sent. We will add these to your organization's Windows domain name that we set for SSO as items that you need to edit in *OracleJavaSecure.java* before compiling. See Listing 9-17.

■ **Note** Edit the code found in the file named *Chapter9/orajavsec/OracleJavaSecure.java.*

Listing 9-17. *Set Company Specific Mail and Paging Addresses*

```
private static String expectedDomain = "ORGDOMAIN";
private static String comDomain = "org.com";
private static String smtpHost = "smtp." + comDomain;
private static String baseURL =
    "http://www.org.com/servlet/textpage.PageServlet?ACTION=2&PAGERID=";

private static String msgURL = "&MESSAGE=";
```

Compile Two-Factor Delivery Route Codes: Binary Math

We assemble a single code to represent the compilation of delivery routes selected for transmitting the two-factor authentication code. For each route selected, we add a constant value (static final) to our cumulative distribution code. At the end of processing, the single code represents all routes that were used. Like bits in a byte (eight bits, each with a value double the previous, represent 256 unique values), we use binary math to accumulate our distribution code. The initial constants are listed in Listing 9-18.

Listing 9-18. *Delivery Route Constants*

```
private static final int USE PAGER = 1;
private static final int USE SMS = 2;

private static final int USE EMAIL = 4;
```

Table 9-1 lists how the sums represent all the different combination of delivery routes. Notice that the maximum value is one less than double the largest value (7 is the max value, which is one less than double 4, the largest constant).

If you haven't already guessed, the next constant would have the value of 8, then 16 and 32. We have only sized the `distrib cd` column in the `t two fact cd cache` table to hold 2 digits, so we are limited to those six constant values. The maximum sum of those six constants is 63. The next, seventh constant would be 64, but a max sum including that value would be 127, a three-digit value.

Exploring a Method to Distribute the Two-Factor Codes

When we defined the `f send 2 factor` Oracle function earlier, we noted that it simply calls the `distribute2Factor()` Java method: it is a pass-through Oracle function, as most Java stored procedures are. The reasoning is a bit esoteric. You see, we are already executing on Oracle database, and in the middle of running the `p check hrview access` procedure when we call `f send 2 factor`. Why not just call the Java `distribute2Factor()` method from `p check hrview access`? That would be a nice capability, but it is not available. We need to approach the Oracle JVM through a dedicated Java stored procedure that is declared `AS LANGUAGE JAVA`. We cannot mix PL/SQL with a Java call in the same function or

procedure; hence, we call a separate PL/SQL function as a Java stored procedure to get access to the Java method.

When we get to the `distribute2Factor()` method, we already know who the user is from our SSO processing. We pass that user ID to the method so that we can send our two-factor code to the devices owned by the intended recipient. Set up and closure of the method are familiar, as you can see in Listing 9-19. This method returns a String which represents the distribution code (summary of routes the two-factor code was sent).

Listing 9-19. *Method to Distribute Two-Factor Codes: Framework, distribute2Factor()*

```
public static final String distribute2Factor( String osUser ) throws Exception {
    // Do not resend this two-factor authentication code,
    //  nor a new one using this session
    if ( twoFactorAuthChars != null ) return "0";

    int distribCode = 0;

    Statement stmt = null;
    try {
        •••

    } catch( Exception x ) {
        java.io.CharArrayWriter errorText = new java.io.CharArrayWriter( 4000 );
        x.printStackTrace( new java.io.PrintWriter( errorText ) );
        stmt.executeUpdate( "CALL app sec pkg.p log error( 0, '" +

            errorText.toString() + "', '')" );
    } finally {
        try {
            if( stmt != null ) stmt.close();
        } catch( Exception y ) {}
    }
    return String.valueOf( distribCode );

}
```

Notice the ellipses (…) in the middle of Listing 9-19. That is where the code resides that we discuss in the subsections that follow.

Creating the Two-Factor Code

Within the `distribute2Factor()` method, we generate two-factor authentication codes that adhere to our prescribed format: 12 numeric characters in three groups of four, separated by dashes (e.g., 1234-5678-9012). I propose that this format is easily read and entered by users. Additionally, and this is significant, very old, numeric-only pagers are only capable of displaying a limited number of characters; often only numeric characters and dashes. The prescribed format conforms to that lowest common denominator.

We build our two-factor code as a character array of 14 characters, as shown in Listing 9-20. We put a random numeric character in each place. Within the ASCII character set, numeric values run from 48 to 58. That span is size 10, and we basically select the next random integer between 0 and 9. We add the first ASCII value for numeric character “0”, (48), so in essence we get a random value between 48 and 58 that we cast as a `char` and set in the `twoFactorAuthChars` array.

As we march through the `for` loop, we observe our position. After every four characters, we want to place a dash. We do a modulus 5 (% 5) to test our position. When the next place, modulus 5 equals 0, we need to set the next place to a dash character. The first array position is 0, so we add one, and we want to test our next place, so we add another one. We test `(i+2)%5`.

When our test is positive, we go ahead and increment it so we can set the next place to a dash and continue; however, we don't want to try to tack another dash character on the end of our array, so we only set a dash `if ( i < twoFactorLength )`.

Listing 9-20. *Build the Two-Factor Code*

```
private static int twoFactorLength = 14;
private static char[] twoFactorAuthChars = null;


twoFactorAuthChars = new char[twoFactorLength];

for ( int i = 0; i < twoFactorLength; i++ ) {
    // Use numeric only to accommodate old pagers
    twoFactorAuthChars[i] = ( char )( random.nextInt( 58 - 48 ) + 48 );

    // Insert dashes (after every 4 characters) for readability
    if( 0 == ( ( i + 2 ) % 5 ) ) {
        i++;
        if ( i < twoFactorLength )
            twoFactorAuthChars[i] = '-';

    }
}
String twoFactorAuth = new String( twoFactorAuthChars );
```

We are only using the `String` representation of our two-factor authentication code in this method, so we set a method member. Everywhere else, the two-factor authentication code is referred to as a `char` array. We test the existence of `twoFactorAuthChars` array when we enter this method (see Listing 9-19), and we retain it as a static class member.

Dealing with Oracle and Java Dates: Standard Practice

Are you familiar with the Jim Croce song, "Time in a Bottle"? One of my favorite lines from that song is, "I've been around enough to know that you're the one I want to go through time with." Some old coding practices are like that. One particular standard practice I maintain is the exchange of dates between Oracle database and Java. I have never gotten into trouble when abiding by this plan, which I'm about to describe.

The only potential problem with what I am going to present to you is that the precision is limited to seconds. If you need milliseconds precision, you will need to modify the approach somewhat. Well, there is also the issue of time zone, which we have discussed; you may have to deal with that if your company is spread across multiple time zones.

The first standard I propose is to always use `java.util.Date`, and never use `java.sql.Date`. Besides the confusion due to the fact that they have the same name (but are mutually exclusive), there is the problem that they operate differently. Learn `java.util.Date`. Always import it and use it exclusively. You can use both `Date` classes, but great care is required to always address them as either `java.util.Date` or

java.sql.Date, and to refer to them correctly. But more reasonably, you will have to choose one or the other. Choose java.util.Date.

One benefit we might garner from using java.sql.Date (not our preferred approach) would be that we could call ResultSet.getDate() and get a java.sql.Date object directly, without casting. However, the most important single function that is missing from java.sql.Date is the ability to get the current time and date. This topic came up in Chapter 7, in the sidebar "A Tale of Two Date Classes." On the other hand, with java.util.Date, we simply get a new Date(), and the default constructor builds a Date based on the current milliseconds value. A second reason to avoid java.sql.Date is that if you decided to standardize on that Date in an application, you would have to import it from the java.sql package even in some classes that have no interaction with SQL databases. In those cases, I would argue, java.sql.Date is out of place.

The format Strings and a SimpleDateFormat class shown in Listing 9-21 are what we need to do date exchange between Oracle database and Java. This is used in our distribute2Factor() method.

Listing 9-21. *Standard Date Exchange Between Oracle Database and Java*

```
import java.text.SimpleDateFormat;
import java.util.Date;


    String oraFmtSt = "YYYY-MM-DD HH24:MI:SS"; // use with to char()
    String javaFmtSt = "yyyy-MM-d H:m:s";
    SimpleDateFormat ora2JavaDtFmt = new SimpleDateFormat( javaFmtSt );
```

The next standard practice I propose is that you always exchange dates as Strings / VARCHAR2s. We will define analogous formats for dates that allow us to exchange them as strings and rebuild representative dates on both Java and Oracle Database. On Oracle database, we use the TO CHAR and TO DATE functions; and in Java, we use SimpleDateFormat.parse() and SimpleDateFormat.format() methods. The standard methods for exchanging dates between Oracle database and Java are presented in Table 9-2.

