```
            try {
                if( null != stmt ) stmt.close();
            } catch( Exception y ) {}
        }
        return mConn;
    }
```

Note that the call to `p check role access` is identical to what we had been asking our developers to do for themselves. The `applicationID` we use in that call is what we set our static member to when we called `setAppContext()`.

Test Application Authentication, Phase 2

If you haven't already, execute the script to create the stub inner class for `TestOracleJavaSecure` in Oracle database.

■ **Note** Execute the script in *TestOracleJavaSecure.sql* on the Oracle database.

You can query Oracle database to see the Java structures that you have just created. There will be one Java structure for the `TestOracleJavaSecure` outer class, and one for the inner class, resembling this list. Here's the query to execute:

```
SELECT * FROM SYS.ALL OBJECTS WHERE OBJECT TYPE = 'JAVA CLASS' AND OWNER = 'APPSEC';
testojs/TestOracleJavaSecure
/545c0b44 TestOracleJavaSecure
```

Next, edit and compile the `TestOracleJavaSecure` class on the client. Modify the call to `putAppConnString()` with the password, server name, instance name and port number for your Oracle instance.

■ **Note** Edit the code found in the file named *Chapter10/testojs/TestOracleJavaSecure.java.*

The tests here will be relatively simple, but everything we have described so far in this book will have taken place by the time we display a line of encrypted data from the database. This can also serve as an example of what an application developer will have to do to use our security structures.

First you will note that this application implements an inner application class in Listing 10-57.

Listing 10-57. *Second Test Application Inner Class*

```
package testojs;

public class TestOracleJavaSecure {
    public static class AnyNameWeWant
        implements Serializable, RevLvlClassIntfc
```

```
    {
...
```

Set the Application Context

The body of test code resides in the main method. In this case, we are presuming that the client user will call the application from the command line. After the first call, he will need to call it again, including the two-factor authentication code on the command line.

See in Listing 10-58 how the application sets the application context in `OracleJavaSecure` by calling the `setAppContext()` method.

Listing 10-58. *Set Second Test Application Context*

```
    public static void main( String[] args ) {
        OracleCallableStatement stmt = null;
        Statement mStmt = null;
        ResultSet rSet;
        try {
            // Submit two-factor auth code on command line, once received
            String twoFactorAuth = "";
            if( args.length != 0 && args[0] != null ) twoFactorAuth = args[0];
            String applicationID = "HRVIEW";
            Object appClass = new AnyNameWeWant();
            OracleJavaSecure.setAppContext( applicationID, appClass, twoFactorAuth );
```

Store the Connection Strings in Oracle

If this is the first time we have run the `TestOracleJavaSecure` class, both before and after we have a two-factor authentication code, then we can run a few lines to populate the `connsHash` list of connection strings and store them in Oracle, as shown in Listing 10-59.

Listing 10-59. *Store Connections for Second Test Application*

```
    // Only do these lines once
    // If we provided an old twoFactorAuth, will not have connHash -
    // null pointer exception here
    OracleJavaSecure.getAppConnections();
    OracleJavaSecure.putAppConnString( "Orcl", "appusr",
        "password", "localhost", String.valueOf( 1521 ) );
    OracleJavaSecure.putAppConnections();
```

After we have succeeded in submitting our application-specific connection strings to Oracle, we can comment those lines of code and simply use the connection strings that we've stored in and retrieved from Oracle Database.

Get an Application Connection with Role

For this test, we are going to call our new method, `getAAConnRole()`, which as you'll recall gets the application connection and sets the secure application role. This call is shown in Listing 10-60.

Listing 10-60. *Call to Get Application Connection with Role*

```
OracleConnection conn =
    OracleJavaSecure.getAAConnRole( "orcl", "appusr" );
```

See the Proxy Connection

Now that we have our connection, let's see how we have been identified by Oracle database. There are a number of Oracle `SYS CONTEXT` settings that we can query. We will query and display those having to do with our identity in Listing 10-61.

Listing 10-61. *View the Proxy Connection Settings*

```
mStmt = conn.createStatement();

rSet = mStmt.executeQuery( "SELECT SYS CONTEXT( 'USERENV', 'OS USER' )," +
"SYS CONTEXT( 'USERENV', 'PROXY USER' ),SYS CONTEXT( 'USERENV', 'IP ADDRESS' )","+
"SYS CONTEXT( 'USERENV', 'SESSION USER' ), "+
"SYS CONTEXT( 'USERENV', 'CLIENT IDENTIFIER' ) " +
"FROM DUAL" );
if ( rSet.next() ) {
    System.out.println( rSet.getString( 1 ) );
    System.out.println( rSet.getString( 2 ) );
    System.out.println( rSet.getString( 3 ) );
    System.out.println( rSet.getString( 4 ) );
    System.out.println( rSet.getString( 5 ) );
}
rSet = mStmt.executeQuery( "SELECT * FROM sys.session roles" );
if ( rSet.next() ) {
    System.out.println( rSet.getString( 1 ) );
}
```

Get Encrypted Data from Oracle

Finally, we will call our standard demonstration stored procedure, `p select employees sensitive`. This requires additional encryption key exchange with the connection listed in the `connsHash`. The call is shown in Listing 10-62. Our first encryption keys that we exchanged for the `appver` connection cannot be reused with a different Oracle connection, as `appusr` for instance.

Listing 10-62. *Call to Get Sensitive Employees Data*

```
String locModulus = OracleJavaSecure.getLocRSAPubMod();
String locExponent = OracleJavaSecure.getLocRSAPubExp();
stmt = ( OracleCallableStatement )conn.prepareCall(
    "CALL hr.hr sec pkg.p_select_employees_sensitive(?,?,?,?,?,?,?,?,?,?)" );
...
```

Chapter Review

So, what have we got working so far? This whirlwind description will attempt to cover the secure programming ground traversed by this test application, `TestOracleJavaSecure`.

We call the `TestOracleJavaSecure` class without a two-factor authentication code for starts. The test application sets its application context in `OracleJavaSecure` by passing its inner class and application ID. Then (in a standard run mode) the test app calls `OracleJavaSecure.getAAConnRole()`.

Behind the scenes, in `OracleJavaSecure`, we proxy connect to Oracle Database as the OS user, proxying through the Oracle `appver` user. Initially, the login trigger for the `appver` schema checks to assure that the OS user is a valid Oracle user.

Once connected to Oracle database as `appver`, we test if the user passes our SSO tests. Then, because we didn't provide a two-factor authentication code, Oracle database generates a code and sends it to the devices and accounts registered for the OS user.

Once we receive the two-factor authentication code, we call `TestOracleJavaSecure` again, passing the two-factor code. The two-factor code gets passed to `OracleJavaSecure` when `TestOracleJavaSecure` sets the application context. Now when we call Oracle database as `appver` and pass SSO, we check the two-factor code; and if it passes, and the inner class from our app context matches something in Oracle, we return the list of encrypted connection strings that has been stored for this application.

Another behind-the-scenes process that occurred is our key exchange for application verification processes. The client sent his RSA public key to Oracle, and Oracle database generated a shared password key, encrypted it with the RSA key, and returned it to the client. The connection strings are encrypted with the shared password key.

Continuing the behind-the-scenes activity of the `getAAConnRole()` method, we decrypt the requested connection string and connect to Oracle database with it. We connect in order to request the secure application role needed by the application in order to read data from Oracle database. In an effort to acquire the secure application role, we once again assure that we pass SSO (perhaps on a different Oracle instance from `appver`), but we do not do two-factor authentication again for this connection.

`OracleJavaSecure.getAAConnRole()` will return an `OracleConnection`, which the developer can use to call one of his application procedures in order to get encrypted application data. When he makes that call, he will exchange encryption keys for the new connection, and will retrieve data encrypted with a new shared password key. We decrypt the aspects of the shared password key and build that key, and then we decrypt the data using that key.

Figures 10-1 and 10-2 illustrate this process. In Figure 10-1 we see the process of getting a list of application Oracle connection strings from Oracle database. This is a two-phase process, with phase one being the first connection when requesting a two-factor authentication code, and phase two being the user's return with a two-factor authentication code in hand. The two outlined boxes illustrate the activities of each of those phases.

At the outset (top) of Figure 10-1, we call the `setAppContext()` method, which simply stages data for the particular client application in the static class member variables of `OracleJavaSecure`. After that, the client application makes one additional call to `getAppConnections()`. Near the end of the second phase of that process, note that the list of connection strings is not returned to the client application, but rather is held in escrow within the client-side `OracleJavaSecure` class. The list is a private class member of `OracleJavaSecure`. You can also see within the second phase that before the connection strings are returned, they are encrypted with secret password key, and they are returned in encrypted form.

Along with the list of encrypted Oracle connection strings for the application, the encrypted artifacts of the DES shared password key are also returned from Oracle database. `OracleJavaSecure` on the client builds an equivalent shared password key for use in decrypting the application connection strings. After building the DES shared password key and storing the key for later use in clone or new static members, the `resetKeys()` method is called so that further Oracle connections as application users may be made.

