -PFILE "D:\app\oracle\admin\apver\pfile\init.ora"
```

■ **Note** We will get an error here about not being able to start the service. That is okay, because we haven't really created the database yet, but now the service to get it started is configured.

Write the Create Database Command

We are going to put our database creation command in a script file named `ApVerDBCreate.sql`. CREATE DATABASE is a single command, but there are many aspects to it, and we don't want to depend on our typing skills to enter everything correctly at the SQL*Plus prompt. Also, we want a script file so we can refer to it if we have any questions about the command we issued.

The first thing I'd like you to notice about our database creation script is that we have not hard-coded passwords for `SYS` and `SYSTEM` (we have commented lines in Listing 11-42, meant to remain commented and unchanged). Those users will be created with the default passwords, "change_on_install" and "manager," respectively. Placing the passwords in this command script is typical, but less secure. It is essential that you change these passwords from the default, but ask yourself this question: will you be more likely to issue an `ALTER USER` command once you log in and continue the installation steps, of which that is step one, or will you remember to go back and edit the database creation script, `ApVerDBCreate.sql`, removing the real passwords from that file? We are going to

immediately change the passwords once the database is created, and we will not have to expunge passwords from the script file later on.

Listing 11-42. *Create Database Command*

```
CREATE DATABASE apver
--USER SYS IDENTIFIED BY password
--USER SYSTEM IDENTIFIED BY password
```

■ **Note** This command is contained in a file named *Chapter11/apver/ApVerDBCreate.sql*.

The next few aspects of the database creation command in Listing 11-43 simply define the redo log files we will be maintaining, and their sizes. These log files will be used in case we need to restore the database from backup, and reapply the transactions that have taken place since the backup, also those that were not applied before a database failure. Ideally, these transaction log files would be on a different hard drive from the database files.

Listing 11-43. *Create Database Log Files*

```
LOGFILE GROUP 1 ('D:\app\oracle\oradata\apver\REDO01a.log',
    'D:\app\oracle\oradata\apver\REDO01b.log') SIZE 16M,
GROUP 2 ('D:\app\oracle\oradata\apver\REDO02a.log',
    'D:\app\oracle\oradata\apver\REDO02b.log') SIZE 16M,
GROUP 3 ('D:\app\oracle\oradata\apver\REDO03a.log',
    'D:\app\oracle\oradata\apver\REDO03b.log') SIZE 16M
```

After that, our command includes some basic parameters that may suffice, or can be adjusted later. See Listing 11-44.

Listing 11-44. *Create Database Configuration*

```
MAXINSTANCES 3
MAXLOGFILES 6
MAXLOGMEMBERS 2
MAXLOGHISTORY 1
MAXDATAFILES 10
CHARACTER SET AL32UTF8
NATIONAL CHARACTER SET AL16UTF16
EXTENT MANAGEMENT LOCAL
```

Next, in Listing 11-45, we will define the database files we will use for the `apver` instance. We define our primary System database file, *SYSTEM01.DBF*, and an Auxiliary System file, *SYSAUX01.DBF*, which is used by some database components that had historically been placed in separate *tablespaces*. Additionally, we create the default tablespace files for `USERS`, `TEMPORARY` and `UNDO` tablespaces. Be sure to give the `UNDO` tablespace the same name that you gave it in your *init.ora* file, described previously. And with this, we come to the end of our `CREATE DATABASE` command (note the semicolon).

Listing 11-45. *Create Database Files and Tablespaces*

```
DATAFILE 'D:\app\oracle\oradata\apver\SYSTEM01.DBF' SIZE 512M REUSE
SYSAUX DATAFILE 'D:\app\oracle\oradata\apver\SYSAUX01.DBF' SIZE 512M REUSE
DEFAULT TABLESPACE users DATAFILE 'D:\app\oracle\oradata\apver\USERS01.DBF'
    SIZE 256M REUSE AUTOEXTEND ON MAXSIZE UNLIMITED
DEFAULT TEMPORARY TABLESPACE tempts1 TEMPFILE 'D:\app\oracle\oradata\apver\TEMP01.DBF'
    SIZE 16M REUSE
UNDO TABLESPACE undotbs1 DATAFILE 'D:\app\oracle\oradata\apver\UNDOTBS01.DBF'
    SIZE 64M REUSE AUTOEXTEND ON MAXSIZE UNLIMITED;
```

Create and Configure the Database

With your configuration and initialization files in place and edited for your unique installation, and with the required directories created, you are ready to create the new database instance, `apver`. First, let's set our environment to specify the instance we want to deal with to `apver`. This keeps us from tramping on the parameters and identity of the default database, `orcl`.

In an administrator command prompt window, set the `ORACLE SID` environment variable. This setting will be in effect as long as the command prompt window is open. After it is closed, the setting will go away, so redo this if you need to come back to these procedures in a new command prompt window.

```
SET ORACLE SID=apver
```

You already created a new Oracle service for the `apver` instance (it will have a name like `OracleServiceapver`), and that service should be running. You can check what Oracle services are running from the Services area of the Computer Management Windows program—scroll down to the services that start with "Oracle." All other Oracle services can be stopped and, in fact, it is safer while manually installing a new Oracle instance if you stop all the others.

We are going to run SQL*Plus and basically connect to NO instance (No Login or `NOLOG`) `AS SYSDBA`. This is the equivalent to what `CONNECT INTERNAL` had been in earlier versions of Oracle database. You must be an administrator on the server, or the account that installed Oracle Database, in order to run this `CONNECT` command.

```
%ORACLE HOME%\BIN\sqlplus /NOLOG

CONNECT/AS SYSDBA
```

■ **Note** These commands can be found in the file *Chapter11/apver/ApverSys.sql*

We want to be sure we are working on the `apver` instance. The `CONNECT` command should display the message "Connected to an idle instance." If you see the error message "ORA-12560: TNS:protocol adapter error," then your `apver` instance service is not running. Start it as described previously, using the Services area of the Computer Management program.

If an Oracle database instance is running, the message will simply be "Connected." If you see that message, check what instance you are using with this command:

```
SELECT VALUE FROM V$PARAMETER WHERE NAME = 'db name';
```

If this shows the instance to be apver, review your progress so far to recall if you've already created the database. If not, just shut down the template apver instance with the shutdown command. (Note that if you stop the service and restart it (or reboot your computer), a template Oracle instance will be started as apver.)