Table 9-2. *Methods for Exchanging Dates Between Oracle Database and Java*

Perspective for Date Exchange	Method Used for Date Representation
Sending date from Oracle to Java	`SELECT TO CHAR( SYSDATE, 'YYYY-MM-DD HH24:MI:SS' ) FROM DUAL;`
Receiving date string in Oracle from Java	`MY DATE DATE := TO DATE( CHAR DATE, 'YYYY-MM-DD HH24:MI:SS' );`
Receiving date string in Java from Oracle	`Date javaDate = ora2JavaDtFmt.parse( rs.getString(1) );`
Sending date from Java to Oracle	`stmt.setString( ora2JavaDtFmt.format( javaDate ) );`

What Routes Are Available for the User?

The next step in the distribute2Factor() method is to query the HR data to see what devices the user has: pagers, cell-phones, etc., to which we might send the two-factor authentication code. While we're at it, in one fell swoop, we will get the cached two-factor code, if any, the timestamp when it was cached and the IP Address from which it was requested.

Notice in our query (Listing 9-22) that we are using the Oracle TO_CHAR method to get the timestamp and using the format string which we pass from Java.

Listing 9-22. *Query for Distribution Routes Available*

```
    stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(
"SELECT m.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, '" + oraFmtSt + "' ), c.ip address " +
"FROM hr.v emp mobile nos m, hr.v employees public e, hr.v sms carrier host s, " +
"v two fact cd cache c WHERE m.user_id = '" + osUser + "' " +
"AND e.employee id =  m.employee id " +
"AND s.sms carrier cd (+)= m.sms carrier cd " +
"AND c.employee id (+)= m.employee id " );
    if ( rs.next() ) {
        String empID     = rs.getString( 1 );
        String pagerNo   = rs.getString( 2 );
        String smsNo     = rs.getString( 3 );
        String smsURL    = rs.getString( 4 );
        String eMail     = rs.getString( 5 );
        String ipAddress = rs.getString( 6 );
```

There is a bit of fault tolerance built into this query. Do you see the (+)= symbols? Those indicate what is called an *outer join*. Notice we are getting the sms carrier url from the v sms carrier host view where the user's sms carrier cd matches what's in v sms carrier host. Now, what if the user doesn't have a cell phone or sms carrier cd? If we left this query as a straight join (s.sms carrier cd = m.sms carrier cd), we would not get a record for that user. However, by adding the outer join indicator, (+) we are asking the query to return a result with the primary data even if this secondary data does not exist. We do another outer join on the two-factor cache data, because the user may not already have a two-factor authentication code in the cache.

I want to reiterate that if the user does not have an entry in the t emp mobile nos table, he will not be able to do two-factor authentication. He does not necessarily need to have a cell phone or pager, but the association of SSO user ID (user id) and employee id requires that there be an entry in t emp mobile nos for the user. If the user's record in t emp mobile nos has no SMS phone number or pager number, then the distribute2Factor() method sends the message to the user's e-mail.

Is the Current Pass Code Still Valid?

In `distribute2Factor()` we want to determine if the user is coming back to request another two-factor authentication code before the 10-minute timeout on an existing two-factor code has expired. The user may or may not have an existing, cached two-factor authentication code: we may get `nulls` back for values from that table. For that reason, we bracket the following code (Listing 9-23) in a try/catch block. Getting `nulls` into our `String` members is no problem, but parsing a `null` into a `Date` and comparing `nulls` with the `String.equals()` method will throw exceptions.

Listing 9-23. *Test Validity of Cached Two-Factor Authentication Code*

```
try{
    String cTimeStamp = rs.getString( 7 );
    String cIPAddr    = rs.getString( 8 );
    // Ten minutes ago Date
    Date tmaDate = new Date( (new Date()).getTime() - 10*60*1000 );

    Date cacheDate = ora2JavaDtFmt.parse( cTimeStamp );
    // If user coming from same IP Address within 10 minutes
    // do not distribute Code (will overwrite code from new IP Addr)
    if( ipAddress.equals( cIPAddr ) && cacheDate.after( tmaDate ) )
        return "0";

} catch( Exception z ) {}
```

The heart of this code is the last line where we test if the user is coming from the same IP Address for which we generated the two-factor code, and if the date is less than 10 minutes old. If that is the case, we do not generate a new two-factor code, and we do not resend the existing code; we simply return.

While here, take a minute to read through the line of code we use to calculate ten minutes ago, `tmaDate`. We get the milliseconds of the current date and subtract 10 minutes of 60 seconds of 1,000 milliseconds.

Distribute the Pass Code to Routes

Next on our agenda in the `distribute2Factor()` method is to send the generated two-factor code to the preferred and/or existing devices. By preference, we will send the two-factor code to the user's pager and cell phone. If neither of those is available, we will send the code to the user's e-mail. See Listing 9-24.

To send the code to a cell phone, the user must have both a phone number and a carrier code. If he does, we call the `distribToSMS()` method. Notice that we add the return value, `int` to our cumulative `distribCode`. Similarly, we send the two-factor code to a pager if the user has pager number. In the following sections, we explore the individual methods to send 2-factor codes to specific devices.

Listing 9-24. *Call Methods to Distribute to SMS, Pager, and/or E-Mail*

```
if( ( smsNo != null ) && ( !smsNo.equals( "" ) ) &&
    ( smsURL != null ) && ( !smsURL.equals( "" ) )
)
    distribCode += distribToSMS( twoFactorAuth, smsNo, smsURL );

if( ( pagerNo != null ) && ( !pagerNo.equals( "" ) ) )
    distribCode += distribToPagerURL( twoFactorAuth, pagerNo );
```

```
    // Recommend not send to e-mail unless no other distrib option succeeds
    // !Uncomment code in next line!
    if( //( distribCode == 0 ) &&
        ( eMail != null ) && ( !eMail.equals( "" ) )
    )
        distribCode += distribToEMail( twoFactorAuth, eMail );
```

Currently, we have commented a test that we will want to implement that avoids sending the two-factor code to e-mail if we were successful in sending the code to a pager or cell phone. We've commented the test so that you can also see the e-mail route in action, but you may want to uncomment the test in production. Also, we can only send the code to e-mail if the user has an e-mail address.

Cache the Pass Code in Oracle

Once we have distributed our two-factor authentication code, if we actually found a route to deliver it (`distribCode > 0 `), we want to cache it for comparison to any code the user enters. The following statement from the `distribute2Factor()` method (Listing 9-25) updates the entry, if it exists. One feature of an update statement is that it returns an integer that indicates how many rows were updated. If we see that less than 1 row was updated, we assume we need to insert a row to cache codes for this particular user. In other words, we attempt an update which also serves as a test to see if we need to insert a row. Once we've got most users entered for most applications, we will almost always want to do an update, so this order, update then insert, is the most efficient.

Listing 9-25. *Cache the Two-Factor Authentication Code*

```
    if( distribCode > 0 || isTesting ) {
        int cnt = stmt.executeUpdate(
"UPDATE v two fact cd cache SET two factor cd = '" + twoFactorAuth +
"', ip address = '" + ipAddress + "', distrib cd = " +
String.valueOf( distribCode ) + ", cache ts=SYSDATE " +
"WHERE employee id = " + empID );
        if( cnt < 1 )
            stmt.executeUpdate(
"INSERT INTO v two fact cd cache( employee id ,two factor cd, distrib cd ) VALUES " +
"( " + empID + ", '" + twoFactorAuth +"', " + String.valueOf( distribCode ) + " )" );
    }
```

Distributing the Code to SMS

We saw earlier in this chapter, in our testing, that we need to set a session property for our SMTP server. That is the first statement we execute in Listing 9-26. After that, we call the `UTL_MAIL` package to send a message to the user's cell phone. The arguments for the send function are sender's e-mail, recipient e-mail, two other distributions we are not concerned with, the message title ("Response"), and the message text (our two-factor authentication code).

We build the recipient e-mail address from the user's cell phone number, `smsNo`, and the SMS gateway address for his carrier, `smsURL`.