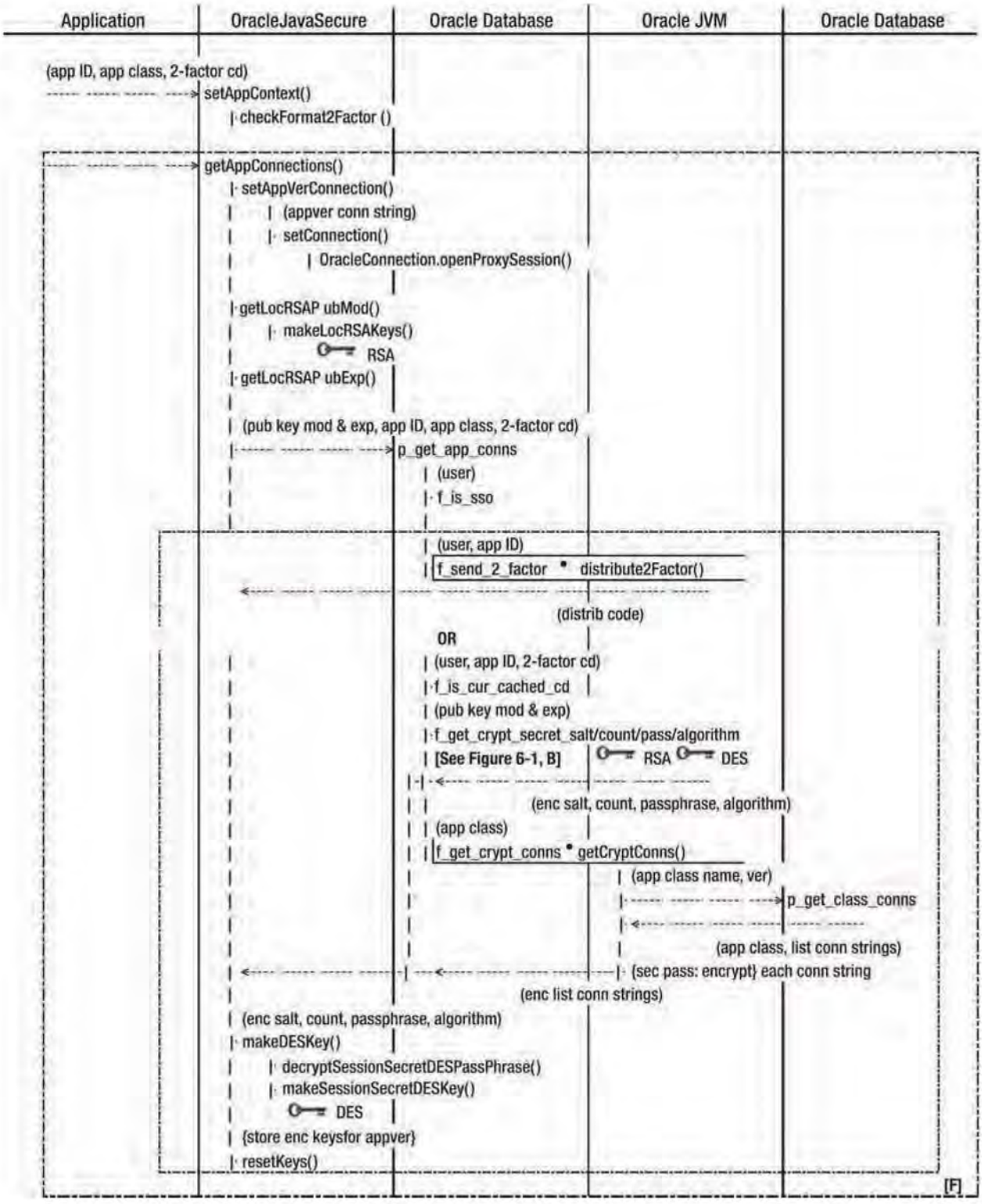


Figure 10-1. *Getting a list of application Oracle connection strings*

The top half of Figure 10-2 illustrates the process of storing Oracle connection strings in a list for the client application. All connection strings in the list on the client are maintained in encrypted form. The first call the client application may make, at the top of Figure 10-2 is a call to put a new connection string into the local list. This addition is only made locally on the client, and does not yet exist in the list stored on Oracle database. Notice that we call an encryption function using the secret password key to encrypt the connection string for storage.

The next call in Figure 10-2 that the client application may make is a call to `putAppConnections()`, which stores the entire list of connection strings in the Oracle database. This process illustrates best our use of different processing environments for different purposes. The individual columns of our illustration are informative. The client application (far left column) calls functions in `OracleJavaSecur`—this is Java calling Java. `OracleJavaSecure` calls Java stored procedures in the Oracle database, which passes the call through to the Oracle JVM—Java running on Oracle database. We also have the need from the Oracle JVM to call stored procedures and functions in Oracle (the far right column.) The two columns labeled "Oracle Database" are one and the same Oracle instance, just called from different directions.

In that process of storing the list of connection strings for the application in the Oracle database, you can see that we decrypt each connection string. We build a new `HashMap` of decrypted connection strings and we store them unencrypted in the Oracle database.

The bottom half of Figure 10-2 illustrates the acquisition of a specific Oracle connection for use by the application. The client application asks `OracleJavaSecure.getAAConnRole()` for a connection that uses a specific Oracle instance and user. Note that the connection string is only briefly decrypted for use in creating the new `OracleConnection`. The `getAAConnRole()` method not only creates the `OracleConnection`, but also sets the role to the secure application role needed for access to the sensitive application data. This `OracleConnection` with secure application role set is returned to the client application for use.

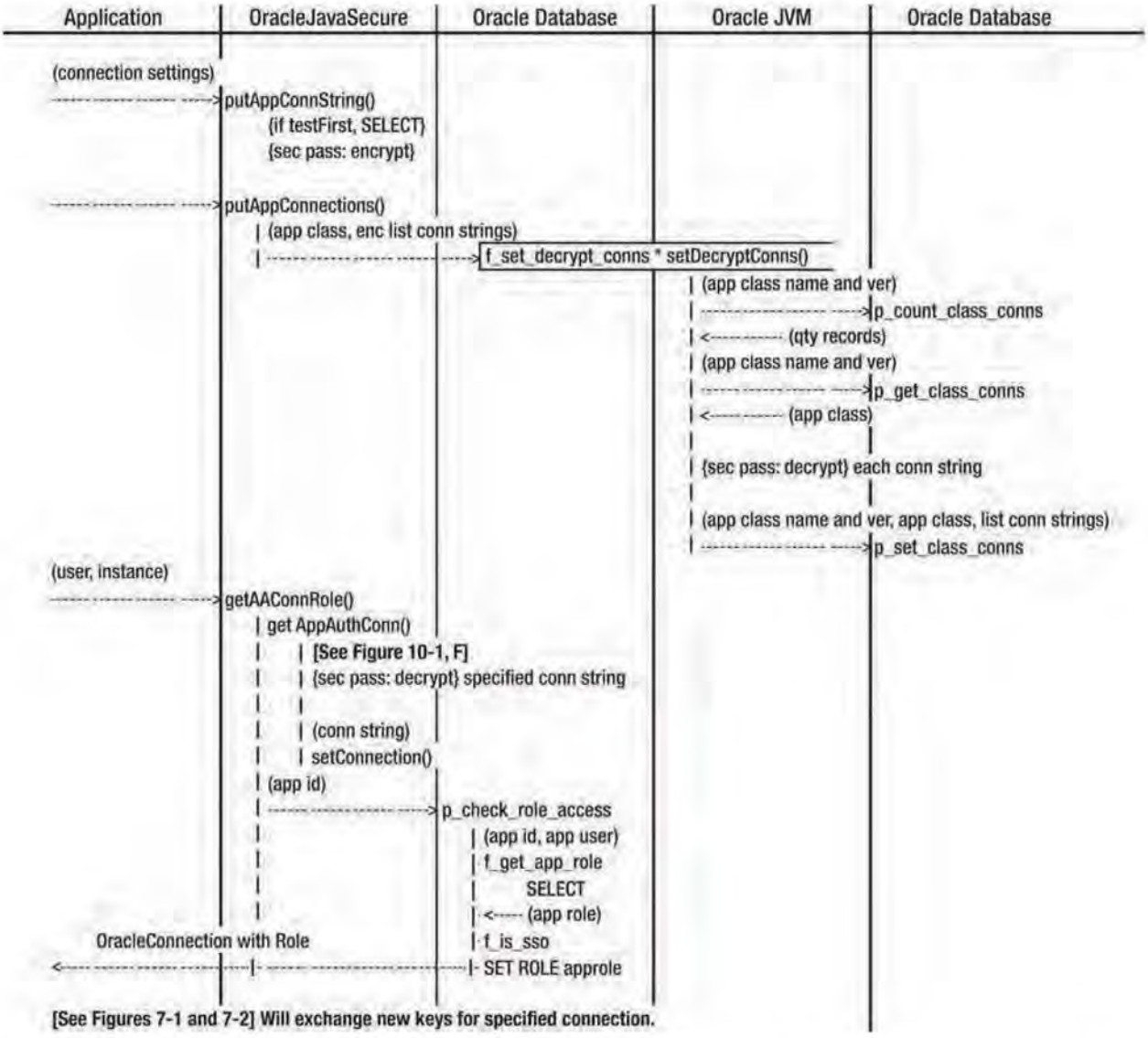


Figure 10-2. *Store the list of application connection strings in Oracle database, and get a specific application Oracle connection for use.*

CHAPTER 11

Enhancing Security

Every year, the president of my company gives us a pep talk. Generally, he talks about the good things we've done throughout the year as a company, and shares an itemized list of our failures or less-than-stellar moments. At the end of the talk, the president recognizes the skills and commitment of the employees and encourages us not only to continue doing what we're doing, but to do even more, taking on every new challenge.

We need to have this is the same pep talk regarding computer security. We are doing well and succeeding, but there are always new challenges and areas where we can do even more.

We have one specific area to address, which I've mentioned in earlier chapters. It has been a weakness in all our Oracle applications: we have embedded passwords. And this problem is not going away. We have removed all passwords from the code except the one for the application verification user, `appver`. Now we need to take additional steps to protect that password.

An additional weakness we have is risk of privileged user attack. Currently, if a rogue DBA (or hacker with DBA privileges) wants, he can read all our passwords in the `t app conn registry` table. Granted, he would have to know some Java to reconstitute the `HashMap`. Anyone with access to the backup tapes or archives might also be able to pull out the table and get access to all our application passwords. What we need is data encryption at rest; that is, data encryption in the Oracle table and on disk.

In the previous chapter, we produced code that will store application connection strings from any application running as any local user. This was convenient, but we need to formalize that process and delegate it to an application verification administrator. We will introduce the `appver admin` role for that purpose and discuss the process of managing our application connection strings.

If an attacker succeeds in getting the `appver` password, we want to severely limit what he can do and see in the database. We already have our login trigger, and `appver` only has access to a few procedures and limited data for our application security process. But we will see that every user in Oracle database has access to `PUBLIC` data that we would prefer to not expose, so we will attempt to tighten our general Oracle application security.

Hide the APPVER Connection String

We are not going to solve the embedded password problem, and you might be wondering why. In a nutshell, this is a "chicken or egg" problem. Perhaps we could hide our passwords in Oracle database, but we would need an Oracle password to request them from the database.