```
SHUTDOWN
```

However, if that SELECT query shows that you are connected to a different instance, you need to stop the Oracle service that runs that instance and assure that your ORACLE SID environment variable is set to apver.

Continuing on, we will request that messages from our SQL*Plus session be spooled to a log file. We will have to remember to close the spool file before we exit this session of SQL*Plus.

```
SPOOL apver.log
```

Now, we start a database instance defined by the *init.ora* parameters file. There are no database files to mount, in fact we haven't even defined the database files yet, so we say NOMOUNT.

```
STARTUP NOMOUNT PFILE=D:\app\oracle\admin\apver\pfile\init.ora
```

This will show you the System Global Area memory allocation, which is available to Oracle database. It is out of this memory pool that Java resources will be allocated (discussed later).

```
Total System Global Area 2522038272 bytes
```

Now call the script to create our database (modify this command with the path to your command file). This will take a few minutes as the large database files are created. (This is an example of how you call a file with SQL commands from within SQL*Plus; prefix the file name with an at (@) sign.)

```
@Chapter11\apver\ApVerDBCreate.sql;
```

You can examine the existence and sizes of the database files by browsing the directories we created previously: *D:\app\oracle\oradata\apver* and *D:\app\oracle\flash_recovery_area\apver*. If you don't get the "Database created" success message, you may have to delete files in those directories and start over—it is probably a typo in the initialization or command files that you will need to fix.

Change the Passwords for SYS and SYSTEM Users

Even without logging in as a SYS user, we can and must set the passwords for both the SYS and SYSTEM users. Substitute a complex password in each of the following commands:

```
ALTER USER SYS IDENTIFIED BY sys password;
ALTER USER SYSTEM IDENTIFIED BY system password;
```

■ **Caution** This is security step one. Do not proceed until you have accomplished this step.

Store Database Parameters in the Database

It is in our best interest to import our parameters settings from init.ora into a server parameter file. This is done through a command on the database.

```
CREATE SPFILE FROM PFILE;
```

This pulls the settings from the *init.ora*, PFILE file in the default location, *%ORACLE_HOME%\DATABASE\INITAPVER.ORA* and places them in a server-formatted (not hand-editable) file that corresponds to this database instance, *SPFILEAPVER.ORA* in the same directory. The main benefit of having our parameter settings in the server parameter file is that they can be dynamically modified by database commands, and the effects occur (often) both in the SPFILE and in the running Oracle instance. Because the SPFILE is modified by those commands, they are retained across Oracle instance restarts and server reboots.

Restart the database to use the new parameter settings. Notice that you don't have to specify the *init.ora* file. This time when we call `STARTUP`, we will use our newly created server parameter file (SPFILE) and will mount the database files:

```
SPOOL OFF;
SHUTDOWN;
STARTUP;
SPOOL apver2.log;
```

■ **Note** These spool log files will be created in the current directory of your command prompt.

Increase the Quantity of Processes

You may recall seeing in the *init.ora* file a standard setting of 150 processes. That sets a limit on concurrent Oracle connections. We would like to handle a large number of concurrent connections to do application verification. Imagine everyone getting to work on Monday morning and logging into one or more of our secure Oracle applications. We could easily exceed 150 concurrent connections.

Also recall that we configured a special profile for the Application Verification, `appver` user, `appver prof`. For that profile, we set the `SESSIONS PER USER` to be unlimited. However, we made note that the actual limit was controlled by the number of processes. Let's bump up the number of processes.

First connect as `SYS`, and enter the new `SYS` password that we just set. You can connect using the `TNSlistener` service through the first syntax shown in the following, or connect directly to the database as in the second syntax, as long as your `ORACLE SID` environment variable is set to `apver`. In either case, you might want to start the listener service from the Windows Computer Management application.

```
CONNECT SYS@apver AS SYSDBA;
CONNECT SYS AS SYSDBA;
```

We will increase the number of processes for this special-purpose instance, `apver` to a quantity of 500. Issue the command to set the number of process to 500.

```
ALTER SYSTEM SET PROCESSES=500
    COMMENT='Allow more concurrent Application Verification sessions.'
    SCOPE=SPFILE;
```

We give the scope of the change to be `SPFILE`, which means that we only change the stored parameter. This is a specific case where we can issue an `ALTER SYSTEM` database command to modify the `SPFILE` settings, but we cannot immediately update this parameter (the number of processes) in the running database instance. To realize the increased number of processes, we need to shut down and restart the Oracle database instance.

```
SHUTDOWN IMMEDIATE
STARTUP
```

Now log back in and check the parameter setting for the number of processes:

```
CONNECT SYS@apver AS SYSDBA;
SELECT VALUE FROM V$PARAMETER WHERE NAME = 'processes';
```

Run Oracle Scripts to Add Basic Database Capabilities

When we run the following scripts to add basic database capabilities, we will enter the path to each script at the SQL*Plus command line. For best practices, enter the complete path so that no other/older/alternate script is run by accident.

The first script we will run is one that will build the data dictionary views. We need the `ALL USERS` view to do our proxy authentication and SSO. Beyond that, other aspects of running an Oracle database are heavily dependent on the Data dictionary, so we need this.:

```
@D:\app\oracle\product\11.2.0\dbhome 1\RDBMS\ADMIN\catalog.sql
```

Next, we will run the scripts to build the PL/SQL Procedural Option and the SQL*Plus Product User Profile. We need these capabilities in order to configure and run stored procedures and functions. These scripts also create some views in the data dictionary. Note that *catproc.sql* takes a good long while to run. I suggest you get a cup of coffee—that's what I do.

```
@D:\app\oracle\product\11.2.0\dbhome 1\RDBMS\ADMIN\catproc.sql
-- catproc also calls catpexec.sql calls execsec.sql calls
--   secconf.sql, which configures default profile and auditing settings
@D:\app\oracle\product\11.2.0\dbhome 1\sqlplus\ADMIN\pupbld.sql
```

The next script we need to run will build the Extensible Markup Language (XML) Database (XDB). XML is a syntax that permits us to present relational data in a structured text format, and XDB specifically allows Oracle data to be presented and delivered by XML. You might wonder why we need this capability—we don't seem to use XML anywhere in this book. However, we do URL lookups (browse to a web page) to accomplish our two-factor authentication, and granting access to use network ports (DBMS_NETWORK_ACL_ADMIN) is a feature provided when we build XDB. So we need this.

The script to build XDB requires that we pass a number of arguments. The first argument is the `SYS` user password. After that, we list the user and temporary tablespaces where this script will be applied. And lastly we indicate that we will not (`NO`) use SECUREFILE Large Objects (lobs) – they are not supported in non-ASSM (Automatic Segment Space Management) tablespaces. We don't need it, but if you want to include ASSM (automatically handles `pctused` and free lists), then specify `SEGMENT SPACE MANAGEMENT AUTO` when you create your tablespace, not included here:

```
@D:\app\oracle\product\11.2.0\dbhome 1\RDBMS\ADMIN\catqm.sql sys password users tempts1 NO
```

Configure Database to Use UTL_MAIL Package

The ability to send mail is not an automatically included feature of an Oracle database. As you recall, we added that feature to the original `orcl` instance when we implemented two-factor authentication. We will add it to the `apver` instance as well. In fact, we will be doing two-factor authentication now as a function within application verification, not in relation to each Oracle connection we make.

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

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

The extension on that second script file (*plb*) should look familiar to us now. It is a wrapped procedure.

Install DBMS_JAVA package

If you have installed Java in Oracle database before, or if you've looked at the instructions for doing so, you may have seen a precursor step listed. It used to be that you needed to specify memory pool sizes for java, setting `java pool size` and `shared pool size` to 150M each. This is no longer required, as of Oracle Database 11g. These allocations are handled automatically as a part of 11g Automatic Memory Management.

You can see the current settings for these parameters by selecting them from the database.

```
SELECT NAME,VALUE FROM V$PARAMETER WHERE NAME IN ('SHARED POOL SIZE','JAVA POOL SIZE');
```

You'll see that in 11g, these parameters are both set to 0. The required memory will be automatically provided from the global `MEMORY TARGET` and `TARGET SGA` settings. Recall that when we mounted the database we saw a report of the Total System Global Area. That is the memory that is partially available to Java in the database.

A number of scripts are used to configure and enable the Oracle JVM. We will only be running two of them—the ones we need for our application verification security processes. Of the five scripts generally listed to build Java into the Oracle database, we will only execute *initjvm.sql* and *catjava.sql*. Again, our reasoning for being selective is that we will only enable the capabilities we need, thus avoiding some potential security weaknesses.

```
@D:\app\oracle\product\11.2.0\dbhome 1\javavm\install\initjvm.sql;
--@D:\app\oracle\product\11.2.0\dbhome 1\xdk\admin\initxml.sql;
--@D:\app\oracle\product\11.2.0\dbhome 1\xdk\admin\xmlja.sql;
@D:\app\oracle\product\11.2.0\dbhome 1\RDBMS\ADMIN\catjava.sql;
--@D:\app\oracle\product\11.2.0\dbhome 1\RDBMS\ADMIN\catexf.sql;
```

Running *initjvm.sql* will take a while, so take a break. After all that, it is time to close our spool log file, so execute the `SPOOL OFF` command. Then you can browse the spool file for errors—it is in the current directory from which you started SQL*Plus.

```
SPOOL OFF
```

See What's Installed

Shut down the database and restart the `TNSListener` and database services from the Computer Management application (type Computer Management at the Start menu/Run command):

```
SHUTDOWN IMMEDIATE;
EXIT

Computer Management
        Restart TNSlistener service
        Start orcl instance service
        Restart apver instance service
```

Then run SQL*Plus and connect to each of our Oracle instances, one at a time. Sometimes, the TNSListener service takes a minute or two to register the database services after being restarted. If your SQL*Plus connect doesn't work, try again in a minute. Execute the SELECT command from ALL_REGISTRY_BANNERS to see what services have been built.

```
SQLPLUS SYS@ORCL AS SYSDBA
SELECT * FROM ALL REGISTRY BANNERS;

CONNECT SYS@apver AS SYSDBA
-- compare to what's installed in initial database
SELECT * FROM ALL REGISTRY BANNERS;

Oracle Database Catalog Views Release 11.2.0.1.0 - 64bit Production
Oracle Database Packages and Types Release 11.2.0.1.0 - Development
Oracle XML Database Version 11.2.0.1.0 - Development
JServer JAVA Virtual Machine Release 11.2.0.1.0 - Development
Oracle Database Java Packages Release 11.2.0.1.0 - Development
```

Connect from Remote AS SYSDBA

You may want to be able to connect to the apver instance as SYS from a remote GUI administrative application, like TOAD, or from any remote application (including SQL*Plus); however, there is a constraint on the ability to connect to a database instance remotely with the AS SYSDBA privilege. In order for that connection to succeed, a remote login password file must exist for the database instance. In order to remotely login as SYS AS SYSDBA to the apver instance, we will need to create the remote login password file for apver.

First of all, check that we do not have a remote login password file for the apver instance. This can be accomplished by executing an innocuous command that touches that file, GRANT SYSDBA. As SYS user, execute this:

```
GRANT SYSDBA TO SYS;
```

If the remote SYSDBA login password file already exists, you will not get an error, but on the apver instance you will probably see this error:

```
ERROR at line 1:
-ORA-01994: GRANT failed: password file missing or disabled
```

You can create a remote login password file by executing the orapwd command (found in the ORACLE_HOME bin directory). You will need to provide the default file name for the apver instance, *PWDapver.ora* and the password for the SYS user.

```
orapwd file=%ORACLE_HOME%\database\PWDapver.ora password=sys password
```

Configure CREATE_SESSION_ROLE and APPSEC User

We should now be able to connect to the apver instance as SYS from either a local command prompt or from a remote session. Go ahead and connect as SYS AS SYSDBA so that we can create the structures needed for application verification.

■ **Note** These commands can be found in the file *Chapter11/apver/NewSys.sql.*

Create a `create session role` role identical to what we created on the `orcl` instance, and then create the `appsec` user. See Listing 11-46. Be sure to give a complex password for `appsec`. Also give a sufficiently large `QUOTA` on a default tablespace for `appsec` in order to hold structures and data for application verification.

Listing 11-46. *Create Initial Role and User*

```
CREATE ROLE create session role NOT IDENTIFIED;
GRANT CREATE SESSION TO create session role;

GRANT create session role TO appsec IDENTIFIED BY password;