Listing 9-26. *Distribute the Code to SMS, distribToSMS()*

```
private static final int distribToSMS( String twoFactorAuth, String smsNo,
    String smsURL )
{
    int distribCode = 0;

    Statement stmt = null;
    try {
        stmt = conn.createStatement();
        stmt.executeUpdate( "ALTER SESSION SET SMTP_OUT_SERVER = '" +
            smtpHost + "'" );
        stmt.executeUpdate( "CALL UTL_MAIL.SEND( 'response@" +

            comDomain + "', '" + smsNo + "@" + smsURL +
            "', '', '', 'Response','" + twoFactorAuth + "' )" );
        distribCode += USE_SMS;

    } catch ( Exception x ) {
    } finally {
        try {
            if( stmt != null ) stmt.close();
        } catch( Exception y ) {}
    }
    return distribCode;

}
```

As an example, this message would look like the following.

```
From: response@org.com
To: 8005551212@txt.att.net
Subject: Response
1234-5678-9012
```

Notice also in this message that we are using the `try/catch/finally` block to close the statement. We set a method member, `distribCode`, to 0 at the outset. If we succeed in sending the code to the cell phone, we add the value of the `USE SMS` constant to `distribCode`. And finally we return `distribCode`.

Distributing the Code to Pager URL

The `distribToPagerURL()` method has many similarities to `distribToSMS()`, which we just discussed. The main difference is that we are not using an Oracle package to send e-mail but are using a pure Java `URL` class to read a web page. See Listing 9-27. In actuality, we are not concerned with what a browser would see when going to this web page. The entry form that a user would see in his browser window is never loaded. It is bypassed because we are submitting the input fields data as part of the URL address. We are using what is called the `GET` method to pass values to the web server on the URL line. Our URL will resemble this:

```
www.org.com/servlet/textpage.PageServlet?PAGERID=12345&MESSAGE=1234-5678-9012
```

We indicate what pager to send in the PAGERID argument and what the message will be in the MESSAGE argument. After we create a new URL instance with this address, we call its getContent() method to "browse" the web address. At that point, the web server with the paging application has responded to the GET method, and if no exception is thrown, we can continue on with returning the value of the USE PAGER constant.

Listing 9-27. *Distribute the Code to Pager, distribToPagerURL()*

```
private static final int distribToPagerURL( String twoFactorAuth,
    String pagerNo )
{
    int distribCode = 0;

    try {
        URL u = new URL( baseURL + pagerNo + msgURL + twoFactorAuth );
        u.getContent();
        distribCode += USE_PAGER;

    } catch ( Exception x ) {}
    return distribCode;

}
```

This method is the most likely to require special editing to be functional in your organization. You will have to research how page messages are sent to your corporate pagers, if at all. Even if you do not have pagers, this code may serve as an example of how you might send the two-factor codes to some other web service.

Distributing the Code to E-Mail

Our distribToEMail() method is not only similar, but also almost identical to the distribToSMS() method, except that the recipient in this case is the user's e-mail address. I'm leaving out the code, but the message resembles this:

```
From: response@org.com
To: OSUSER@org.com
Subject: Response
1234-5678-9012
```

If successful, we return the value of the USE EMAIL constant.

Testing Two-Factor Authentication

In this section, we again draw on all our resources to demonstrate and test two-factor authentication along with everything else we have discussed so far. We will test our two-factor authentication by using the code we have been examining to generate and transmit two-factor authentication codes. In order to experience this fully, you will need to enter your pager, cell phone, and/or e-mail addresses in the database, as we described. Alternatively, you can query the database to select the generated two-factor codes from the appsec.v two fact cd cache view.

If you haven't been executing the SQL commands we've been exploring throughout this chapter as we discussed them, you will need to open the *Chapter9* folder and the files *Sys.sql*, *SecAdm.sql*, *AppSec.sql*, and *HR.sql* and execute those commands as the appropriate user. You can execute them in

about that order, but when you get about halfway through *AppSec.sql*, you will need to stop and execute *HR.sql* before continuing as `appsec`.

You will also need to execute *OracleJavaSecure.java* in the Oracle database to create the Java structure there. We will then run the class `TestOracleJavaSecure` on the client to accomplish all our demonstrations and tests.

Updating OracleJavaSecure Java in Oracle

If you haven't already, edit the code for *OracleJavaSecure.java* to provide our company-specific addresses for two-factor authentication. We saw this previously in Listing 9-17. Edit the code found in the file named *Chapter9/orajavsec/OracleJavaSecure.java*.

```
private static String expectedDomain = "ORGDOMAIN";
private static String comDomain = "org.com";
private static String smtpHost = "smtp." + comDomain;
private static String baseURL =
    "http://www.org.com/servlet/textpage.PageServlet?ACTION=2&PAGERID=";

private static String msgURL = "&MESSAGE=";
```

Load the new *orajavsec/OracleJavaSecure.java* code into Oracle database. You will be loading this into the `appsec` schema, so you should be connected to Oracle Database as that user and don't forget to set your role to `appsec role`. Again, uncomment the top line that begins `CREATE OR REPLACE AND RESOLVE JAVA…` and execute it in your SQL client (SQL*Plus, SQL Developer, JDeveloper, or TOAD). Remember to set the role first. (This file will also be compiled and executed on the client, so re-comment the first line before saving the file.)

You may also need to modify the environment of your SQL client in order to include an ampersand in your URL string, as in the example `baseURL`, shown previously. Notice the ampersand between the parameters for "ACTION=2" and "PAGERID=". When some SQL clients see an ampersand like that, they will assume it is a marker for a variable to be substituted on execution, in this case "&PAGERID". Most times that would be a good assumption, but not in this case and never with Java code that we are loading into Oracle database. We can remedy this situation in one of two ways: we can use a different SQL client, or even resort to using the loadjava utility, as described in Chapter 4. However, a simpler and more immediate fix is to tell our SQL client not to do variable substitution with this simple command.

```
SET DEFINE OFF;
```

Editing the Test Code

We are going to execute a separate Java class, `TestOracleJavaSecure`, to test our two-factor authentication. Edit the code near the top to set the appropriate password and connection string for the `appusr` user.

■ **Note** You can find the code for the `TestOracleJavaSecure` class in the file *Chapter9/TestOracleJavaSecure.java*.

Planning to Pass the Two-Factor Code as an Argument to Main

The entire code of the `TestOracleJavaSecure` class resides within the `main()` method. So, we simply run the code from top to bottom when we call Java with this class from the command line. This is similar to how we did testing in Chapter 7. Here, however, we will have to call the class two times. The first time, we call it without any arguments. At the end of that call, if all goes as planned, a two-factor authentication code will have been sent to our cell phone, pager, and e-mail.

Once we have received that code on any of our devices, and within 10 minutes, we can execute Java again calling `TestOracleJavaSecure`, except this time we will include the two-factor code as an argument. The two commands that we execute later will resemble these:

```
java TestOracleJavaSecure
java TestOracleJavaSecure 1234-5678-9012
```

If you don't have any devices or e-mail on which to receive two-factor authorization codes, you can set the `isTesting boolean` to `true` in *OracleJavaSecure.java* and reload it in Oracle database. Then recompile and run the commands given above. This will place a two-factor code in the `t two fact cd cache` table, even if no distribution devices are found. You may then get the generated two-factor code from this query:

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

Arguments on the command line are delivered to the `main()` method as an array of `Strings`. We can test for the presence of a two-factor code by testing if the `String` array length is greater than 0. We also assure ourselves that the first element of the array is not `null`, as shown in Listing 9-28.

Listing 9-28. *Pass Two-Factor Code to TestOracleJavaSecure main() Method*

```
public static void main( String[] args ) {
    try {
        // Passing 2-factor code in as argument on command line
        String args0 = "";
        if( args.length != 0 && args[0] != null ) args0 = args[0];
        args0 = OracleJavaSecure.checkFormat2Factor( args0 );
```

When the person enters the code he received on his cell phone, we want to make sure we put the least amount of burden on him to understand our formatting rules. If he only types in the numeric characters, leaving off the dashes, we want to accept that. If he appends other characters, like time and date or whatever his phone displays, we should pick out our two-factor code, if readily available. We send whatever he provides at the command prompt to the `OracleJavaSecure.checkFormat2Factor()` method. If you find other typographical errors are frequent, you might add some more intelligence to that method.