You cannot talk to Oracle database without a user/password, so the question arises of where to store the password. We have solved that issue for all our application passwords except for the gatekeeper account, `appver`. So let's look at some of the prospective solutions for hiding that password.

Get It from a Second Source/Server

Well, if we don't want the password in our code, and we can't get it from Oracle database (without a password), perhaps we need to store it and retrieve it from a secondary server. At the end of the day, I'm going to promote the idea of just sticking with Oracle database as our server. Supporting and securing one server well beats distributing your security roles across many platforms, primarily because it's easier to account for and easier to keep an eye on.

From a Port Server

Where I work, we have a port server (a multithreaded application that opens a `ServerSocket` and listens on a network port) that performs some of the same activities as the `appver` structures we described in Chapter 10 and returns connection strings. That was a brilliant solution, and one I can't take credit for—my associate conceived it. That server does just what it is intended to do, and delivers the connection strings securely. It was homegrown before J2EE.

We could implement some additional security in our port server, like SSO and two-factor authentication, things that are not there at present. However, the primary faults of this server are twofold: first, there is less-than-Oracle-quality security of the storage of the connection strings. Now, instead of an Oracle account to get access, there is an operating system account or group that gives access. And second, managing another kind of server with its unique configuration and code means potentially greater support requirements and unique knowledge.

From RMI

Remote Method Invocation (RMI) is a system for separating Java functionality in half. Some Java code runs on the client, and some on the RMI Server. Using RMI, you could retrieve your connection strings from a Java RMI service.

This approach has very much the same strengths and weaknesses as the port server described previously. Its one advantage is that it is an industry-standard approach rather than a home-grown port server, so perhaps there is less concern with unique requirements for support.

From a URLClassLoader

If you have ever run an Applet in a browser, you have seen the `URLClassLoader` in action. With this approach, some of your application code, perhaps that which retrieves and delivers your connection strings, is not stored on the client computer. This is a separation of code, but not a separation of processing. When needed, the local application pulls down jar files or class files from an HTTP (web) server and runs the code in those files. Your application is running stand-alone on the client computer, and provides no server-side authentication. Additionally, if an application can read the files from a URL, so can any other web client on the local machine.

Briefly, if we were running in a browser, we might have server-side authentication. There can be realm authentication from a browser to certain document/application realms, which generally means another username/password prompt to access the URL. Or in a browser, there might also be SSO if the web page or server directs the browser to authenticate the logged-in user to active directory (or similar) before returning the jar file (still talking about the URLClassLoader). But in the present discussion, let's stick to client applications.

Perhaps we could modify the standard behavior of the `URLClassLoader` and read code from a secure repository, and hence get our connection strings securely. Well, we're talking about pretty much a

combination of an RMI server (with the unique class loader) and a port server, again with the same weaknesses.

Weaknesses in All These Approaches

A computer hacker will do better to attack the client than the server. In Java, eventually all those connection strings are converted into a human-readable form in memory. No matter how fleeting and transitory, that is our biggest security weakness, and one that we have not solved. Our best efforts in the utility `OracleJavaSecure` class only succeed at keeping the connection strings out of the application code, and in keeping them encrypted until the moment they are needed, then discarding the unencrypted string.

It is beyond the scope of this book to look at an obvious protection from an attack on the client – that is connecting to Oracle database only from a server. Web applications are an example of this: the user sees the web page, but it is generated from data that is queried by the web server. We have mentioned RMI, and it could play a potential role in this. With RMI, the client may request data from the RMI server, and the RMI server would be responsible for querying Oracle database.

If you decide to run all your Oracle queries from a web services, HTTP, or RMI server, then you need to turn your attention to securing acquisition and use of the Oracle passwords on the server, which is an identical prospect, albeit more centralized, to what we have been discussing for client application security.

Get It from a Native Call: JNI

Perhaps we can protect our Oracle password by using a different programming language from Java for storing and reading our password, maybe even for querying Oracle database. We can write a procedure in C+ and call it from our Java application using Java native interface (JNI).

Granted, because Java is compiled into byte code that is interpreted by the JVM, the byte code is slightly more susceptible to decompiling back into source code than compiled C+ code. But code written in any language can be decompiled or disassembled. Dynamic link libraries (DLLs) and executable programs are just as open to attack.

I recommend just sticking with Java. For security, maintaining and applying one secure programming skill set well is better than being semi-fluent and semi-trustworthy in multiple languages.

Get It from an Encrypted Java Class

Okay, we're back to our client Java code, and we're trying to secure our embedded Oracle password. If we could encode our password so that only our program can read it, that would be good. But we'd have to hide how our program reads it to keep hackers from duplicating the process and reading the password.

The question has been asked, and we ask it here again, can we hide what we're doing in Java by encrypting our class files doing byte code encryption? The answer is yes, and an excellent description of the process and results was provided by Vladimir Roubtsov in his *JavaWorld* article, "Cracking Java byte-code encryption" from May 2003. He subtitles his article "Why Java obfuscation schemes based on byte-code encryption won't work."

The heart of Roubtsov's article describes his `EncryptedClassLoader`. If you recall, we discussed class loaders in the last chapter regarding realizing our application object and class in the Oracle database. We said that the `ClassLoader` cannot realize a class for which he has no prior knowledge, so we put a stub class on the Oracle database.

RMI has an `RMIClassLoader` that can load a class from a stub or skeleton class and match it with the implementation class on the RMI server. The `URLClassLoader` can load classes from a jar file or class files located remotely, for instance on a web server.

In the case of Roubtsov's `EncryptedClassLoader`, the byte code needs to be decrypted before the object or class is realized. That is all well and good, and we're all thinking green light!

However, Roubtsov points out the fallacy of trusting in this security. At some point in every class loader, the byte code is passed to the `defineClass()` method, which must present the class in a form readable by the JVM. It is at that point that the form and function of our classes are revealed to any Java hacker.

I'm not saying that encrypted byte code is a bad idea, just that we are not achieving meaningful encryption—just a complex obfuscation. Perhaps that can be said about all encryption: it is only encryption up to the point that it needs to be decrypted for use.

There are some concerns with implementing our own class loader: it is so close to the core of the JVM that we will need to take caution and may need to revisit it with every Java update. An error there in the `ClassLoader` could be devastating to our applications. That alone repels us from byte code encryption.

Get It from an Encrypted String

Why don't we just encrypt our password and only decrypt it as needed? That's a good idea, but what about the encryption key, how do we protect that? Is this another "chicken and egg" problem? I think so. Exposing the key is tantamount to exposing the password.

Get It from an Encoded String

What if we encode our string so that it is unrecognizable? This is perhaps as good as encrypting it, but it doesn't require an encryption key. We will look at some example Java code in a class we will call `OJSCode` to encode our password.

Using the Encode Method

In brief, our `OJSCode.encode()` method will take the connection string and encode it byte by byte by doing a binary Exclusive Or (XOR) with other bytes. An XOR conversion can be done twice to get back to the original byte. In XOR, the resultant bit is 1 only if the analogous bit in one or the other, but not both original bytes is 1. This is what the process looks like:

```
Original Byte   0 1 0 0 1 1 0 1
Other Byte      1 1 1 0 0 1 0 1
Result of XOR   1 0 1 0 1 0 0 0 (resultant bit is 1 where only one of bits is 1)
Other Byte      1 1 1 0 0 1 0 1
Result 2nd XOR  0 1 0 0 1 1 0 1 (notice this is the same as our original Byte)
```

For security reasons that we'll discuss later, we will reach outside the `OJSCode` class to get our "other" bytes. Our intent for `OJSCode` is only to use it from the `OracleJavaSecure` class, so how about we reach back to that expected class to get our other bytes? We will assemble our other bytes from a static string named "location" in `OracleJavaSecure`.

Because we know we are going to be doing an XOR on our original bytes, we will need to have the same number of other bytes as we have original bytes. Listing 11-1 shows how we get two byte arrays representing our connection string (`encodeThis`) and an array of other bytes of the same length.

Listing 11-1. *Get Two Same-Length Byte Arrays for XOR Encoding*

```
{ static String location = "in setAppVerConnection method."; }

    String location = OracleJavaSecure.location;
    byte[] eTBytes = encodeThis.getBytes();
    int eTLength = eTBytes.length;
    while( eTLength > location.length() ) location += location;
    String xString = location.substring( 0, eTLength );
    byte[] xBytes = xString.getBytes();
```

First, we get a local copy of the `location` string from `OracleJavaSecure`. It was defined with default (package) scope so code outside of the `orajavsec` package cannot see it. We concatenate that to itself sufficient times to get a string equal in length or longer than `encodeThis`. Then we get the substring of equal length to `encodeThis`.

To do byte XOR, we simply use the "^" operator. In Listing 11-2, we walk through both the original and other byte arrays and do an XOR on each byte pair, casting the `bytes` as `ints` for this process. We cast the result as a `byte` and store it in the `code[]` byte array.

Listing 11-2. *Byte-by-Byte XOR encoding*

```
    byte[] code = new byte[eTLength];
    for( int i = 0; i < eTLength ; i++ ) {
        code[i] = (byte)( (int)eTBytes[i] ^ (int)xBytes[i] );
    }
```

Our connection string resembles this:

"`jdbc:oracle:thin:appver/password@localhost:1521:orcl`"

We want to store an encoded replacement for that in the `OracleJavaSecure` class—something that will fit between quotation marks and be handled as a string. So the `OJSCode.encode()` method will return a String, but not just any string. Strings can contain unprintable characters such as carriage returns (character 13, decimal) and beeps (character 7). Those are difficult to represent as symbols between quotation marks.

When we do the XOR on our original bytes, we will end up with some unprintable characters, so we need a process to convert them all to readable characters without losing any fidelity, without losing information. We shouldn't need to solve this problem from scratch—we are not the first folks to have this concern.