ALTER USER appsec DEFAULT TABLESPACE USERS QUOTA 10M ON USERS;
```

Create a Database Link to the ORCL Instance

As I mentioned at the onset of this section, we will be configuring the bare minimum configuration in order to do application verification in this new, `apver` instance. For that reason, we will rely on the `SYS` user to configure all our structures, even those in the `appsec` schema. Well, that is except for one item that we might as well get out of the way right now.

We want to create a database link to be used by the `appsec` structures in particular to do two-factor authentication. Recall that we have stored an e-mail address in the `HR.EMPLOYEES` table, and we created another table with addresses to use for two-factor authentication in the `HR` schema, `emp mobile nos`. However, those tables are on a different Oracle instance, `orcl`. As a part of our application verification process, occurring on the `apver` instance, we need to read those tables across a database link.

Update TNSNAMES.ora for our Database Link

The way we read data from a different instance is by using a database link. To use the database link, the database that wants to read data, `apver` needs to know how to find the other database instance. This location and direction information is normally retained in a *TNSNAMES.ora* file on the Oracle database, as in Listing 11-47. Assure you have an entry for the `orcl` instance. While we are here, add another entry for the new `apver` instance.

Listing 11-47. *Edit TNSNAMES.ora File*

```
edit %ORACLE HOME%\NETWORK\ADMIN\tnsnames.ora

ORCL =
  (DESCRIPTION =
    (ADDRESS = (PROTOCOL = TCP)(HOST = localhost)(PORT = 1521))
    (CONNECT DATA =
      (SERVER = DEDICATED)
      (SERVICE NAME = orcl)
    )
  )

APVER =
  (DESCRIPTION =
    (ADDRESS = (PROTOCOL = TCP)(HOST = localhost)(PORT = 1521))
    (CONNECT DATA =
      (SERVER = DEDICATED)
      (SERVICE NAME = apver)
    )
  )
```

Permit appsec to Create a Database Link

As SYS user grant the privilege for appsec user to create a personal database link in the appsec schema. Unlike most other create statements, this is one that cannot be done in a different schema; we need to be appsec user to create a personal database link in the appsec schema. As SYS, execute the commands in Listing 11-48 to create a limited appsec role for this grant.

Listing 11-48. *Grant Create Link Privilege to appsec*

```
-- Must grant to user, not role since roles not exist without session
-- This is used in MASK/UNMASK - not needed on apver instance
GRANT EXECUTE ON sys.dbms crypto TO appsec;

CREATE ROLE appsec role NOT IDENTIFIED;
-- Give Application Security privilege to create Database Link
GRANT CREATE DATABASE LINK TO appsec role;
GRANT appsec role TO appsec;
-- Make the APPSEC ROLE a non-default role for the APPSEC user
ALTER USER appsec DEFAULT ROLE ALL EXCEPT appsec role;
```

Create the Personal Database Link as APPSEC

We made the appsec role a non-default role, so now we need to log in as appsec and set our role to appsec role.

```
SET ROLE appsec role;
```

■ **Note** These commands can be found in the file *Chapter11/apver/NewAppSec.sql.*

Execute the command in Listing 11-49 to create the database link we need to attach to the orcl instance (substitute the password for appsec user on orcl instance into this command).

Listing 11-49. *Create Database Link as appsec User*

```
CREATE DATABASE LINK orcl link
CONNECT TO appsec
IDENTIFIED BY password
USING 'orcl';
```

Test the new database link by selecting from a table that is available to appsec on the orcl instance:

```
SELECT * FROM hr.v emp mobile nos@orcl link;
```

We are returning to the SYS user now in order to accomplish much of the rest of this installation of our Application Security structures on the new apver instance.

Now that we have created the database link as appsec, we will no longer need to log in as our appsec user, so we will want to disable logins. The quickest way to do that is to expire the password for appsec. To do that, execute the following command as SYS user:

```
ALTER USER appsec PASSWORD EXPIRE;
```

You will want to remember that you did this so you don't worry when you cannot connect to apver as appsec. In effect, you can only connect to apver instance as SYS. You could connect as appver user, but the logon trigger and other security will prevent or limit what you can do.

Grant Access for APPSEC User to Reach out of the JVM Security Sandbox

In order to accomplish two-factor authentication, our appsec user will need to send e-mail and open connections to web servers. These abilities require that appsec be able to reach out from the Oracle JVM, outside the standard security sandbox. We will grant the privilege to do that here in Listing 11-50, as we did for the orcl instance in Chapter 9.

Listing 11-50. *Grant Oracle JVM Security Sandbox Permissions*

```
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;
/

CALL DBMS JAVA.GRANT PERMISSION(
    'APPSEC',
    'java.net.SocketPermission',
    'www.org.com:80',
    'connect, resolve'
);
```

Revoke PUBLIC Grant on Sensitive Data Dictionary Views

One of our primary reasons for establishing a separate Oracle instance in order to do application verification/authorization is that we want to be even more restrictive on what a hacker would be able to see and do if he gained access to the appver password. Remember, we have only obfuscated the password; we have not encrypted it.

Besides limiting the number of users with passwords on the new apver instance, we also want to remove the ability to SELECT on certain by-default PUBLIC views from the Oracle Database Data Dictionary. In particular, we are going to remove PUBLIC access from the ALL USERS, ALL SOURCE, ALL TRIGGERS and ALL VIEWS views. These particular data dictionary views expose accounts and code that can be leveraged in a computer security attack. Even though for most of these views, the exposure is only to code that has been granted to the user, we prefer to allow the user to run code without being able to see the code.

The ALL USERS view will still need to be accessible by appsec in order to successfully execute procedures for 2-factor authentication. In Listing 11-51, we will GRANT SELECT on ALL USERS directly to the appsec user, not to a role. The appsec user password is expired, so the ALL USERS view will remain inaccessible to hackers.

Listing 11-51. *Secure Public Data Dictionary Views*

```
GRANT SELECT ON sys.all users TO appsec;