Planning to Acquire the Secure Application Role

Whether or not we have a two-factor code, we call our Oracle procedure, `p check hrview access`, for the secure application role, `hrview role`. If we do not have a two-factor code yet, we pass an empty string to the procedure; otherwise, we pass the code. You are now familiar with reading procedures like this, in Listing 9-29.

Listing 9-29. *Set Secure Application Role from TestOracleJavaSecure*

```
stmt = ( OracleCallableStatement )conn.prepareCall(
    "CALL appsec.p check hrview access(?,?,?)" );

stmt.registerOutParameter( 2, OracleTypes.NUMBER );
stmt.registerOutParameter( 3, OracleTypes.VARCHAR );
stmt.setString( 1, args0 );
stmt.setInt(    2, 0 );
stmt.setNull(   3, OracleTypes.VARCHAR );
stmt.executeUpdate();
errNo = stmt.getInt( 2 );
errMsg = stmt.getString( 3 );
if( errNo != 0 ) {
    System.out.println( "Oracle error 1) " + errNo + ", " + errMsg );
} else if( args0.equals( "" ) ) {
    System.out.println( "DistribCd = " + errMsg );
    System.out.println( "Call again with 2-Factor code parameter" );
} else {
    if( null != stmt ) stmt.close();
    System.out.println( "Oracle success 1)" );

    OracleResultSet rs = null;
    RAW sessionSecretDESPassPhrase = null;
    RAW sessionSecretDESAlgorithm = null;
    RAW sessionSecretDESSalt = null;
    RAW sessionSecretDESIterationCount = null;

    String locModulus = OracleJavaSecure.getLocRSAPubMod();

    String locExponent = OracleJavaSecure.getLocRSAPubExp();


    stmt = ( OracleCallableStatement )conn.prepareCall(
        "CALL hr.hr sec pkg.p_select_employees_sensitive(?,?,?,?,?,?,?,?,?)" );
```

We report any errors coming back from the procedure. One definite potential error is the possibility that the user has entered a wrong or old code, in which case a "DATA NOT FOUND" error will be returned.

If there are no errors and we didn't have a two-factor code when we called, then we presume the two-factor code was distributed, and we display the distribution code that was returned in the `errMsg` and ask the user to come again with their two-factor code. If they had a two-factor code when they called, and no errors, then we assume the `p check hrview access` succeeded, and the user's connection has been granted the `hrview role` and can proceed to read the data.

Our `TestOracleJavaSecure` class will execute the `p select employees sensitive` procedure to demonstrate that access has succeeded. You are, no doubt, all too familiar with that procedure.

Running the Tests and Observing the Results

To recap the requirements: To run the test, you must have created the tables, procedures, and grants that we have described in this chapter. You will have inserted a record for your user ID in the

`HR.emp mobile nos` table. And you will also have edited the company-specific values at the top of *Chapter9/orajavsec/OracleJavaSecure.java*. You should also uncomment the top line of *OracleJavaSecure.java* and execute it in Oracle database to load the Java structures.

After loading `OracleJavaSecure` into Oracle database, restore the comment on the top line and save the file. From a command prompt in the *Chapter9* directory, compile both *OracleJavaSecure.java* and *TestOracleJavaSecure.java* with these commands.

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

Then run the test by executing the first line, `java TestOracleJavaSecure`, and see the results. Execute the same command again to see that the two-factor code is not sent again (DistribCd = 0). After you receive the two-factor authentication code, execute the last command, passing the two-factor code on the command line.

```
java TestOracleJavaSecure
    DistribCd = 5
    Call again with 2-Factor code parameter
java TestOracleJavaSecure

    DistribCd = 0
java TestOracleJavaSecure 1234-5678-9012

    DistribCd = null
    Oracle success 1)
    Oracle success 2)
    198, Donald, OConnell, DOCONNEL, 650.507.9833, 2007-06-21 00:00:00, SH CLERK, 26
    00 (AD6E5035FAB394A8), null, 124, 50
```

After 10 minutes have elapsed, the two-factor code will no longer succeed in getting access. Try executing `TestOracleJavaSecure` again after 10 minutes with the same code and observe the "no data found" error message.

```
java TestOracleJavaSecure 1234-5678-9012
    DistribCd = ORA-01403: no data found
    Oracle error 1) 100, ORA-01403: no data found
```

Chapter Review

We enabled Oracle Database to send E-Mail messages and to browse to web pages. Using those abilities, we developed a process to send 2-factor authentication codes to our application users on their cell phones, pagers and e-mail accounts. Users will have to enter those 2-factor codes in order to get access to our application data.

We built a table to hold mobile phone numbers for users and another to hold web addresses of SMS gateways for each cell phone carrier. We also added a table to hold cached 2-factor codes.

We implemented foreign keys to maintain referential integrity between our tables. And we discussed a coding standard for exchanging dates between Oracle Database and Java.

An overview of the 2-factor code distribution process is provided in Figure 9-1. Notice a reference in this Figure to Figure 8-1, block [**D**]. It was there that we saw an illustration of the process of getting our SSO proxy connection. Using that connection, we call the `p check hrview access` procedure two times. The first time, we don't have a two-factor authentication code. Therefore, we do the process depicted in

block [E1] in Figure 9-1. In that block, we see an illustration of the process used to distribute the two-factor code to the individual devices associated with the user.

The second time we call `p check hrview access`, we provide the two-factor authentication code as an argument. Because we have returned with that code, we perform the process depicted in block [E2] of Figure 9-1. It is there that we see the tests for validity of the two-factor code, and if all is well, the `SET ROLE` command to acquire the `hrview role`.

With that role, our user can continue on to select and update sensitive data in the HR schema using encryption. Those processes are included in Figure 9-1 by a reference to Figures 7-1 and 7-2.

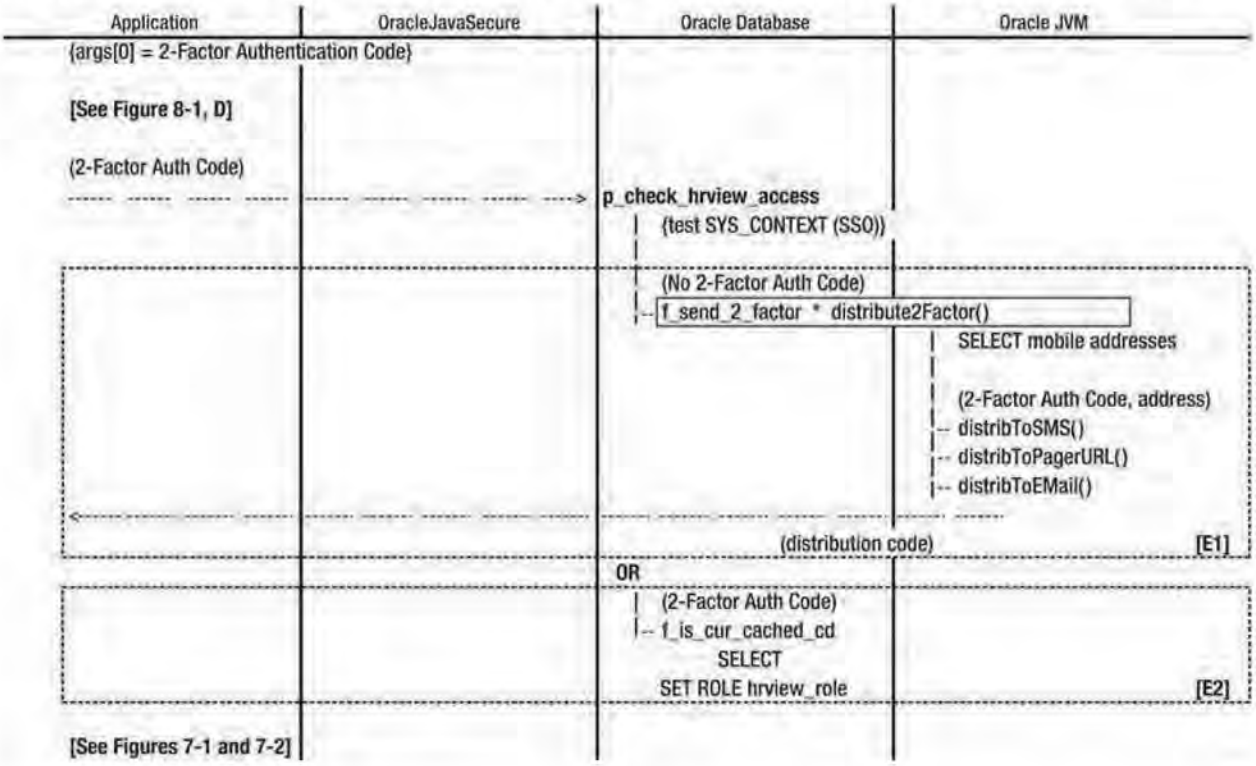


Figure 9-1. *The two-factor authentication code process*

CHAPTER 10

Application Authorization

Every Oracle application connects to one or more Oracle database instance, perhaps as one or more user. We have already seen how we can accomplish this with security and encryption for one application at a time. Our first application attached as the `appusr` account and had access to data in the `HR` schema via the `hr view` role.