An industry-standard approach for representing unprintable characters as strings is by Base64 encoding the characters. In Base64 encoding, the bits from a string of bytes is concatenated and broken up into 6-bit sequences that are translated into characters from a table of 64 printable characters. One or more "=" signs are often seen at the end of the Base64 encoded string as a padding character (for when the 6-bit values extend across an 8-bit boundary). When a string of those printable characters is seen with an equals signs at the end, it is a clear indication of Base64 encoding, and a give-away as to the process used to do encoding, if that's a concern. This is not necessarily a problem, because Base64 encoding is only intended to make extended characters palatable to web browsers and such where regular alphanumeric characters are about all that is expected. However, if we are additionally trying to disguise text, then Base64 encoding fails. Base64 encoding is surprisingly often used to convey passwords for basic authentication to proxy servers and web server realms, as if it were encryption. Listing 11-3 shows the Java call we make to do Base64 encoding.

Listing 11-3. *Base64 Encoding*

```
String decodeThis = (new BASE64Encoder()).encode( code );
```

Oracle has included Base64 encoding in the JVM; however, it is a bother to use. We can call upon **sun.misc.**BASE64Encoder to do our encoding, but look at the package it is in. This is a part of the JVM, but is proprietary. When we compile code that uses that class, we get this warning message:

```
sun.misc.BASE64Encoder is Sun proprietary API and may be removed in a future release
```

We can acquire the Base64 algorithm form many free sources, if we want. But fortunately, Base64 encoding is not the only game in town. We can create our own printable encoding. Look at Listing 11-4. Let's take every int resulting from the XOR and get the String representation in hex format by calling Integer.toHexString(). Our hex strings will be either one or two characters in length, ranging from 0 (0x00) to fe (0xfe) in hexadecimal (0 to 254 in decimal). When the number is less than 0x10, it will have only one character, but we want to know how to break up our encoded string, so we will prepend those hex strings with a "0" character. That way each byte is represented by two characters, and we know how to parse and decode each character. We might call this Padded Hex XOR Coding.

Listing 11-4. *Padded Hex XOR Encoding*

```
StringBuffer sBuf = new StringBuffer();
String oneByte;
for( int i = 0; i < eTLength ; i++ ) {
    oneByte = Integer.toHexString( (int)eTBytes[i] ^ (int)xBytes[i] );
    if( oneByte.length() == 1 ) sBuf.append( "0" + oneByte );
    else sBuf.append( oneByte );
}
String decodeThis = sBuf.toString();
```

Decoding this OJSCode'ed connection string is just the same process, in reverse. We call Integer.parseInt() on each pair of characters, using a *radix* of 16, so we convert from hex to decimal. Then we XOR the integer with the same other byte from our "location" string that we had used before. This gets us back to the original characters from which we rebuild our connection string.

Using Base64 encoding, the coded string is shorter. Here is an example (notice the telltale "=" sign padding at the end):

```
IE5DEgtTNVAUOUUGKw4aQkUnFR8KQA==
```

Our own encoding creates longer strings. This is the same string encoded with our algorithm.

```
204e43120b533550143945062b0e1a424527151f0a40
```

Obfuscating the Algorithm/Code

Our plan for OJSCode is to encode our appver password. In this case, our Java code is analogous to an encryption key. With a key, what we encrypt can be decrypted. With OJSCode, our encoded password can be decoded and read by anyone who has access to the OJSCode class.

If a hacker has this class but cannot see the OJSCode Java code, then she won't be able to reproduce the logic, and there is one hidden aspect that he won't be able to duplicate: we call out of the class to get the location string in the OracleJavaSecure class. That isn't much of a security feature, but if we hide it, it will be a major hindrance to the hacker.

What I am suggesting here is code obfuscation. The logic is hidden in plain sight. It is hard to know whether this produces much of an obstacle to reading our code, but it would require several conversion steps—the same steps we take to obfuscate our code, in reverse.

Obfuscate the Logic

One of my favorite aspects of Java code is its readability. I do compress my code with stacked method calls, and there is the object-oriented aspect to Java code that must be comprehended, but in all it is very readable. What if we make it our goal to make it less readable? The first step we can take is to use our hex encoding rather than the standard Base64. The rest of the ideas here are intended to give you an idea of what can be done to obfuscate code. Be aware that what makes your code harder for a hacker to understand will also make your code harder for you to understand and to maintain. For that reason, keep a copy of your original code in a secure location for your own reference.

We have a calculated field called `eTLength`, which is the length of our initial connection string, but we don't need a method member if we are willing to get the length of the string each time it's needed. We can add complexity by concatenating `Strings` rather than using a `StringBuffer`. It is not as efficient—in fact, it clutters memory with transient strings, but the logic is less obvious. Let's also convert the `for` loop to an eternal `do while true` loop. We will break the loop by breaking to a label, `GT`: we are creating the Java equivalent of a `GOTO` statement. We have a bit of inside knowledge about the expected input/output; the encoded string (using our padded hex encoding) will be twice as long as the connection string. Using that knowledge, we will break out of our loop when the encoded string is long enough; that is, twice the length of the connection string, modulus the length of the connection string equals 0. We know the next position in our original byte array that we need to deal with is the one at half the length of our encoded string as it is being concatenated together. There may be insufficient characters in our other byte array to XOR with the connection string bytes, so let's wrap around to the front of the "other" byte array by using modulus. We do that instead of concatenating `location` to itself until long enough. So far our obfuscated code is as shown in Listing 11-5.

Listing 11-5. *Obfuscated Logic, Step 1*

```
static String encode( String encodeThis ) {
    byte[] eTBytes = encodeThis.getBytes();
    byte[] xBytes = OracleJavaSecure.location.getBytes();
    String decodeThis = "";
    String oneByte;

    GT: do {

        oneByte = Integer.toHexString(
            (int)eTBytes[decodeThis.length()/2] ^
            (int)xBytes[( decodeThis.length()/2 ) % xBytes.length] );
        if( oneByte.length() == 1 ) decodeThis += "0";
        decodeThis += oneByte;
        if( ( ( decodeThis.length()/2 ) % eTBytes.length ) == 0 )
            break GT;
    } while( true );
    return decodeThis;
}
```

Looking at our code, we see a number of hard-coded integers, most of them equal to 2. Let's create a member integer that by complex calculation works out to be 2 and use it in pace of those integers. We

will also capitalize on the fact that our integer divided by itself is 1 and subtracted from itself is 0. Once we get the bytes of the original connection string, we can reuse that original connection string as our return string. Several places in `OJSCode` we get a byte array from a String by calling the `String.getBytes()` method. Let's rewrite that as a method called `traverse()` to replace `String.getBytes()`. We will allow `traverse()` to take a `null` and to get the bytes of `location` string in that case. Further, we will create an additional member in `OracleJavaSecure`, named lower case `l` (ell) that we will set equal to `location`, and we will point at that instead. Now our code looks like Listing 11-6.

Listing 11-6. *Obfuscated Logic, Step 2*

```
static String encode( String encodeThis ) {
    byte[] eTBytes = traverse( encodeThis );
    byte[] xBytes = traverse();
    encodeThis = "";
    String oneByte = "*";
    int twoI = Integer.parseInt(
        String.valueOf( Integer.toHexString(
        (int)(oneByte.charAt(twoI - twoI))).charAt(twoI - twoI)));
    GT: do {
        oneByte = Integer.toHexString(
            (int)eTBytes[encodeThis.length()/twoI] ^
            (int)xBytes[( encodeThis.length()/twoI ) %
            xBytes.length] );
        if( oneByte.length() == ( twoI/twoI ) )
            encodeThis += "0";
        encodeThis += oneByte;
        if( ( ( encodeThis.length()/twoI ) % eTBytes.length )
            == ( twoI - twoI ) )
        {
            System.arraycopy( xBytes, twoI - twoI,
                eTBytes, twoI * 0, twoI );
            break GT;
        }
    } while( true );
    return decodeThis;
}

static byte[] traverse( String encodeThis ) {
    int twoI = 0;
    if( encodeThis == null )
        encodeThis = OracleJavaSecure.l;
    byte[] eTBytes = new byte[encodeThis.length()];
    do eTBytes[twoI] = (byte)(encodeThis.charAt(twoI++));
    while( twoI < eTBytes.length );
    return eTBytes;
}
```

We have also added some misdirection. Now, in the final `if` statement of our `do while` loop, before we break to the `GT` label, we do an `arraycopy()`. We copy the first two bytes from our "other" byte array to the beginning of our connection string bytes. This is essentially meaningless, because we are through dealing with those arrays; however the tactic of misdirecting is often used in obfuscating code.

Obfuscate the Naming

Let's take just one more step—although, numerous other obfuscation processes could be accomplished. This step is merely replacing our member names with meaningless names that are difficult to read. Let's swap names according to Table 11-1. These sequences of the number 1 (one), capital I, and lowercase l (ell) are selected to be confusing. They all start with an alphabet character to comply with the compiler. The greatest loss in this step of obfuscation is the loss of meaningful names for these methods and members.

Table 11-1. *Obfuscated Naming*

Previous Name	Obfuscated Name
OJSCode	OJSC
encode()	x()
decode()	y()
traverse()	lI1ll()
encodeThis	I1ll1
eTBytes	lII1l
xBytes	ll1I1
oneByte	ll11I
twoI	IlIl1
GT	I11lI

After these translations, our code is barely recognizable. Only the code structure looks like Java, and a few general method references. Listing 11-7 shows the encode() [x()] and traverse() [lI1ll()]methods.