REVOKE SELECT ON sys.all users FROM PUBLIC;
REVOKE SELECT ON sys.all source FROM PUBLIC;
REVOKE SELECT ON sys.all source ae FROM PUBLIC;
REVOKE SELECT ON sys.all triggers FROM PUBLIC;
REVOKE SELECT ON sys.all views FROM PUBLIC;
REVOKE SELECT ON sys.all views ae FROM PUBLIC;
```

■ **Note** If this were being used in production at the moment, you would grant the appropriate narrow privilege (as shown) before revoking the inappropriate broader privilege in order to keep appropriate functions working during the update.

Create the Remaining Structures for Application Authorization

The remainder of the *NewSys.sql* script for the `apver` instance configures all the structures we need to do application authorization. Most of the structures are created in the `appsec` schema. To do this as `SYS` user, we simply prefix the name of the structure we are creating with the schema name `appsec.`

We are doing this as `SYS` user to keep from granting administrative privileges to any other user, even `appsec`. This is the most secure, but control of the `SYS` password is mandatory.

Before running the script, copy and paste over the wrapped functions, `f mask` and `f unmask`, with the versions that you created and wrapped. Table 11-2 provides a list of the structures we will be creating, in creation order.

Table 11-2. *Structures Created in apver Instance*

TABLE	APPSEC.T_APPSEC_ERRORS
TABLE	APPSEC.T_APPSEC_ERRORS_MAINT
PACKAGE	APPSEC.APP_SEC_PKG
PROCEDURE	APPSEC.P_APPSEC_ERRORS_JANITOR
TRIGGER	APPSEC.T_APPSEC_ERRORS_IAR
TABLE	APPSEC.T_TWO_FACT_CD_CACHE
PROFILE	APPVER_PROF
USER	APPVER
USER	OSUSER
USER	OSADMIN
TABLE	APPSEC.T_APPLICATION_REGISTRY
TABLE	APPSEC.T_APP_CONN_REGISTRY
TABLE	APPSEC.T_APPLICATION_KEY

TRIGGER	APPSEC.T_APPLICATION_KEY_BUDR
FUNCTION	APPSEC.F_IS_SSO
PACKAGE	APPSEC.APPSEC_ONLY_PKG
PROCEDURE	APPSEC.P_CHECK_ROLE_ACCESS
PROCEDURE	APPSEC.P_APPVER_LOGON
TRIGGER	APPSEC.T_SCREEN_APPVER_ACCESS
PACKAGE	APPSEC.APPSEC_ADMIN_PKG
ROLE	APPVER_ADMIN
PACKAGE	APPSEC.APPSEC_PUBLIC_PKG
FUNCTION	APPSEC.F_MASK (wrapped)
FUNCTION	APPSEC.F_UNMASK (wrapped

■ **Note** Because we are not creating the `secadm` user on the `apver` instance, we cannot create the `t screen appver access` trigger in the `secadm` schema. We will create it in `appsec` schema instead.

There are a couple places in the script where you will need to substitute a real OS User ID (i.e., Windows login name) for a placeholder. Search and replace both `OSUSER` and `OSADMIN`. `OSUSER` is a person (or multiple persons) who wants to run our secure applications. `OSADMIN` is a person, like you, who will need to connect in order to register application connection strings in the database.

On `apver`, a couple stored procedures go across *database links*, `f is cur cached cd` and `p get emp 2fact nos.` In those structures, we see the reference types (like that for `os user in` Listing 11-52) were converted to standard type declarations; we can't reference across a database link. Look at the `FROM` clause in the `SELECT` statement in Listing 11-52. We are getting data from three views across the database link: `v emp mobile nos@orcl link`, `v employees public@orcl link`, and `v sms carrier host@orcl link`.

Listing 11-52. *Get Data Across Database Link into Procedure*

```
PROCEDURE p get emp 2fact nos(
    --os_user               hr.v_emp_mobile_nos.user_id%TYPE,
    os_user               VARCHAR2,
    fmt string            VARCHAR2,
    m employee id     OUT NUMBER,
```

```
    m com pager no    OUT VARCHAR2,
    m sms phone no    OUT VARCHAR2,
    m sms carrier url OUT VARCHAR2,
    m email           OUT VARCHAR2,
    m ip address      OUT v two fact cd cache.ip address%TYPE,
    m cache ts        OUT VARCHAR2,
    m cache addr      OUT v two fact cd cache.ip address%TYPE,
    m application id      v two fact cd cache.application id%TYPE,
    m err no          OUT NUMBER,
    m err txt         OUT VARCHAR2 )
IS BEGIN
    m err no := 0;
    SELECT e.employee id, m.com pager no, m.sms phone no, s.sms carrier url,
        e.email, SYS CONTEXT( 'USERENV', 'IP ADDRESS' ),
        TO CHAR( c.cache ts, fmt string ), c.ip address
    INTO m employee id, m com pager no, m sms phone no, m sms carrier url,
        m email, m ip address, m cache ts, m cache addr
    --FROM hr.v emp mobile nos m, hr.v employees public e,
    --    hr.v sms carrier host s, v two fact cd cache c
    FROM hr.v_emp_mobile_nos@orcl_link m, hr.v_employees_public@orcl_link e,
        hr.v_sms_carrier_host@orcl_link s, v two fact cd cache c
```

Create Java Structures

Open each of these files and execute them on the apver instance from a SQL client. For each of the *.java* files, uncomment the top line with the SQL `CREATE OR REPLACE AND RESOLVE JAVA SOURCE` statement, and for *OracleJavaSecure.java*, edit the domain, SMTP host and base URL members, as before. Execute each of these to create the Java structures in the apver instance. Be sure to substitute values for domains and hostnames appropriate to your corporate environment into the *OracleJavaSecure.java* before creating the Java structures in Oracle.

Chapter11/orajavsec/OracleJavaSecure.java
Chapter11/orajavsec/RevLvlClassIntfc.java
Chapter11/orajavsec/OJSC.java
Chapter11/testojs/TestOracleJavaSecure.sql

Remove Application Verification from the ORCL Instance

At this point, we no longer need application verification in the ORCL instance. We will disable connections by the application verification, appver user to ORCL. Do that from a SQL client connection to ORCL instance as appsec or SYS user with the following command.

■ **Note** This is to be done on the ORCL instance only, and only if you are installing a second database instance, apver, to do application verification.

```
ALTER USER appver PASSWORD EXPIRE;
```

Test Enhanced Security

We are at the point now where we can test everything we have established, including the segregation of Application Authentication on a separate Oracle instance, `apver`. We will do our testing in two parts: first using the `main()` method in `OracleJavaSecure`, then using the separate application, `testojs.TestOracleJavaSecure`.

Once again, assure that you have set the domain and other addresses at the top of *OracleJavaSecure.java* to be appropriate for your organization. Also assure that the SQL command at the top of the file has been commented. (The SQL commands at the top of the other *.java* files in this chapter should also remain commented in order to be compiled.)

■ **Note** The assumption in the following text is that you are running application verification on a separate database instance, `apver`. If you are not, then the only difference is that you will be able to connect as `appsec` user to the `orcl` instance—you will not have to connect as `SYS`.

Encode the APPVER User Password for APVER Instance

The first time we run `OracleJavaSecure`, we will have just one goal: to create a new encoded Oracle connection string for the `appver` user on the `apver` instance. Scroll down to the `main()` method in *OracleJavaSecure.java* and edit the `encodeThis String` components, shown in Listing 11-53, to point at the `apver` instance instead of `orcl`.

Listing 11-53. *Switch appver Connection String from ORCL to apver*

```
encodeThis = "jdbc:oracle:thin:appver/" + encodeThis +
    //"@localhost:1521:orcl";
    "@localhost:1521:apver";
```

■ **Caution** If you did not create an additional Oracle database instance dedicated to application verification, then do not make this change to the code. You do not need to update the encoded connection string, `prime`.

Then compile the class and run it. From the Chapter11 directory, execute:

```
javac orajavsec/OracleJavaSecure.java
java orajavsec.OracleJavaSecure appverPassword
```

You will see something like the following as a result:

```
Main encodes a new APPVER password if given.
After encoding, paste encoded string
in setAppVerConnection() method.