For that application, and any similar one, the developer will develop a procedure for a secure application role, like our `p check hrview access`. But for all our security, there is one aspect that we haven't addressed: developers are still required to hard-code application user passwords into their code (or find another mechanism.) What if we provide a secure password store for them, one that is not as susceptible to imposter applications as other solutions?

In this chapter, we are going to build a dynamic Oracle procedure for use in validating *all* secure application roles. Individual developers will not have to provide such procedures for their applications. Instead, they will provide a list of three unique items related to their secure application role: the application ID (name), the application user, and the secure application role name. We will store those elements in a table. Then we will provide the dynamic procedure for all secure application roles, and organize our code around it.

Next we will build out some of our other tables to handle multiple applications per user, keeping application-specific security data for users separate from security data for that user for other applications.

We will also have each application send us a representative piece of code—an application-specific class, provided as an object represented by a byte array. (The details of that will become clearer.) By examining that object, we will know what application is requesting services, and in return we will provide that application with a list of the specific connection strings needed by the application. Of course, all of this is done with encryption and security in mind.

We will store the application connection strings in an Oracle table. Some procedures, functions, and Java code are required to accomplish this, and a bit more code will be required to update the data once stored. We will also support multiple versions of an application.

We are storing all the data in Oracle database, and using the security of Oracle database, but recall that we do not have our application Oracle connections until after we have passed the security checks. This is a quandary along the lines of, "which came first, the chicken or the egg?" Or in our case, "which came first, an Oracle connection or our list of Oracle connection strings (passwords)?" The answer is that we need a connection to Oracle database first, so we will use an alternate Oracle application verification, the `appver` user, to provide the necessary initial connection for all applications. The `appver` user will be a hardened user with as many restrictions as we can muster. His sole purpose is to guard and provide application-specific Oracle connection strings.

Because we will want to muster all our security forces before distributing connection strings, we will have `appver` determine the specific application, as described previously. He will also provide the single sign on (SSO) and two-factor Authentication we have discussed in earlier chapters. And he will establish an initial set of encryption keys for use in distributing and decrypting the connection strings.

Note that each new connection established for an application may require a new and distinct set of encryption keys. There are limitations to how many sets of keys we will maintain for concurrent use in the code provided in this book. We will retain the encryption keys for the appver connection, in order to continue decrypting the connection strings for this application, and we will allow each application user to use one additional set of encryption keys at a time. Additional Oracle connections without encryption can be used concurrently. However, a diligent programmer can overcome those limitations.

Secure Application Role Procedure for Multiple Applications

Thinking back on our secure application role—hrview role and the procedure that sets the role—what application-specific features did it implement? It tested for a number of things that are not application specific: IP address, time of day, two-factor authentication, and most important, SSO identity. However, a couple things were application-specific: the user who was seeking the role (appusr) and the role itself (hrview role).

Our goal at this juncture is to build a single secure application role procedure that will work for any application, enforcing all our connection security requirements, but granting the specific role required to the specific application user. We can build a procedure to handle that, but first we need a registry of those application-specific features. We will create the t application registry table in the appsec schema to hold the data. The code for creating this table is in Listing 10-1.

■ **Note** You can find the script in Listing 10-1 in the file named *Chapter10/AppSec.sql*.

Listing 10-1. *Application Registry Table of Salient Features*

```
CREATE TABLE appsec.t application registry
(
    application id VARCHAR2(24 BYTE) NOT NULL,
    app user       VARCHAR2(20 BYTE) NOT NULL,
    app role       VARCHAR2(20 BYTE) NOT NULL
);
```

We will also create a view of the table for general use. And, though not shown here, we will make the application id and app user columns a unique index and our primary key. We will not depend on that key until we get to Chapter 12. For now, suffice it to say that each application may use multiple secure application roles. We will acquire these roles by proxying through a variety of application users. So a pair of application id and app user is a unique key to acquire an app role.

While we're at it, let's insert a data record with the labels we already know: user APPUSR and role HRVIEW ROLE. We give those settings to the application id of HRVIEW, as shown here:

```
INSERT INTO appsec.v application registry ( application id, app user, app role )
VALUES ( 'HRVIEW', 'APPUSR', 'HRVIEW ROLE' );
```

We are introducing the application id column here as a handle to acquire the required role. Each application will need a unique application id, which, with a couple more additions, will allow our existing code to provide two-factor authentication, SSO, and secure application roles to multiple applications.

Rebuild Two-Factor Cache Table for Multiple Applications

It is possible that a user will want to use more than one application at a time, and we will want them to operate independently. These applications will have grants to diverse data and will use diverse sets of Oracle instances. They may also be started at the client within the ten-minute, two-factor time to live off one another.

To provide for the independent operation of these applications, we will add the `application id` column as a part of the primary key for the `t two fact cd cache` table. See Listing 10-2. In this way, we can create and assign a two-factor authentication code for each application that a user uses. This makes obvious the fact that two-factor authentication for applications is already a step away from single sign on. If your corporate computing environment enforces two-factor authentication for the initial logon (e.g., Windows password and a Secure ID token), then there is no need for two-factor authentication to applications. However, two-factor authentication for basic computer access, whether secure tokens, biometric scanners, or electronic badges, still seems more prevalent in movies than in the corporate setting. If you have a computer with biometric interface (fingerprint or facial recognition) or a secure card slot, it is not two-factor authentication as a substitute for entering a password. Only when you use it in addition to entering a password is it two-factor authentication.

So we will build two-factor authentication, independent for each application. We start by dropping the previous `t two fact cd cache` table, and creating a new one with an `application id` column.

Listing 10-2. *Redo Two-Factor Code Cache Table*

```
DROP TABLE appsec.t two fact cd cache CASCADE CONSTRAINTS;

CREATE TABLE appsec.t two fact cd cache
(
    employee id    NUMBER(6) NOT NULL,
    application id VARCHAR2(24 BYTE) NOT NULL,
    two factor cd  VARCHAR2(24 BYTE),
    ip address     VARCHAR2(45 BYTE) DEFAULT SYS CONTEXT( 'USERENV', 'IP ADDRESS' ),
    distrib cd     NUMBER(1),
    cache ts       DATE DEFAULT SYSDATE
);

CREATE UNIQUE INDEX two fact cd emp id pk ON appsec.t two fact cd cache
    (employee id,application id);
```

We also recreate a view (code not shown) of this table for general references. There is an example insert and some cache aging test code in the file *Chapter10/AppSec.sql*.

Update Two-Factor Code Functions to Use Application ID

Please refer to the full listing of the code in AppSec.sql file. We modify our existing `f is cur cached cd` function to take the `application id` as an argument and to select from `v two fact cd cache` based on the `application id`. We also update our existing `f send 2 factor` function to take the `application id` as an argument, and to pass it to the `distribute2Factor()` method.

Move Test for SSO to Separate Function

For division of labor among pieces of our code, we are going to separate the SSO process into a separate function, `f is sso`. Into this function, we pass the value for application user. Traditionally, `appusr` has been our application user, but we will pass whatever user we find registered for the application in the `t application registry` table. Recall that for SSO, the application user has to either be the connected user for our session, or the proxy user. The `f is sso` function will return the validated SSO user ID, or an empty string if invalid. Listing 10-3 shows just the signature of `f is sso`.

Listing 10-3. *Function to Test if User Passes SSO Requirements, f_is_sso*

```
CREATE OR REPLACE FUNCTION appsec.f is sso( m app user VARCHAR2 )
RETURN VARCHAR2
AUTHID CURRENT USER
AS
    return user     VARCHAR2(40) := '';
...
```

■ **Note** Procedures and functions that execute as `AUTHID CURRENT USER` are never placed in packages, and they are usually granted `EXECUTE` to `PUBLIC`.