Listing 11-7*. Obfuscated Naming*

```
public class OJSC {
    static String x( String I1ll1 ) {
        byte[] lII1l = lI1ll( I1ll1 );
        byte[] ll1I1 = lI1ll( null );
        I1ll1 = "";
        String ll11I = "*";
        int IlIl1 = Integer.parseInt(
            String.valueOf( Integer.toHexString(
            (int)(ll11I.charAt(0))).charAt(0)));
        I11lI: do {
            ll11I = Integer.toHexString(
```

```
            (int)lII1l[I1ll1.length()/IlIl1] ^
            (int)ll1I1[( I1ll1.length()/IlIl1 ) %
            ll1I1.length] );
        if( ll11I.length() == ( IlIl1/IlIl1 ) )
            I1ll1 += "0";
        I1ll1 += ll11I;
        if( ( ( I1ll1.length()/IlIl1 ) % lII1l.length )
            == ( IlIl1 - IlIl1 ) )
        {
            System.arraycopy( ll1I1, IlIl1 - IlIl1,
                lII1l, IlIl1 * 0, IlIl1/IlIl1 );
            break I11lI;
        }
    } while( true );
    return I1ll1;
}

static byte[] lI1ll( String I1ll1 ) {
    int IlIl1 = 0;
    if( I1ll1 == null )
        I1ll1 = OracleJavaSecure.l;
    byte[] lII1l = new byte[I1ll1.length()];
    do lII1l[IlIl1] = (byte)(I1ll1.charAt(IlIl1++));
    while( IlIl1 < lII1l.length );
    return lII1l;
}
```

Generating an Encoded APPVER Connection String

We will add code to the `main()` method of `OracleJavaSecure`, as shown in Listing 11-8, to accept the password for `appver` as a command line argument and encrypt a connection string for `appver`. We check the format of the argument to see if it is possibly a two-factor authentication code. If it's not, then we call the obfuscated code in `OJSC` to do encryption and decryption for display. This uses the additional `appver` connection string attributes defined here.

Listing 11-8. *Utility Encoding of appver Password and Connection String*

```
if( args.length != 0 && args[0] != null ) {
    String encodeThis = args[0];
    if( ! encodeThis.equals(checkFormat2Factor(encodeThis)) ) {
        encodeThis = "jdbc:oracle:thin:appver/" + encodeThis +
            "@localhost:1521:orcl";
            //"@localhost:1521:apver"; // for use later in Chapter 11
        String encoded = OJSC.x( encodeThis );
        System.out.println( encoded );
        encodeThis = OJSC.y( encoded );
        System.out.println( encodeThis );
    }
} else System.out.println(
```

```
        "You may enter APPVER password on command line." );
```

The intention at this point is that you would call `OracleJavaSecure` once with your intended `appver` password, then copy the encoded connection string into the code of *OracleJavaSecure.java* in the `setAppVerConnection()` method.

Hard Coding the Encoded APPVER Connection String

We will modify the `setAppVerConnection()` method by hard-coding our encoded `appver` connection string, and by calling the `OJSC.y()` method to decode it when calling `setConnection()`. (See Listing 11-9.) By decoding the connection string at the last moment when calling `setConnection()`, and not setting a member variable to the decoded value, we minimize the attack profile and duration when a hacker may view the clear-text password.

Listing 11-9. *Hard-Coding the Encoded appver Connection String*

```
    String prime =
"030a42105f1b3311133a0048370707005f020419190b524215151b1c13411b0a601f0a17201c18391606795e5b5c5
4591b1b0c02"
    setConnection( OJSC.y( prime ) );
    appVerConn = conn;
```

Create an Oracle Client Wallet

There is a standard way for obfuscating/encrypting passwords stored on the client computer that is provided in the Oracle client. It is called the secure external password store, but it is better known as the wallet. Because wallets can be used on both the server and the client, we should call this the client wallet. I hope to give you a good overview of Oracle client wallets, but we will not end up using them in our security infrastructure.

The first thing about client wallets that you should know is that, by default, they must exist in each OS user's home directory. Client wallets are considered to be personal stores for Oracle credentials. This is quite the opposite of what I'd like to achieve, which is no Oracle user credentials on the desktop. And now, if we mix our `appver` user password in with potentially other additional user passwords, there is no practical way we can centrally manage it, e.g., update the password.

The second thing you should know about Oracle client wallets is that they can be copied and used by anyone who possesses them. I can create a client wallet and put the `appver` password in it, then copy it to every computer user' s home directory, and they can all use the wallet to log in as `appver`. This might appear to be a way to do centralized distribution of password updates. Perhaps, but it is also problematic. These wallet files can be given away like candy, e-mailed to buddies, stolen from backup tape—whatever. That is not protection. The assumption is that the OS is providing access control to user home directories and can protect the files.

The third thing to know is that client wallets are protected from modification and viewing by a wallet password, but there is no authentication required in order to use the client wallet if you have the files. If we place our `appver` password in a client wallet, then literally anyone who has the wallet files can connect to Oracle database as `appver`.

Install the Oracle Client

The Oracle client is included with the Oracle database installation, but you can also download and install the Oracle client separately. Having the Oracle client installed on a separate computer is the best way to test your client applications, rather than running them on the Oracle database, which you can also do. In the following discussions, we will assume that you have set your `ORACLE HOME` environment variable to something that resembles one of these:

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

SET ORACLE HOME=C:\app\oracle\product\11.2.0\client 1
```

Create the Wallet

The first thing you will do when you decide to place your connection passwords in a client wallet is pick a location for the wallet. While there is no restriction on where wallets can be created, when you go to use the wallet to connect to Oracle database, the presumed location is within the current OS user's home directory. You can create your client wallet in any secure location, but you might as well create it where it needs to be for your own use. Otherwise, you'll need to copy it to your home directory for your use and testing.

The expected wallet location is in a subdirectory with the same name as the OS user ID in the user's home directory. For example, if the user's OS user ID is FredF, then we will create a `fredf` directory in his home directory and create or copy the wallet files there. We will use FredF as the OS username in our examples. Create a directory for your client wallet.

```
mkdir C:\Users\FredF\fredf
```

■ **Note** In Windows XP, you will substitute *Documents and Settings* for *Users* in these commands.

To create your password store, you will issue commands to the *mkstore* utility. One command will create the store and one will create the password credential. In each of these commands, you will point to the directory location for your wallet files. The first *mkstore* command creates the wallet. It will prompt you to enter a wallet password that meets certain complexity rules.

```
%ORACLE HOME%\bin\mkstore -wrl C:\Users\FredF\fredf -create
```

The second command creates the encrypted password/credential. The command will prompt you to enter the password for the `appver` user twice; then it will prompt you to enter the password for the wallet—that is the same password you entered for the first *mkstore* command. While we are here, let's also create an entry in the client wallet for the `appusr` user:

```
mkstore -wrl C:\Users\FredF\fredf -createCredential orcl_appver appver
mkstore -wrl C:\Users\FredF\fredf -createCredential orcl_appusr appusr
```

There are several commands you may use to view the contents of your client wallet. You will need to enter the wallet password to use these commands:

```
mkstore -wrl C:\Users\FredF\fredf -list
mkstore -wrl C:\Users\FredF\fredf -listCredential
```

```
mkstore -wrl C:\Users\FredF\fredf -viewEntry oracle.security.client.connect string1
mkstore -wrl C:\Users\FredF\fredf -viewEntry oracle.security.client.username1
mkstore -wrl C:\Users\FredF\fredf -viewEntry oracle.security.client.password1
```

Notice that we give the name of the first credential as `orcl appver` – that stands for the `appver` user on the *orcl* Oracle database instance. We need to make some additions to the *sqlnet.ora* and *tnsnames.ora* files that configure the *TNSNames* (transparent network substrate) search for database instances. TNS is to *SQLnet* (Oracle's database network communication protocol) as domain name services (DNS) is to TCP/IP. TNS allows us to have multiple names (aliases) for a single Oracle database instance and to write our applications with references to aliases that can at various times point at different Oracle instances. This flexibility is a primary reason for naming services; the other primary reason is to do remote lookup (not storing all names and addresses locally) and its corollary reason: centralized management of name/address associations. Of course we need to use LDAP or something similar for our TNSNames service to achieve that second goal.

There are many features of the TNSNames service that we are not covering in this book. Please refer to the Oracle Database Net Services Reference book for more information.

The *sqlnet.ora* and *tnsnames.ora* files exist or need to be created in a specific directory of the client. Both of them are in a directory like *%ORACLE_HOME%\network\admin*, depending on your installation. Add the line in Listing 11-10 to your *sqlnet.ora* file. For a basic client wallet installation, you only need to specify the `WALLET OVERRIDE` directive. You could also specify the `WALLET LOCATION` directive, but it would most likely be unused. I have found that the format for the `WALLET LOCATION` directive is a bit sensitive; while (for specific Oracle client versions) drive letters are allowed, quotation marks and a trailing "\" character are not. Also note that wallets created by the Oracle 11g client cannot be used with Oracle 10g clients, but 10g wallets can be used in 11g clients.

Listing 11-10. *Addition to Client sqlnet.ora File for Wallet*

```
SQLNET.WALLET OVERRIDE=TRUE
```

■ **Caution** Placing the `WALLET OVERRIDE` directive in a **server** *sqlnet.ora* file (e.g., *%ORACLE_HOME%\NETWORK\ADMIN\sqlnet.ora*) can keep the Oracle database from responding to client connections. My suggestion, if you are testing the client wallet on the same computer as the Oracle database, is that you start the database without `WALLET OVERRIDE` in the *sqlnet.ora* file, and then add that directive *temporarily* when testing the client wallet.

Add the lines in Listing 11-11 to your client *tnsnames.ora* file. The first section is a standard TNSNames entry for an Oracle instance. For each password we enter in the wallet, we will need an additional entry in *tnsnames.ora*. If you have made entries in *tnsnames.ora* before, but never used the wallet, this may seem a bit strange to you. But consider that you are giving the password for a specific user for each credential in the wallet, so you are coordinating that password with an entry in *tnsnames.ora*. For example, `orcl appver` is an entry in *tnsnames.ora* that is specifically for use by the `appver` user.

Listing 11-11. *Additions to Client tnsnames.ora File for Wallet*

```
orcl =
        (DESCRIPTION=
                (SOURCE ROUTE=YES)
                (ADDRESS=(PROTOCOL=tcp)(HOST=orcl.org.com)(PORT=1521))
                (CONNECT DATA=(SERVICE NAME=ORCL)))

orcl appver =
        (DESCRIPTION=
                (ADDRESS=(PROTOCOL=tcp)(HOST= orcl.org.com)(PORT=1521))
                (CONNECT DATA=
                        (SERVER=DEDICATED)
                        (SERVICE NAME=ORCL)
                        (SID=ORCL)))

orcl appusr =
        (DESCRIPTION=
                (ADDRESS=(PROTOCOL=tcp)(HOST= orcl.org.com)(PORT=1521))
                (CONNECT DATA=
                        (SERVER=DEDICATED)
                        (SERVICE NAME=ORCL)
                        (SID=ORCL)))
```

■ **Note** At least the HOST name will be different in your case, so modify those settings.

Use the Wallet from SQL*Plus

This is the slickest aspect of using the wallet. You can connect to Oracle database without entering the password, presuming the wallet is created and placed where needed and that your configuration files are correct. Just enter this SQL*Plus command to connect:

```
%ORACLE HOME%\bin\sqlplus /@orcl appusr
```

Keep in mind that the password is acquired somehow through the wallet; therefore, it still exists on the client. One might examine the computer memory during processing to capture the clear text password as it is acquired from the wallet. One might also examine the Java code, obfuscated Java code, DLLs, and obfuscated DLLS used by Oracle database to maintain and use the wallet, thereby revealing the process for independently deciphering the wallet password. I don't pretend to know how difficult that would be.

Use the Wallet from Java

Using the client wallet for authentication in a Java application requires an additional jar file in your `CLASSPATH`. The *oraclepki.jar* file is required for reading the wallet files. On the command line, you can run a test in the *Chapter11/wallet* folder.