030a42105f1b3311133a0048370707005f020419190b524204041819015c390f5300121b3314303a
```

0a112203060116174e585a5c115704041e0a16

```
jdbc:oracle:thin:appver/appverPassword@localhost:1521:apver
```

We need to place that encoded string (see the bold data) into the `setAppVerConnection()` method in *OracleJavaSecure.java*. That method will look like Listing 11-54 when done (the `prime String` is all one line, though shown here with word-wrap onto a second line).

Listing 11-54. *Embed Newly Encoded appver Connection String into OracleJavaSecure Code*

```
    private static void setAppVerConnection() {
        try {
            // Set this String from encoded String at command prompt (main)
            String prime =
"030a42105f1b3311133a0048370707005f020419190b5242040418190l5c390f5300121b3314303a0a112203060
116174e585a5c115704041e0a16";
            setConnection( OJSC.y( prime ) );
            appVerConn = conn;
        } catch( Exception x ) {
            x.printStackTrace();
        }
    }
```

■ **Note** It is this one value, `prime` that directs our applications to use an alternate database instance for application verification.

Edit the Application Passwords to Be Used

We are going to upload a list of connection strings to Oracle database for the `HRVIEW` application. Update the `main()` method of `OracleJavaSecure` to have the correct passwords (substitute for the "password" string) for `HR` and `appusr` users, shown in Listing 11-55. Correct the other aspects of your Oracle application connection strings as well.

■ **Note** These connection strings are intended to connect to the `orcl` instance, not the new `apver` instance.

Listing 11-55. *Connection Strings for Application, OracleJavaSecure.main()*

```
    putAppConnString()( "Orcl", "hr",
        "password", "localhost", String.valueOf( 1521 ) );
    putAppConnString()( "Orcl", "appusr",
        "password", "localhost", String.valueOf( 1521 ) );
```

Then compile the class again. From the Chapter11 directory, execute:

```
javac orajavsec/OracleJavaSecure.java
```

Run Main to Test

Now we will run `OracleJavaSecure` at least five more times. You must run this as an OS (Windows) user whose matching Oracle user has been granted the `appver admin` role: your equivalent of `OSADMIN`. The first time, we will generate a two-factor authentication code. The results will look like this:

```
Chapter11>java orajavsec.OracleJavaSecure
Main encodes a new APPVER password if given.
After encoding, paste encoded string
in setAppVerConnection method.
You may enter APPVER password on command line.
Domain: ORGDOMAIN, Name: OSADMIN
Please rerun with 2-Factor Auth Code!
```

We should either receive that code by two-factor authentication code distribution, or find it by querying the `appsec.v two fact cd cache` view on the `apver` instance. You will not be able to connect as `appsec` user to see that view, because we expired the `appsec` password; but you can select from the view as `SYS` user.

Then we execute the same command with the two-factor authentication code as a parameter on the command line:

```
Chapter11>java orajavsec.OracleJavaSecure 1234-5678-9012
Main encodes a new APPVER password if given.
After encoding, paste encoded string
in setAppVerConnection method.
Domain: ORGDOMAIN, Name: OSADMIN
connsHash.size = 0
connsHash.size = 0
Domain: ORGDOMAIN, Name: OSADMIN
2011-06-05 21:00:06
```

You can see that when we get the Oracle connections list for this application from the `apver` instance, the list is empty, `connsHash.size = 0`. We are able to insert connection strings into the list and use them. Using one of the connection strings, we query the database to get the `SYSDATE`.

When we run this command again, we see that for a minute our `connsHash.size = 2`, and later `= 1`. The first value is the number of connection strings in the list we get for this application from Oracle. The second value is the number we have after calling `removeAppConnString()`. We immediately call `putAppConnString()` twice to add and overwrite the connection strings, and then we call `putAppConnections()` to store the list of two connection strings in Oracle for this application. We again use one of them to get `SYSDATE` from Oracle.

```
Chapter11>java orajavsec.OracleJavaSecure 1234-5678-9012
Main encodes a new APPVER password if given.
After encoding, paste encoded string
in setAppVerConnection method.
Domain: ORGDOMAIN, Name: OSADMIN
connsHash.size = 2
connsHash.size = 1
Domain: ORGDOMAIN, Name: OSADMIN
```

```
2011-06-05 21:00:23
```

Run Main to Copy Connection Strings to New Version

We will edit *OracleJavaSecure.java* one more time in order to test our ability to copy a list of connection strings from an older version of an application to the new version. Open the file and edit the inner class `InnerRevLvlClass` (near the top), incrementing the version number, for example from `20110101a` to `20110101b`. See Listing 11-56.

Listing 11-56. *Change Version/Revision of Application Inner Class*

```
    private static class InnerRevLvlClass
        implements Serializable, RevLvlClassIntfc
    {
        private static final long serialVersionUID = 2011010100L;
        private String innerClassRevLvl = "20110101b";
        public String getRevLvl() {
            return innerClassRevLvl;
        }
    }
```

■ **Note** We do not have to CREATE a new version of this java code on the Oracle Database, because the serialVersionUID has not changed.

In the `main()` method, find the commented call to `copyAppConnections()` and uncomment that call. Assure that the old version number `String` in that call, Listing 11-57, matches the `innerClassRevLvl` you changed from previously.

Listing 11-57. *Copy List of Connection Strings from Previous Version*

```
copyAppConnections( "20110101a" );
```

Compile `OracleJavaSecure` and run it. Use the same two-factor authorization code if it has been less than ten minutes, else request and use a new one. This will exit after copying the list of connection strings from the old version to the new one.