Add an Oracle Package for Use Only by Application Security

We will be increasing the number of functions and procedures for use strictly by our application security user, `appsec`, by almost an order of magnitude, so we are going to employ a package to group and organize them. We'll call this package the `appsec only pkg` package. The package also allows us to secure the code in one location – in this case, we will not grant anyone execute on `appsec only pkg`. We will drop the functions `f is cur cached cd` and `f send 2 factor`, and move them into our package as in Listing 10-4.

Listing 10-4. *Package for Application Security Use Only*

```
DROP FUNCTION appsec.f is cur cached cd;
DROP FUNCTION appsec.f send 2 factor;

CREATE OR REPLACE PACKAGE appsec.appsec only pkg IS

    FUNCTION f is cur cached cd(
        just os user     VARCHAR2,
        m application id v two fact cd cache.application id%TYPE,
        m two factor cd  v two fact cd cache.two factor cd%TYPE )
    RETURN VARCHAR2;

    FUNCTION f send 2 factor(
        just os user     VARCHAR2,
```

```
    m application id v two fact cd cache.application id%TYPE )
RETURN VARCHAR2;
```

Add Helper Function to Get APP_ROLE

We are going to need one piece of data from the `t application registry` table, the secure application role name, when we run the procedure that sets those roles. We have already discussed the need to run our secure application role procedure as `AUTHID CURRENT USER`. We do this so we can assure the validity of the current user, not the schema owner, `appsec`. This is called invoker's rights. Also, this is the only way that roles may be set, by the `CURRENT USER`.

In order to execute the secure application role procedure, we need to grant execute on the procedure to `PUBLIC`. However, we do not want to grant data privileges to `PUBLIC` on the `v application registry` view. If we have a helper function that belongs to the same schema as the `v application registry` and the secure application role procedure, then the helper function can be executed by the procedure and read the data in the table on its behalf. PUBLIC has execute on the procedure, but no grants on the view or the functions; yet the procedure has access to the functions in order to select on the view. The result of this is that we can run a procedure to grant the role, but not expose the data used to evaluate access.

We read the application role name, based on the `application id` and `app user` from the `f get app role` function. We will add this function to our `appsec only pkg` package. See Listing 10-5.

Listing 10-5. *Helper Function to get Application Role Name*

```
FUNCTION f get app role(
    m application id v two fact cd cache.application id%TYPE,
    m app user       v application registry.app user%TYPE )
RETURN VARCHAR2
AS
    m app role v application registry.app role%TYPE;
BEGIN
    SELECT app role INTO m app role
    FROM v application registry
    WHERE application id = m application id
    AND app user = m app user;
    RETURN m app role;
END f get app role;
```

Replace Procedure for hrview_role Access with Dynamic Procedure

We will be replacing the application-specific procedure, `p check hrview access`, which set the `hrview role` with a generic procedure, `p check role access`, which will set the secure application role for any application. Each application will require an entry (one or more) in the `t application registry` table. With this new procedure, we can easily apply our SSO and two-factor authentication to multiple applications.

Code for the New Procedure

By way of cleaning up, we will drop the old `p check hrview access` procedure. That will assure us that we are using the new procedure, even for setting `hrview role`. Refer to Listing 10-6.

Our new procedure, `p check role access` looks very similar to our previous secure application role procedure. We enter this procedure with a connection that is already proxying through our application Oracle user, which we get into `app user` variable. The new procedure takes the additional `application id` argument, which it in turn passes with the `app user` identity to our new `f get app role` helper function in order to read the role name from `v application registry`. Also, instead of having the code for SSO right in this method, we pass the `app user` to our new `f is sso` function to get back the validated user.

Listing 10-6. *Dynamic Secure Application Role Procedure, p_check_role_access*

```
DROP PROCEDURE appsec.p check hrview access;

CREATE OR REPLACE PROCEDURE appsec.p check role access(
    --m_two_factor_cd      v two fact cd cache.two factor cd%TYPE,
    m application id     v two fact cd cache.application id%TYPE,
    m err no         OUT NUMBER,
    m err txt        OUT VARCHAR2 )
AUTHID CURRENT USER
AS
    return user VARCHAR2(40);
    m app user  v application registry.app user%TYPE;
    m app role  v application registry.app role%TYPE;
BEGIN
    m err no    := 0;

    m_app_user  := SYS_CONTEXT('USERENV','PROXY_USER');

    m app role  := appsec only pkg.f_get_app_role( m application id, m app user );
    return user := f is sso( m app user );
    IF( return user IS NOT NULL )
    THEN
    -- Code for two-factor Auth moved to appver login process
    --    IF( m two factor cd IS NULL OR m two factor cd = '' )
    --    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
          EXECUTE IMMEDIATE 'SET ROLE ' || m_app_role;
    --    ELSE
    --        RAISE NO DATA FOUND;
    --    END IF;
        app sec pkg.p log error( 0, 'Success getting SSO and setting role, ' ||
            SYS CONTEXT( 'USERENV', 'OS USER' ) );
    ELSE
        app sec pkg.p log error( 0, 'Problem getting SSO, ' ||
            SYS CONTEXT( 'USERENV', 'OS USER' ) );
    END IF;
EXCEPTION
    WHEN OTHERS THEN
        m err no := SQLCODE;
        m err txt := SQLERRM;
```

```
        app sec pkg.p log error( m err no, m err txt,
            'APPSEC p check role access' );
END p check role access;
/
```

Notice that the two-factor code is no longer required, and all the logic related to processing two-factor authentication has been commented. We are moving that logic to take place immediately after the application verification, the appver user, connects. The job of appver, after controlling initial application access, is to provide a list of connection strings to the application. It is in this process, a call to a new procedure, P GET APP CONNS that we will do two-factor authentication.

If we pass SSO, then we proceed to set the role to the value we looked up in v application registry. If the user's connection/session fails to pass our SSO requirements, then we log the error "Problem getting SSO," and return without setting the role. Something is seriously wrong with that user, and we don't want to deal with him at all.

Put Dynamic Secure Application Role Procedure to Work

As a one-stop-shop for all applications to gain the privileges needed to operate, p check role access needs to be executable by all. We will grant execute on p check role access to PUBLIC. The procedure runs with AUTHID CURRENT USER, so it doesn't have direct access to data in the appsec schema; however, this procedure, by virtue of its definition in the appsec schema, can execute other functions and procedures in the appsec schema, like the f get app role helper function, which provides the required data. Following is our grant of execute privileges:

```
GRANT EXECUTE ON appsec.p check role access TO PUBLIC;
```

We will add some additional auditing, because we want to see if there is a trend of errors in this procedure. We can combine this audit information with information in the v appsec errors log view. For example:

```
AUDIT EXECUTE ON appsec.p check role access
    BY ACCESS WHENEVER NOT SUCCESSFUL;
```

■ **Note** You can find a script with the preceding statements in the file named *Chapter10/SecAdm.sql*.

As you may recall, when we initially created the hrview role back in Chapter 2, we specified that it would be IDENTIFIED USING appsec.p check hrview access. We are going to have to redirect it to be identified by our new procedure. We will drop the role and recreate it. Also, we need to repeat the grants that we had made to that role as shown in Listing 10-7.

Listing 10-7. *Recreate HR View Role Identified by New Procedure*

```
DROP ROLE hrview role;
CREATE ROLE hrview role IDENTIFIED USING appsec.p check role access;

GRANT EXECUTE ON hr.hr sec pkg TO hrview role;
```

Rewrite and Refactor Method to Distribute Two-Factor Code

We are going to make one more pass at the `distribute2Factor()` method. We need to incorporate the `application id` in several places. While we are here, we are also going to *refactor* the code with a bit more security and organization.

If you look back at this method in the last chapter, you will see that we have two dynamic queries: one query we build to get data from the `HR.emp mobile nos` table and others, and another query we build to update the `v two fact cd cache` view. For security, we prefer parameterized procedures and functions to dynamic queries. This method and these dynamic queries run in the Oracle database and are unlikely to be susceptible to SQL injection, but we should consider the possibility. What would it take to do SQL injection in those queries?

The first query takes two parameters. It takes the `oraFmtSt` string for formatting dates, which is defined locally—that is tamper-proof. It also takes the `osUser` name, which is passed from `f send two factor`, which is in turn passed from our secure application role procedure and is derived from our SSO process. I can surmise that, in order to accomplish SQL injection in that dynamic query, the user would have to have an extremely bizarre user name in the operating system—not likely.