```
java -cp %CLASSPATH%;%ORACLE HOME%\jlib\oraclepki.jar TestWallet
```

In the *TestWallet.java* file, you'll find a `main()` method with the code shown in Listing 11-12. You may need to modify this code to run the test.

Listing 11-12. *Configure Java to Use Client Wallet*

```
System.setProperty("oracle.net.tns admin",
    "C:/app/oracle/product/11.2.0/client 1/NETWORK/ADMIN");
Properties info = new Properties();
String username = System.getProperty( "user.name" );
info.put("oracle.net.wallet location",
    "(SOURCE=(METHOD=file)(METHOD DATA=(DIRECTORY=C:/Users/" +
    username + "/" + username + ")))");
```

We set the `tns admin` system property so that we can find the *tnsnames.ora* file, and the connect identifier, `orcl appusr` inside that file. We instantiate a `Properties` object that we will pass when we get the `Connection`. One of the properties we need to set is the `wallet location` property. You'll recall that we created or copied the wallet files into the user's home directory in a directory by the same name as the user. In order to set that wallet property, we get the `user.name` `System` property, and concatenate a directory location from that value.

We use an `OracleDataSource` class to get our connection as shown in Listing 11-13. We saw `OracleDataSource` in Chapter 8 when we looked at single sign on with one of the varieties of pooled connections. In this case, we set the connection URL to use our wallet connect identifier, `orcl appusr`. We also pass in the `Properties` class we instantiated earlier, and we call the `getConnection()` method.

Listing 11-13. *Get a Connection in Java Using Client Wallet*

```
OracleDataSource ds = new OracleDataSource();
ds.setURL("jdbc:oracle:thin:@orcl_appusr");
ds.setConnectionProperties(info);
Connection c = ds.getConnection();
```

In *TestWallet.java*, we also set up a proxy connection, identical to what we are using in `OracleJavaSecure`. For that reason, only OS users who can do SSO (who have matching Oracle users) can run `TestWallet`.

Administer Wallet Security

A client wallet might be the perfect solution for protecting our application verification `appver` user password. I would encourage you to use it in this one instance, except for my reservations.

True, the wallet protects the password from being read by hackers, probably better than anything we have developed so far in this book. It works as described—once you get it working. It is easy to copy around to each person's home directory for his or her use. Additionally, we have already portrayed the `appver` password as little more than data: a gatekeeper or bouncer for the real application accounts.

Yet some of the positives of the wallet have their dark side. Anyone who has the wallet files can connect to Oracle database as the user specified. That connection doesn't have to happen through your application; it can happen through a SQL*Plus session, as we saw. With the `appver` user, that is not much of a concern—but with most any other Oracle database user, it is a big problem. Don't only think of legitimate computer users, but also (for example) those users who get access to your offsite backup repositories and glean the wallet files from there. It is assumed that protection of the wallet files is

incumbent on the OS permissions, but that is not reliable—what if a legitimate user e-mails the files to a hacker posing as a computer support technician?

Another concern is just a practical one that has to do with file distribution, management, and updates. Possibly you already have a system for that, but getting the files to each user's home directory and keeping them updated is not a trivial concern.

Finally, a concern that is not mentioned in regard to client wallets and security but follows logically from an examination of the files used to configure wallet authentication is the disconnect of the password from the host/Oracle database instance specification. A user might try substituting other Oracle instances for the one you have specified in the tnsnames.ora file; thereby testing a specific wallet user and password against every instance. For example, if a hacker changes the tnsnames.ora file on his machine so that the `orcl appusr` entry points to `(SID=TestOrcl)`, he can attempt to connect to that instance as `appusr` with this same command:

```
sqlplus /@orcl appusr
```

■ **Caution** With the Oracle client wallet files, an attacker can connect as the specified user you configured on any instance where that user exists and has the same password.

Perhaps your users only exist on one instance, or have only the exact same privileges on each instance; however, it is more likely that a user may have extended privileges on lower priority (sandbox, development, or acceptance) regions than on production. Those extended privileges on any instance may provide additional attack vectors.

Trace Oracle Client Code

Often you will have the experience during development that things are not working, and your application can't tell you what's wrong. The error is happening in the underlying protocol somewhere, and is hidden or obscured. You may even see misleading error messages.

At that point, you may have to call in the reserves, asking your network administrators to put a network sniffer on your subnet and capture packets to analyze; hopefully, you'll spot the problem.

However, before it comes to that, you have some options yourself, when dealing with Oracle database. You can turn on trace in your client to see that underlying protocol dialog between the client and the Oracle database. Simply set the trace level in the client *sqlnet.ora* file. A value of 16 is the maximum detail trace; you can also choose less detail in levels 8, 4, 2, and 1.

```
TRACE LEVEL CLIENT=4
```

■ **Caution** When you are done troubleshooting your current problem, be sure to disable trace logging. It generates lots of files with potentially lots of data that can be both a hog of disk space and revealing in a security sense. At higher levels of trace logging, the data returned from a query is also presented in the trace file.

One of the default locations for trace files (there are several) is a folder named like this:

```
%ORACLE HOME%\log\diag\clients\user UserID\host ########## ##\trace
```

On the Oracle database, you may have to make the basic directory tree and grant all users the privilege to write to it.

```
mkdir %ORACLE HOME%\log\diag\clients
```

Here is an example of what you would see with level 4 trace enabled (only the first few lines are shown):

```
Trace file D:\app\oracle\product\11.2.0\dbhome 1\log\diag\clients\
user UserID\host ########## ##\trace\ora 2988 5952.trc
2011-07-09 07:22:35.826970 : --- TRACE CONFIGURATION INFORMATION FOLLOWS ---
2011-07-09 07:22:36.058905 : New trace stream is
D:\app\oracle\product\11.2.0\dbhome 1\log\diag\clients\
user UserID\host ########## ##\trace\ora 2988 5952.trc
2011-07-09 07:22:36.058968 : New trace level is 4
2011-07-09 07:22:36.059012 : --- TRACE CONFIGURATION INFORMATION ENDS ---
2011-07-09 07:22:36.059057 : --- PARAMETER SOURCE INFORMATION FOLLOWS ---
2011-07-09 07:22:36.059105 : Attempted load of system pfile source
D:\app\oracle\product\11.2.0\dbhome 1\network\admin\sqlnet.ora
2011-07-09 07:22:36.059203 : Parameter source loaded successfully
2011-07-09 07:22:36.059241 :
2011-07-09 07:22:36.059275 : Attempted load of local pfile source
C:\OraJavSecure\Chapter11\wallet\sqlnet.ora
2011-07-09 07:22:36.059308 : Parameter source was not loaded
2011-07-09 07:22:36.059337 :
2011-07-09 07:22:36.059367 : -> PARAMETER TABLE LOAD RESULTS FOLLOW <-
2011-07-09 07:22:36.059402 : Successful parameter table load
2011-07-09 07:22:36.059435 : -> PARAMETER TABLE HAS THE FOLLOWING CONTENTS <-
2011-07-09 07:22:36.059472 :  TRACE LEVEL CLIENT = 4
2011-07-09 07:22:36.059504 :  SQLNET.WALLET OVERRIDE = TRUE
...
```

Logging Oracle Thin Client Trace Data

When you are using Java thin ojdbc drivers, you cannot configure trace by the setting in `sqlnet.ora`. You will need to use the logging features of the ojdbc drivers. To accomplish this, you need to place a different ojdbc drivers jar, *ojbdc6_g.jar*, on your `CLASSPATH` (in front of or as a replacement for *ojdbc6.jar*). This alternate, logging jar file is available in the Oracle Client directory %ORACLE_HOME%\jdbc\lib, or on the Oracle downloads web site:

http://www.oracle.com/technetwork/indexes/downloads/index.html.

Change directories to wallet. To run the `TestWallet` class using this driver file (and the wallet), you can run a single Java command line. Trace logging and client wallet are unrelated; we are just using `TestWallet` as a convenient example to show trace logging.