```
Chapter11>javac orajavsec/OracleJavaSecure.java
Chapter11>java orajavsec.OracleJavaSecure
Chapter11>java orajavsec.OracleJavaSecure 1234-5678-9012
Main encodes a new APPVER password if given.
After encoding, paste encoded string
in setAppVerConnection method.
Domain: ORGDOMAIN, Name: OSADMIN
```

As SYS user on apver, you can see that there is a new version of the application class file and associated list of connection strings by querying the `appsec.v app conn registry` view.

```
"CLASS NAME"                                "CLASS VERSION" "CLASS INSTANCE" "UPDATE DT"
```

```
"orajavsec.OracleJavaSecure$InnerRevLvlClass" "20110101b"     "ACED00057372... "11-JUN-11"
"orajavsec.OracleJavaSecure$InnerRevLvlClass" "20110101a"     "ACED00057372... "11-JUN-11"
```

Once more we edit `OracleJavaSecure.java` and comment two areas of the `main()` method. First comment the line that copies the list of connection strings from an old version to the new version.

```
//copyAppConnections( "20110101a" );
```

Then comment the lines that would overwrite and insert new connection strings into the list.

```
//putAppConnString( "Orcl", "hr",
//    "password", "localhost", String.valueOf( 1521 ) );
//putAppConnString( "Orcl", "appusr",
//    "password", "localhost", String.valueOf( 1521 ) );
```

For the final times, compile `OracleJavaSecure` and run it. The first time it runs, you will see that there were already two connection strings (`connsHash.size = 2`) in the list that was copied from the previous version of the application. In the middle of the run, we call `removeAppConnString()` and then we run `putAppConnections()` to update Oracle with a list of only one connection string for this application. The second time it is run, you will see that, because we didn't update and add connection strings, there is only one to start with (`connsHash.size = 1`).

```
Chapter11>javac orajavsec/OracleJavaSecure.java
Chapter11>java orajavsec.OracleJavaSecure
Chapter11>java orajavsec.OracleJavaSecure 1234-5678-9012
Main encodes a new APPVER password if given.
After encoding, paste encoded string
in setAppVerConnection method.
Domain: ORGDOMAIN, Name: OSADMIN
connsHash.size = 2
connsHash.size = 1
Domain: ORGDOMAIN, Name: OSADMIN
2011-06-11 18:04:15
Chapter11>java orajavsec.OracleJavaSecure 1234-5678-9012
Main encodes a new APPVER password if given.
After encoding, paste encoded string
in setAppVerConnection method.
Domain: ORGDOMAIN, Name: OSADMIN
connsHash.size = 1
connsHash.size = 1
Domain: ORGDOMAIN, Name: OSADMIN
2011-06-11 18:04:26
```

We conclude that our `copyAppConnections()` process works to replicate an existing list of connection strings from one version of the application to the next (with an updated `innerClassRevLvl` in the inner class.) This will be a timesaver when we upgrade our application and don't need to change connection strings.

Test from a Different Application, TestOracleJavaSecure

`TestOracleJavaSecure` is independent from whatever instance is running application verification. It will act as a client application, calling methods in the `OracleJavaSecure` class in order to use those security features. Using `OracleJavaSecure`, the test application will do the following things:

- Accomplish SSO and 2-factor authentication.
- Register itself (its name and application inner class).
- Store a list of connection strings for the application.
- Retrieve the list of connection strings from Oracle.
- Transfer and store the connection strings in encrypted form.
- Decrypt and use the connection strings to query application data.
- Query sensitive data in encrypted form and decrypt it for use

Edit the file *Chapter11/testojs/TestOracleJavaSecure.java*. Correct the password for `appusr` shown in Listing 11-58. We will add this user to a list of connection strings for the `TestOracleJavaSecure` application. We are still using `HRVIEW` as our application ID, which means we will be connecting as `appusr` and using the role `hrview role` (as registered in `appsec.v application registry`).

Listing 11-58. *Edit TestOracleJavaSecure to Test*

```
String applicationID = "HRVIEW";
Object appClass = new AnyNameWeWant();
OracleJavaSecure.setAppContext( applicationID, appClass, twoFactorAuth );

OracleJavaSecure.getAppConnections();
if( twoFactorAuth.equals( "" ) ) {
    return;
}

// Demonstrate copy connsHash from previous version
// Only do this once, make sure it worked (see appsec.V APP CONN REGISTRY),
// then comment this line
//OracleJavaSecure.copyAppConnections( "20110131a" );
// Get copied list of connection strings for new version number
OracleJavaSecure.getAppConnections();

OracleJavaSecure.putAppConnString( "Orcl", "appusr",
    "password", "localhost", String.valueOf( 1521 ) );
//Only do this line once -- must be admin account
OracleJavaSecure.putAppConnections();
```

The last line in Listing 11-58 is a call to `putAppConnections()`. That method calls the Oracle function `f set decrypt conns`, which we moved to the `appsec admin pkg` package. Only users with the `appver admin` role can execute structures in the `appsec admin pkg`. In our example, we granted `appver admin` to `osadmin` user. So, to test this facet of security, we will run `TestOracleJavaSecure` as both `osadmin` (an administrative account) and `osuser` (a non-administrative account). You can test `copyAppConnections()` for `TestOracleJavaSecure` by modifying the version number of the inner class and uncommenting `copyAppConnections()`, passing the previous version number.

Compile and Run as Administrative User, OSADMIN

In the code of `TestOracleJavaSecure.main()`, we have three attempts to connect to Oracle database before we try to get encrypted data. We attempt to do these things

- Call `getAppConnections()` to do key exchange and get the list of connection strings for this application.
- Call `putAppConnections()` to update the list of connection strings in Oracle.
- Call `getAAConnRole()` to decode and use a connection string to get a `Connection`.

We are going to compile and run `TestOracleJavaSecure` as an administrative user (a user with the `appver admin` role, `OSADMIN` in our example—probably your OS user ID.) We start out running this application without a 2-factor authentication code. Here are the commands and results:

```
Chapter11>javac testojs/TestOracleJavaSecure.java
Chapter11>java testojs/TestOracleJavaSecure
Domain: ORGDOMAIN, Name: OSADMIN
Please rerun with 2-Factor Auth Code!
java.lang.NullPointerException
Please rerun with 2-Factor Auth Code!
```

The call to `putAppConnections()` reports a `NullPointerException` – we don't trap exceptions from that administrative command as closely as we trap exceptions from methods being called by regular client applications. The other attempts to connect to Oracle point out the need to return with a 2-factor authentication code.

Let's run the code again with a bogus two-factor authentication code:

```
Chapter11>java testojs.TestOracleJavaSecure 123
Domain: ORGDOMAIN, Name: OSADMIN
Oracle error 21) 100, ORA-01403: no data found
java.lang.NullPointerException
Oracle error 21) 100, ORA-01403: no data found
Wrong or old 2-Factor code parameter
```

The same `NullPointerException` was reported by `putAppConnections()`, and the other two attempts to connect to Oracle reported "no data found" error, which is what we get from a bad user or two-factor authentication code. We also trap a `NullPointerException` in Listing 11-59 when we attempt to use the `Connection` coming from `getAAConnRole()` to get an Oracle `Statement`. If we got this far, it is our understanding that we have provided a dubious two-factor authentication code, so we report the problem and exit.

Listing 11-59. *Catch Incorrect Two-Factor Authentication Code*

```
try {
    mStmt = conn.createStatement();
} catch( NullPointerException n ) {
    System.out.println( "Wrong or old 2-Factor code parameter" );
    return;
}
```

For this new user, there will be a new entry in the `appsec.v two fact cd cache` view. This new entry will be designated for the new `employee id`, 304 in this example. Here is an example selection from `appsec.v two fact cd cache`.

```
SQL> select * from appsec.v two fact cd cache;

EMPLOYEE ID APPLICATION ID TWO FACTOR CD  IP ADDRESS DISTRIB CD CACHE TS
----------- -------------- -------------- ---------- ---------- ---------
        304 HRVIEW         2747-4367-3056 127.0.0.1           1 12-JUN-11
        300 HRVIEW         3471-8557-5210 127.0.0.1           3 12-JUN-11
```

When we finally return to execute `TestOracleJavaSecure` with the correct two-factor authentication code, we are able to print out many aspects regarding our proxy connection to Oracle for data queries, connected through the `appusr` user, using the `hrview role`. We also select and decrypt data from our familiar `p select employees sensitive` procedure.

```
Chapter11>java testojs.TestOracleJavaSecure 1234-5678-9012
Domain: ORGDOMAIN, Name: OSADMIN
Domain: ORGDOMAIN, Name: OSADMIN
osadmin
```

APPUSR

```
127.0.0.1
OSADMIN
OSADMIN
HRVIEW ROLE
Oracle success 2)
100, Steven, King, SKING, 515.123.4567, 2003-06-17 00:00:00, AD PRES, 24000, null, null, 90
```

Run as Non-Administrative User, OSUSER

We are going to test our limitation of application connection string updates to just users who have been granted the `appver admin` role. To do this, you need to log in as an OS user who corresponds to an Oracle user without the `appver admin` role, your equivalent of `OSUSER` in our examples. Run `TestOracleJavaSecure` again. You will see that when the application calls the method `putAppConnections()` an exception is thrown with regard to the `appsec admin pkg` package. This user does not have privileges to execute the Oracle function, `f set decrypt conns` that is called from that method.

```
Chapter11>java testojs/TestOracleJavaSecure
Domain: ORGDOMAIN, Name: OSUSER
Please rerun with 2-Factor Auth Code!
java.sql.SQLException: ORA-06550: line 1, column 13:
PLS-00201: identifier 'APPSEC.APPSEC_ADMIN_PKG' must be declared
ORA-06550: line 1, column 7:
PL/SQL: Statement ignored
```

We want to demonstrate other, non-administrative functions using the non-administrative user. To do that, edit *TestOracleJavaSecure.java* and comment the line calling the `putAppConnections()` method. Then recompile and rerun the application. You will see the following successful results:

```
Chapter11>javac testojs/TestOracleJavaSecure.java
Chapter11>java testojs.TestOracleJavaSecure
```

```
Chapter11>java testojs.TestOracleJavaSecure 1234-5678-9012
Domain: ORGDOMAIN, Name: OSUSER
Domain: ORGDOMAIN, Name: OSUSER
osuser
APPUSR
127.0.0.1
OSUSER
OSUSER
HRVIEW ROLE
Oracle success 2)
100, Steven, King, SKING, 515.123.4567, 2003-06-17 00:00:00, AD PRES, 24000, null, null, 90
```

Chapter Review

Our goal in this chapter has been to enhance the security of everything we had built so far. We accomplished that goal on the following fronts:

- We programmed Java to encode the appver user password (connection string).
- We obfuscated the Java program that does encoding/decoding.
- We accomplished secured data encryption for data being stored in the database—specifically our lists of connection strings.
- We established an administrative role that limits who can update connection strings for applications.
- We moved the application verification processes to a new, hardened Oracle database instance, apver.

In addition to working toward enhancing the security, we also delved into the following timely topics:

- Securing an Oracle user password by various means
- Using the Oracle client wallet
- Using Oracle client trace logging
- Using Oracle thin client (JDBC) trace logging
- Using the Oracle wrap utility to obfuscate Oracle functions
- Copying application connection strings from a previous version to the current application version
- Adding other authentication credentials, such as FTP passwords, to our application connection strings list
- Reviewing weaknesses in PUBLIC access to the data dictionary views
- Using database links to read data from another database instance

CHAPTER 12

Administration of Security

Now that we have built Oracle structures and Oracle and Java code, we will need to maintain the data that keeps it running. Primarily, this data consists of users, proxy grants, application registrations, and application connection strings. It has not been too difficult to execute SQL scripts to insert records for one or two users, and one or two applications—especially while we are studying the issues and requirements. But in a year or so, the steps to accomplish these tasks will be long forgotten, along with the reasons for doing them.

However, if we can encapsulate the business rules, logic, and procedure steps in a user-friendly application, we will have a much easier time adding new users and applications. In fact, once we have done that, we will have developed a pretty handy interface, part of which we will provide as a template to the application developers in our organizations, so that they can implement our security code.

A Security Administration Interface

When I talk about user-friendly applications, what I mean is actually well designed, simple GUIs.

The security administration interface we will be exploring in this chapter consists simply of a Login screen, a Menu screen, and seven functional screens. The Login screen is where we expect the user to enter a two-factor authentication code that we send him. It is this screen that we will provide as a GUI template to other application developers so they can implement our security structures.

■ **Note** Files for the security administration interface application can be found in the directory *Chapter12/orajavsec*.

The functional screens will walk the administrator through tasks such as editing employee and user data (especially those elements that we use for single sign-on and two-factor authentication). There are also functional screens for granting both administrative and application proxy privileges. We need to be able to register a new application, and most of that process is done through the GUI. Then we want to edit existing connection strings for an application, and occasionally copy connection strings from a previous version of an application. All those functions are represented on screens in our GUI.