The second query (update or insert) takes the two-factor code that we generated locally (tamper-proof), the IP Address which is sensed by Oracle database (only suspect in our wildest dreams), and the employee ID, which we got from our `HR` tables and must meet rigid type constraints, `NUMBER(6)`. Once again, this is not a likely candidate for SQL injection.

Our impetus for moving these queries out of our Java code and into stored procedures, then, is not justified as a counter to SQL injection. We will do it in any case, because having the database logic in a stored procedure makes our Java code less dependent on the data organization and more tolerant of database changes. If the DBAs or our application security manager requires that data tables be changed or moved, our procedure can be modified to accommodate the changes without requiring a change to the Java code. We would prefer that database changes only affect native database structures, not Java.

Procedure to get Employee Addresses for Two-Factor Code Delivery

We build a procedure to get the pager, phone, and other numbers that we will use to distribute our two-factor authentication code. At the same time, with a single query, we can get the employee e-mail address and the IP address of the session. We will also get the cached two-factor authentication code for this user on this application, and the timestamp of the cache. Getting all these data elements at once, based on the user ID and the application ID provides sufficient data for all our two-factor code distribution tests and delivery.

Take a look at the list of parameters in Listing 10-8. You'll see that they are mostly `OUT` parameters—we are returning a lot of data. We only pass three parameters to this procedure: the user ID, the date format string we discussed in the last chapter, and the application ID that we introduced in this chapter. We will place `p get emp 2fact nos` in the `appsec only pkg` package.

Listing 10-8. *Procedure to Get Addresses for Two-Factor Code Distribution*

```
PROCEDURE p get emp 2fact nos(
    os user               hr.v emp mobile nos.user id%TYPE,
    fmt string            VARCHAR2,
    m employee id     OUT hr.v emp mobile nos.employee id%TYPE,
    m com pager no    OUT hr.v emp mobile nos.com pager no%TYPE,
    m sms phone no    OUT hr.v emp mobile nos.sms phone no%TYPE,
```

```
    m sms carrier url OUT hr.v sms carrier host.sms carrier url%TYPE,
    m email           OUT hr.v employees public.email%TYPE,
    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
    WHERE m.user id = os user
    AND e.employee id =  m.employee id
    AND s.sms carrier cd (+)=  m.sms carrier cd
    AND c.employee id (+)= m.employee id
    AND c.application_id (+)= m_application_id;
EXCEPTION
    -- User must exist in HR.V EMP MOBILE NOS to send 2Factor, even to email
    WHEN OTHERS THEN
        m err no := SQLCODE;
        m err txt := SQLERRM;
        appsec.app sec pkg.p log error( m err no, m err txt,
            'app sec pkg.p get emp 2fact nos' );
END p get emp 2fact nos;
```

The query in Listing 10-8's procedure is almost identical to the select query that we had in the previous version of the `distribute2Factor()` method. The only difference is the addition of the last line: `AND c.application id (+)= m application id`. With that addition, we are selecting from the two-factor cache for this user, and only for the specific application ID. Again, the (+) symbol indicates an outer join, so we will return the primary data (e.g., pager number), even if no two-factor code already exists in the cache for this user and this application.

Stored Procedure to Update Two-Factor Code Cache

The second Oracle statement that we are moving out of Java as dynamic SQL into an Oracle stored procedure was what we used in `distribute2factor()` for updating the two-factor cache. Our parameters for this procedure are primarily `IN` parameters. We pass in the user ID and the application ID for which the two-factor code was generated. We also pass in the two-factor code and the distribution code—a numeric value indicating which routes were used for distribution of the two-factor code. Listing 10-9 shows this procedure. We will place `p update 2fact cache` in the `appsec only pkg` package.

Listing 10-9. *Procedure to Update Two-Factor Code Cache*

```
PROCEDURE p update 2fact cache(
    m employee id        v two fact cd cache.employee id%TYPE,
    m application id     v two fact cd cache.application id%TYPE,
    m two factor cd      v two fact cd cache.two factor cd%TYPE,
    m distrib cd         v two fact cd cache.distrib cd%TYPE,
    m err no         OUT NUMBER,
    m err txt        OUT VARCHAR2 )
IS
    v count          INTEGER;
BEGIN
    m err no := 0;
    SELECT COUNT(*) INTO v count
        FROM v two fact cd cache
        WHERE employee id = m employee id
        AND application id = m application id;
    IF v count = 0 THEN
        INSERT INTO v two fact cd cache( employee id, application id,
            two factor cd, distrib cd ) VALUES
        ( m employee id, m application id, m two factor cd, m distrib cd );
    ELSE
        UPDATE v two fact cd cache SET two factor cd = m two factor cd,
            ip address = SYS CONTEXT( 'USERENV', 'IP ADDRESS' ),
            distrib cd = m distrib cd, cache ts=SYSDATE
        WHERE employee id = m employee id
        AND application id = m application id;
    END IF;
EXCEPTION
    WHEN OTHERS THEN
        m err no := SQLCODE;
        m err txt := SQLERRM;
        appsec.app sec pkg.p log error( m err no, m err txt,
            'app sec pkg.p update 2fact cache' );
END p update 2fact cache;
```

You can see that we have three Oracle statements in the body of this procedure: a `SELECT`, an `INSERT`,and an `UPDATE`. If you recall, in the `distribute2factor()` method in the last chapter, we only required two statements. There we attempted an update and read the value of a returned integer. If the return value was 0, then no records were affected by the update, and we did an insert. Here, however, we must do the `SELECT COUNT` manually to see if there are any records to update. If the count is 0, we do the `INSERT`, or else we do the `UPDATE`.

In each of these statements, you can see that we are dealing with records associated with both the user (employee) id and the application ID. A user may have multiple two-factor codes in the cache for multiple applications.

Changes to the Method to Distribute Two-Factor Codes

The changes in the `OracleJavaSecure.distribute2Factor()` method have mostly to do with calling and receiving output parameters from `p get emp 2fact nos` and `p update 2fact cache`. One other addition

is the receipt and use of the applicationID argument as shown in Listing 10-10. We are not going to go into the details of our call to our new procedures—you have seen similar calls before.

Listing 10-10. *Header for Method to distribute Two-Factor Code*

```
    private static String applicationID = null;

    public static final String distribute2Factor( String osUser, String applicationID )
        throws Exception
    {
        // Set class static member equal to what passed here from Oracle
        OracleJavaSecure.applicationID = applicationID;
```

■ **Note** You can find the code just described, which you can see in Listing 10-10, in the file named *Chapter10/orajavsec/OracleJavaSecure.java.*

One change that might go unnoticed is suggested by our setting of the static class member, applicationID, from what we receive in this procedure. In the last chapter, we had a small issue with the twoFactorAuth member: we generated it in the distribute2Factor() method, used it there and handed it to the distribToSMS(), distribToPagerURL(), and distribToEMail() methods. Additionally, we entered it on the command line in the client, handed it to our secure application role procedure, and eventually handed it to the f is cur cached cd function. Thus, we had two different points of entry—this will continue. The problem is that we had two different references for the same thing in OracleJavaSecure; that is unneeded, because we can call it the same thing whether generated on Oracle database for distribution or entered on the client for return to Oracle database for testing. It is in our benefit to move that member to a static class member and have a standard location to refer to. Then we can stop passing it from method to method and just refer to it locally in each method.

We have a couple more static class members that we will similarly put to use in this chapter, the first of which is coming into play here in the distribute2Factor() method. You can see that we pass the applicationID to this method and use it to set a static class member. We will refer to this in many places, and be thankful we don't have to list it in method arguments and pass it around from method to method.

We pass applicationID into the distribute2Factor() method because this is the first Java method called on the Oracle database. On the client side, the applicationID is entered by the user and passed to OracleJavaSecure from the specific application that is using our services. We will discuss that process later in this chapter.

Update to Two-Factor Distribution Formats

Now, because we will be sending two-factor authentication codes to the user for multiple applications, we will have to identify them in the messages we send. We will use the application ID to identify what app the code is used for. We will place the words "for APPID" on the Subject line and in the message body, where "APPID" is the specific application ID we are referencing. Here is an example.

```
From: response@org.com
To: 8005551212@txt.att.net
```

```
Subject: Response for HRVIEW
1234-5678-9012 for HRVIEW
```

Application Authorization Overview

We are going to implement this entirely with Oracle database as the backend, of course running with the Oracle JVM. That means we have to deal with one important security aspect. To talk to Oracle database at all, we need to connect with a user and password. We will spend significant time addressing that issue.