```
cd Chapter11/wallet

java -Doracle.jdbc.Trace=true
    -cp .;%ORACLE HOME%\jdbc\lib\ojdbc6 g.jar;%ORACLE HOME%\jlib\oraclepki.jar
    -Djava.util.logging.config.file=OracleLog.properties TestWallet > temp.txt 2> temp2.txt
```

Several things about that command line are worth mentioning. The directive `-Doracle.jdbc.Trace=true` turns on trace logging. Notice the *ojdbc6_g.jar* in the `CLASSPATH, -cp` directive. Using that jar file enables logging. We also have the *oraclepki.jar* file on the `CLASSPATH` for enabling the wallet.

The `-Djava.util.logging.config.file=OracleLog.properties` directive tells the logging classes to find their properties settings in a file named *OracleLog.properties*. An example properties file for logging may have the entries shown in Listing 11-14.

Listing 11-14. *Configure Java (ojdbc) Trace Logging, OracleLog.properties*

```
handlers = java.util.logging.ConsoleHandler
java.util.logging.ConsoleHandler.level = ALL
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter
.level=CONFIG
oracle.jdbc.level = FINE
oracle.jdbc.connector.level = FINE
oracle.jdbc.driver.level = FINE
oracle.jdbc.pool.level = FINE
oracle.net.ns.level = TRACE 20
```

■ **Note** You will find a complete configuration properties file at *Chapter11/wallet/OracleLog.properties.*

The first three property lines configure the console (command prompt window) as a logging handler. The `ConsoleHandler.level = ALL` directive indicates that we want everything that's generated to go to the console. That is my preference. Then, if there's too much logging data coming at me, I can redirect it to a file. The last portion of the command line shows a redirect of the two standard output streams of data, which normally come to the command prompt window, `> temp.txt 2> temp2.txt`. The first greater-than symbol redirects the "standard out" stream to a file named *temp.txt* in the current directory. The second greater-than symbol, with the "2" prefix redirects the "standard err" stream to a file named *temp2.txt*. You can send both these streams to the same file with this directive at the end of a command: `> temp.txt 2> &1.`

The last few lines of the properties file configure the level of detail of trace logging. The first property, `.level=CONFIG`, sets a default level of logging for all aspects of the ojdbc classes to use. `CONFIG` level is medium detail. The other levels are set at `FINE` and `TRACE 20`, which are more detailed levels. The example *OracleLog.properties* file lists of all the aspects to which logging applies, and all the levels that can be set.

With the setting provided in *OracleLog.properties*, a very large trace output is generated in temp2.txt. The following lines are just a snippet of the output produced:

```
Jul 9, 2011 8:32:37 AM oracle.jdbc.pool.OracleDataSource <init>
TRACE 1: Public Enter:
Jul 9, 2011 8:32:37 AM oracle.jdbc.pool.OracleDataSource <init>
TRACE 1: Exit
Jul 9, 2011 8:32:37 AM oracle.jdbc.pool.OracleDataSource setURL
TRACE 1: Public Enter: "jdbc:oracle:thin:@orcl appusr"
Jul 9, 2011 8:32:37 AM oracle.jdbc.pool.OracleDataSource setURL
TRACE 1: Exit
```

```
Jul 9, 2011 8:32:37 AM oracle.jdbc.pool.OracleDataSource setConnectionProperties
TRACE 1: Public Enter: {oracle.net.wallet location=(SOURCE=(METHOD=file)
(METHOD DATA=(DIRECTORY=C:/Users/OSUSER/OSUSER))), oracle.net.encryption types client=AES192}
Jul 9, 2011 8:32:37 AM oracle.jdbc.pool.OracleDataSource setConnectionProperties
TRACE 1: Exit
...
 00 00 00 00 00 53 45 4C     |.....SEL|
 45 43 54 20 55 53 45 52     |ECT.USER|
 20 46 52 4F 4D 20 44 55     |.FROM.DU|
 41 4C 01 01 00 00 00 00     |AL......|
```

The last few lines are a portion of a packet sent to Oracle database, showing the query that was used. The entire contents of every packet in both directions are included in the trace log. Data encryption is not being used. Using trace logging is something to be done while troubleshooting, but it should not be done in production.

Encrypt Data Stored on Oracle Database

So far, we have encrypted data as it traversed the network between the client and the Oracle database. We have also encrypted data while in memory, but not in use on the client (the list of connection strings.) However, we have stored data on the Oracle database in clear text. In particular, we are concerned for the connection string lists that we have stored on Oracle in clear text. Well, we are going to address that concern here.

DBMS_CRYPTO Package

Oracle database provides a PL/SQL package that can assist us in doing data encryption at rest (while data is in storage in the database). By using the `DBMS_CRYPTO` package, we can selectively encrypt specific columns of data. Other alternatives exist, including full-table encryption. The Oracle Advanced Security product, which can be purchased separately from Oracle, can be used to accomplish that.

The `DBMS_CRYPTO` package is installed by default in the Oracle Database 11g, but it is not enabled by default. We will enable it for our application security, `appsec` user. We are going to accomplish encryption/decryption using a stored procedure that is owned by `appsec`, but is executed by `appver`. I'm telling you this to explain why we are granting execute on `DBMS_CRYPTO` to `appsec` instead of to the `app_sec_role`. The reason is that no roles exist unless you are in a session, and in the scenario I just described, no session exists for `appsec`. At an Oracle SQL command prompt as `SYS`, we will execute this command.

```
GRANT EXECUTE ON sys.dbms_crypto TO appsec;
```

Passwords and Keys

Now, as a matter of full disclosure, I hasten to add that `DBMS_CRYPTO` can be used for a variety of encryption/decryption tasks. Theoretically, we could do encryption on the client using Java and JCE, and do decryption on Oracle database using `DBMS_CRPTO`. There are two problems with that proposal. First, `DBMS_CRYPTO` is not nearly flexible enough for us to achieve the same level of security as we have with the `OracleJavaSecure` process. And second, `DBMS_CRYPTO` doesn't act like a single side of an encryption/decryption communication; rather, it assumes it is the party doing both encryption and decryption. It doesn't readily exchange encryption keys, nor expose its algorithms.

To meet the requirements of encrypting data at rest, we need to be able to decrypt data at a different time, in a different place by a different user. That said, it should be obvious that we cannot "negotiate" a key or set of keys at the time they are needed. Somewhere we need to store the encryption key, and retrieve it at some unspecified time in the future to decrypt the data.

The burden falls on the developer to decide where to store the encryption keys for use with `DBMS_CRYPTO`. There are benefits of having the password stored right in the Oracle database with the encrypted data. One benefit is that when we restore a backup of the encrypted data, we also restore the decryption key. The risk here is that, if someone gets access to the encrypted data, they might also get access to the decryption key.

One viable alternative would be to store the decryption key in the client application and to pass it to Oracle database when we want to decrypt the data; however, that hardly seems more secure. No, we will stick with storing the encryption key in the database, and we will take a couple steps to assure that it can't be used by anyone outside our application. Before I tell you how I do it, think about how you would do it.

Encryption at Rest Key Store

We will store our encryption/decryption keys, at least the genesis of the keys, in an ordinary Oracle table, `t_application_key`—see Listing 11-15. It is possible we may want to store these keys on a separate Oracle instance and acquire them across a DB link. In that way, the encrypted data and encryption keys will be backed up separately. We will provide a version number column, in case we want more than one key per database, possibly for a different application. You'll see why that's probably unnecessary. We also create an index and a view, `v_application_key`, of this table.

■ **Note** You will find this SQL script in a file named *Chapter11/AppSec.sql*.

Listing 11-15. *Table of Encryption at Rest Keys*

```
CREATE TABLE appsec.t_application_key
(
    key_version NUMBER(3) NOT NULL,
    -- Max Key size 1024 bits (128 Bytes)
    key_bytes   RAW(128) NOT NULL,
    create_ts   DATE DEFAULT SYSDATE
);
```

One assurance we want to make is that the key is never changed. We are going to do that with an on update/delete trigger. This trigger, shown in Listing 11-16, will basically turn the attempted update around and reject it.

Listing 11-16. *Before Update/Insert Trigger on Table of Encryption at Rest Keys*

```
CREATE OR REPLACE TRIGGER appsec.t_application_key_budr BEFORE UPDATE OR DELETE
    ON appsec.t_application_key FOR EACH ROW
BEGIN
    RAISE_APPLICATION_ERROR(-20001,'Cannot UPDATE or DELETE Records in V_APPLICATION_KEY.');
END;
```

```
/

ALTER TRIGGER appsec.t application key budr ENABLE;
```

Let's go ahead and insert several records into `v application key`, one of which we will use. See Listing 11-17. We will use the `RANDOMBYTES` function in the `DBMS CRYPTO` package to generate a string of 128 random bytes to use as our key genesis. I'm saying *key genesis* because, as you'll see, the real encryption/decryption key is assembled from these key bytes later on.

Listing 11-17. *Insert Several Random Encryption at Rest Keys*

```
INSERT INTO appsec.v application key
( key version, key bytes )
VALUES
( 1, SYS.DBMS CRYPTO.RANDOMBYTES(1024/8) );
```

Also insert values for `key version` numbers 2 through 5 using the same `INSERT` command.

Functions to Encrypt/Decrypt Data at Rest

We are going to build two Oracle stored functions to do server-side encryption of data to be stored in the database, and to decrypt it for use, as required. They are keyed to a specific version of the key genesis bytes in `t application key`, and they use `DBMS CRYPT` package to do encryption. We will make this difficult to duplicate by not taking our data and our key bytes and going straight to encryption; rather, we will perform several operations on the key bytes first. Anyone reading these functions will be able to tell what we are doing, and will be able to duplicate it, so we are going to hide the function code by passing these functions through the Oracle Wrap utility to obfuscate the code.

I encourage you to take these functions as a starting place, and modify them sufficiently to change the encryption process, then save a copy somewhere safe but hidden. You will convert them to wrapped functions, which will not be legible.

Our original `f mask` function, shown in Listings 11-18 through 11-21, takes the clear text connection strings list in the form of a `RAW`. It also takes the class name and version of the application inner class. It returns an encrypted `RAW` holding the list of connection strings.

Listing 11-18. *Signature of Function to Encrypt Data for Storage*

```
CREATE OR REPLACE FUNCTION appsec.f mask(
    clear raw       RAW,
    m class name    v app conn registry.class name%TYPE,
    m class version v app conn registry.class version%TYPE )
RETURN RAW
```

We hard-code a specific `key version` number of the genesis key bytes—version 4 in this instance (see Listing 11-19). This is a rather random decision on our part. We select the `key bytes` for that version from `v application key`. We get the `key bytes` into a variable named `app key`.

Listing 11-19. *Hard-Code Key Version*

```
AS
    crypt raw RAW(32767) := NULL;
    app_ver   v_application_key.key_version%TYPE := 4;
    app key   v application key.key bytes%TYPE;
    iv        RAW(16);
BEGIN
    SELECT key_bytes INTO app_key FROM v_application_key WHERE key_version = app_ver;
```

app key is like the cup with the marble under it in that three-cups concentration betting game. The code is quicker than the eye. We are going to process the bytes of the app key. The first process we perform is to take the class version and concatenate with the string "SufficientLength". Then, as shown in Listing 11-20, we XOR the app key with that concatenated string. Perhaps only the first 20 or so bytes of app key are modified by XOR.

■ **Note** We have just made this process unique to the specific version of the specific application (the one presenting the inner class).

Listing 11-20. *XOR the Key with the Class Version and Get MD5 Hash of Key*

```
    app key := SYS.UTL RAW.BIT XOR( app key,
        SYS.UTL RAW.CAST TO RAW(m class version||'SufficientLength') );
    app key := SYS.DBMS CRYPTO.HASH( app key, SYS.DBMS CRYPTO.HASH MD5 );
    app key := SYS.UTL RAW.CONCAT( app key, app key );
```

Our next process sets app key equal to the Message Digest (MD5) hash of app key. Listing 11-20 shows this. MD5 is a one-way hash algorithm that creates a 16-byte (128-bit) hash representing the initial value. Any modification to the initial value will cause the hash to change, and if the initial value doesn't change, MD5 will always calculate the same hash. Then, to get 32-byte key, we set app key equal to the concatenation of two of the MD5 hashes.

For the encryption algorithm we will be using, we will also need a 16-byte RAW initialization vector (IV). We are going again to make this function *specific to the application* by using the application inner class name as part of the IV. See Listing 11-21. Actually, we concatenate the class name with the string "SufficientLength," cast that to a RAW, and get the first 16 bytes as the IV.

Listing 11-21. *Get Initialization Vector with Class Name and Call DBMS_CRYPTO.ENCRYPT*

```
    iv := SYS.UTL RAW.SUBSTR(
        SYS.UTL RAW.CAST TO RAW(m class name||'SufficientLength'), 0, 16 );
    crypt raw := SYS.DBMS CRYPTO.ENCRYPT( clear raw,
        SYS.DBMS CRYPTO.ENCRYPT AES256 + SYS.DBMS CRYPTO.CHAIN CBC +
        SYS.DBMS CRYPTO.PAD PKCS5, app key, iv );
    RETURN crypt raw;
END f mask;
```

Then it's just a matter of calling the DBMS_CRYPTO.ENCRYPT function as in Listing 11-21. We pass our clear text connection string list clear_raw, the app_key and the iv. We also tell the function to use 256-bit Advanced Encryption Standard (AES256) with Block Chaining and PKCS padding.

Ta-da! We have an encrypted list of connection strings that we can trust for storage on disk and in backup. Our f_unmask function that decrypts the data is almost identical to f_mask. We build the app_key and iv in the very same way, then pass the encrypted connection string list to the DBMS_CRYPTO.DECRYPT function, using the same encryption algorithm series. Voila, we have our clear text connection strings back from cold storage.

Wrap Utility

Oracle Corporation has been in the business of protecting its intellectual property for decades. The company has developed a process whereby it can publish business-sensitive PL/SQL code, much like our Oracle procedures, functions, and packages, and distribute it to customers without exposing the inner workings of the code. Oracle devised the wrap utility, which will obfuscate PL/SQL code so that it can't be read. I should only say that wrapped procedures cannot be read without some effort, because there are purported to be tools that will unwrap procedures.

We will use the wrap utility to obfuscate the f_mask and f_unmask functions. As a reminder, you should first modify f_mask and f_unmask so that they are unique to your company, then wrap them. That is to avoid the obvious loose lips of this book from sinking your ship.

Save a copy of your *F_MASK.sql* and *F_UNMASK.sql* files in a secure location, then pass the files to the wrap utility. The wrapped files will have a ".plb" extension, and are viewable in any text editor— they are not binary code. The resulting Oracle 11g wrapped procedure will always resemble Listing 11-22.

```
%ORACLE_HOME%\BIN\wrap INAME=F_MASK.sql
%ORACLE_HOME%\BIN\wrap INAME=F_UNMASK.sql
```

Listing 11-22. *Wrapped Version of Mask Function*

```
CREATE OR REPLACE FUNCTION appsec.f_mask wrapped
a000000
b2
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
8
3d9 237
GehnTGWDxAhWnsVg2jYOTJ2/sF4wg/BeTCCsfI5VgpOGvFbmFJFF9PpfKGM8NUbmI21KsMmT
9YLZz1gSTsZkw/skypO3G2z+bhL/AGJObl6IY3bf/PjNwdlhZ5argmaJytVXORDALqjMIRvj
```

```
GLdGjZoM6cJZs4nHbLQMRgmOh9ZTnOnUOfQMGOvDHhtBLOCZSmx1ROSWpFQ2OIui96EL3CD4
...
1atpfb/f+oVZAZkY78TOYBdSmyOSgifZtmOIiEdc5rh/Lbn5pmTzHV8=

/
```

Notice the first line of the wrapped procedure is a `CREATE OR REPLACE` statement. We can copy and paste this code to any SQL editor, like SQL*Plus, and create the Oracle structure in the database.

I know there is a lot going on, so let me reiterate our goal. In wrapping these functions, our intent is to keep folks, whether hackers or just snoopers, from knowing how we encrypted the list of connection strings, and from being able to independently decrypt the strings and read them.

Changes to setDecryptConns()/getCryptConns()

In the middle of the `setDecryptConns()` method that runs as a Java stored procedure, we take the list of connection strings that we are about to store in Oracle database, and we pass them to the `f mask` function. Listing 11-23 shows this. The encrypted bytes that are returned from `f mask` are then stored in the database.

Listing 11-23. *Call to Encrypt Connection Strings for Storage*

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

■ **Note** This code can be found in *Chapter11/orajavsec/OracleJavaSecure.java.*

We also modify the `getCryptConns()` method to decrypt the list of connection strings before we return them to the client application. This is shown in Listing 11-24.

Listing 11-24. *Call to Decrypt Connection Strings from Storage*

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

oins = new ObjectInputStream( new ByteArrayInputStream(
```

```
        stmt.getBytes(1) ) );
    Object currentConns = oins.readObject();
```

Manage Connection Strings for Applications

We have already seen, and have been using the functionality to add and update connection strings in our list, and to save them in Oracle database. Recall that those are two separate steps. The first step, adding or updating connection strings in our list, happens on the client and only affects the list currently being used by the client application. It is only when we save the list to Oracle database that we make the new or replacement connection string available to all future application users.

In the same way, we will need to remove connection strings from the list. Again, we will make the removal permanent by saving the list to Oracle database in a separate step.

Finally, we will want to be able to replicate the list of connection strings for use by a new version of our application. This replication process must be application-specific, so a poser application can't grab our list of connection strings for her illicit use.

Be forewarned that anyone who gains access to the application security, `appsec` user has access to copy connection strings from one application to another. Fortunately, we have taken care of that issue by making the `f mask` and `f unmask` functions both application and version specific. Another application may lay hold of our list of connection strings, but they won't be able to decrypt them. That means, though, that when we replicate our list of connection strings to a new version of the application, we will have to `f unmask` them with the old version and `f mask` them with the new version.

Create an Application Administrative User

Up to this point, we have described an application user named `OSUSER` whom I suggested was probably you (your OS user ID). This is an ordinary application user who was granted access to see sensitive data in the `HR` schema by way of having been granted `CONNECT THROUGH appusr`.

Now it has come time for us to differentiate between regular application users and administrative application users. We need a second user, whom I am going to call `OSADMIN`. The difference between these users will come in the next section through a single role grant. The reason for that additional role will be to manage the update of connection string lists for applications on the Oracle database.

For now, let's create a second application user. First of all, you will need to have an additional OS user account named `osadmin` in our example. As a Windows administrative user, you can create that account through the Control Panel/User Accounts utility. Once that exists, run the commands in Listing 11-25 to create the `OSADMIN` user (substitute the OS user ID of the OS user you just created) and grant him the same access to sensitive data in the `HR` schema that we have given to `OSUSER`. We do this so we can test just the difference that is made by the new administrative role.

Listing 11-25. *Create the OSADMIN Oracle User*

```
CREATE USER osadmin IDENTIFIED EXTERNALLY;
GRANT create session role TO osadmin;
ALTER USER osadmin GRANT CONNECT THROUGH appusr;


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
    (EMPLOYEES_SEQ.NEXTVAL, 'First', 'Last', 'OSADMIN.MAIL',
```

```
    '800.555.1212', SYSDATE, 'SA REP', 5000, 0.20, 147, 80);

COMMIT;

SELECT EMPLOYEE_ID FROM EMPLOYEES WHERE EMAIL='OSADMIN.MAIL';


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 EMPLOYEES WHERE EMAIL='OSADMIN.MAIL'),
      'OSADMIN', '12345', '8005551212', 'Verizon' );

COMMIT;

SELECT * FROM hr.v_emp_mobile_nos WHERE user_id = 'OSADMIN';
```

The significant aspects of this code are as follows:

1. We create an Oracle user by the same name as our new OS user ID.
2. We grant the new user permission to `CONNECT THROUGH appusr` so that he can select encrypted sensitive data from the `HR` schema.
3. We insert a record for this user in the `HR.EMPLOYEES` table and get the next sequential value from `EMPLOYEES SEQ` as the new `EMPLOYEE ID.`
4. We can select the new user based on `EMAIL` address, which is a unique field.
5. Using that selection (for simplicity here), we can insert a matching record into `hr.v emp mobile nos` with the `user id` field set to the new OS user ID.
6. We finish by selecting the record we just created from `hr.v emp mobile nos.`

We require the records for this user in the `EMPLOYEES` table and the `hr.v emp mobile nos` view so that we can accomplish two-factor authentication for this user. Our administrative users will have to accomplish both SSO and two-factor authentication, just like every other user.

Create an Administrative Role for Application Verification

So far, we have allowed any successful application user to both insert and update the list of connection strings and store them in Oracle database for the application. Let's tighten up that process by creating an Oracle role that is required in order to administer the lists of connection strings for applications. We will use the application verification administrator, `appver admin` role for this duty. As `SYS`, execute these commands.

```
CREATE ROLE appver admin NOT IDENTIFIED;
```

Still, any user can manage her own copy of the connection strings in the local instance of the client application (we give users that freedom), but inserting, updating, or replicating the connection strings in Oracle database will require preapproval. We will preapprove certain persons, via the user IDs they have in the operating system that is SSO'd to Oracle database. To those user IDs, we will grant our new administrative role. It is a default role for that person.

```
GRANT appver admin TO osadmin;
```