In a nutshell, here is how we will do Application Authorization, which we will also refer to as Application Verification:

1. We will first connect to Oracle, proxying through a new user, `appver`.
2. As in the past, we will need to pass the SSO and two-factor authentication requirements.
3. Once we have attained two-factor authentication, we exchange encryption keys.
4. We also retrieve an encrypted list of connection strings that we can use in our current application.
5. When we use one of those application connection strings, we again assure our SSO and we exchange additional encryption keys to query encrypted data, as before; however, we do not re-do our two-factor authentication.

All of this looks well and good, but the question arises: how do we know what application is being used? Well, we certainly don't want to merely take the application's word for it. A standard assumption in secure computing is that given an opportunity, a hacker's code will lie—a sort of application identity theft. We are distributing a list of connection strings, including user names and passwords, to known applications. They are encrypted, but we don't want to hand them to just anyone.

In order to assure an application is who it says it is, we are going to ask the application to give us a piece of itself, one which we have already received and registered. We will compare what the application provides with what we have registered, and if they are "equal," then we recognize the application.

The piece of an application that we demand be presented is an inner class. This class must have the same name as the one we registered. We may register multiple versions of the same inner class, to handle application upgrades, but connection strings need to be recreated or duplicated to the new version.

When I say that we assure that the inner classes are "equal," that is the litmus test. The default equality test, which we do not override, is that the classes are equivalent; they reside at the same memory address and are the same class. We can also say `class1 == class2`.

In a JVM, classes are modeled in memory, but only a single place in memory is required to model a specific class. There can be multiple instances, but they all use the same model. When we say `class1.equals( class2 )`, the class has to be the same model. I'll give you more insight: if these classes (the one the application provides and the one we have registered) claim to be the same—same package and same class name—but are not, then even before we get to the `.equals()` method, we will see an exception. When we attempt to deserialize a `Class` based on the serialized object in hand, if it is not the same as one we've already instantiated, then it looks like a square peg trying to fit in a round hole. Specifically, an `InvalidClassException` is thrown. A JVM only has room for one name to class instance image relationship. Try to introduce another, different class as the same name and the JVM will reject it.

Let me be subtle and say that you will probably find ways that this will need to be improved. One obvious way to improve it would be to run our applications on a server, perhaps as web applications, so

we do not need to deal with client-side application authentication. However, even in that case, you may still want to have application authentication for server applications in order to restrict Oracle connection strings to specific applications.

User for Application Authorization

We need a doorkeeper. He will be our bouncer, rejecting the riff-raff and troublemakers. We will call him our application verification, `appver` user. In this analogy, the applications (not people) are our clients. Some applications are permitted because they have pre-registered and present their identities and submit to and pass identity checks.

In addition to guarding the entrance, `appver` provides every successful entrant with the keys needed to enter the inner doors where they are allowed to go. Again in our analogy, the keys are Oracle connection strings that have user ids and passwords to connect to Oracle database. These are not personal users and passwords, but application passwords, permitting secure application roles with access to application data.

Like a doorkeeper, `appver` is available to all and has sufficient information to grant passage and direction. He also has sufficient power to prevent passage, as needed.

A New Profile with Limits and Unlimited

Normally, we would limit user accounts for security, but in setting up our application authorization, we are going to need some flexibility. We will assign the limits and permissions for this account through a unique profile, `appver prof`, shown in Listing 10-11.

Listing 10-11. *Profile for Application Authorization*

```
CREATE PROFILE appver prof LIMIT
    CONNECT TIME          1
    IDLE TIME             1
    SESSIONS PER USER     UNLIMITED
    PASSWORD LIFE TIME    UNLIMITED
    FAILED LOGIN ATTEMPTS UNLIMITED;
```

■ **Note** You can find this script in the file named *Chapter10/SecAdm.sql*.

Contrary to our ordinary standards, our application verification, `appver` user, will have a password that will never expire. We will also allow this user to type in the wrong password an unlimited number of times without locking it out (keeping it from further login attempts, even with the correct password). And, we will allow this account an unlimited number of concurrent sessions.

I know that all sounds backwards from security, but we have the potential of hundreds of persons attempting to access applications, verified by this user. We need to tightly control when this password expires—we will periodically, manually reset the password—some file distribution will probably be required to the clients. However, we don't want it to expire automatically. Also, we do not know how many people will concurrently want to authorize for applications, hence the unlimited sessions. Perhaps down the road, once we have some history, we will know what the correct number of sessions is.

Sure we can say unlimited sessions, but in reality there is a limit. The hard limit is the number of processes that the Database was created to service. The default number of processes is 150. Looking ahead, when we create a dedicated database for application authentication in Chapter 11, we will bump the number of processes up to 500.

The unlimited failed login attempts are a bit harder to justify, because it gives a hacker a wide open door to use brute-force attack to guess the password. The alternative, however, is the potential for a slew of broken applications. If a hacker, or errant application, made several attempts to log in with the wrong password and locked out the user, all our applications that depend on this user for authorization would fail until the account was reset.

We are going to limit this account as much as possible. There will be only a few privileges we will give it. It will need to pass SSO, two-factor authentication, encryption, and application authorization—the reason for its existence. A couple of the limits we will set for this user will be set through parameters in the `appver prof` profile. We will only allow this account a one minute connect time, and one minute of idle time (the minimums).

Application Verification User

It's time to create our application verification user, which we will name `appver`. Listing 10-12 shows the commands to create the appver user. We assign a password to this user, but for now, we are treating it more like a line of code or an address. It gets people to the workplace, but doesn't do any work, itself. For now, this password will be hard-coded into the `OracleJavaSecure` class, on the client. In Chapter 11, we will obfuscate and encrypt the password. In any case, please assign a complex password to `appver`.

Listing 10-12. *Create Application Verification User*

```
CREATE USER appver
    IDENTIFIED BY password
    QUOTA 0 ON SYSTEM
    PROFILE appver prof;

GRANT create session role TO appver;
```

Notice that we assigned the `appver prof` profile to `appver`. We also give `appver` no space, 0 QUOTA for storage. Finally, we grant the `create session role` to `appver` so that he can connect to Oracle database.

The Application Verification Logon Trigger

We are going to create a logon trigger for the `appver` account. We have already seen database triggers, but this one is different—it defines an action we will have Oracle database take whenever a user logs onto the `appver` schema. Listing 10-13 shows that our logon trigger simply calls a procedure, `p appver logon`.

Listing 10-13. *Logon Trigger for Application Verification*

```
CREATE OR REPLACE TRIGGER secadm.t screen appver access AFTER LOGON ON appver.SCHEMA
BEGIN
    appsec.p_appver_logon;
END;
/
```

Application Verification Logon Procedure

The procedure for our appver logon trigger, `p appver logon` is shown in Listing 10-14. It would have been nice if we could have done a complete SSO check during the logon trigger, but alas, the proxy session and the `CLIENT IDENTIFIER` setting in the `USERENV` are not available at logon time. However, we can still assure that the session user is appver (who else could it be?), and that our IP Address is acceptable. We also call a function, `f is user` to assure that the `os user` is also a database user. That may be the most significant test since it is also what we will assure for our proxy login.

Listing 10-14. *Application Verification Logon Procedure*

```
CREATE OR REPLACE PROCEDURE appsec.p appver logon
AUTHID CURRENT USER
AS
    just os user    VARCHAR2(40);
    backslash place NUMBER;
BEGIN
    just os user := UPPER( SYS CONTEXT( 'USERENV', 'OS USER' ) );
    backslash place := INSTR( just os user, '\', -1 );
    IF( backslash place > 0 )
    THEN
        just os user := SUBSTR( just os user, backslash place + 1 );
    END IF;
    -- For logon trigger - limited SSO, no PROXY USER and no CLIENT IDENTIFIER
    IF( SYS CONTEXT( 'USERENV', 'SESSION USER' ) = 'APPVER'
    AND( SYS CONTEXT( 'USERENV', 'IP ADDRESS' ) LIKE '192.168.%' OR
        SYS CONTEXT( 'USERENV', 'IP ADDRESS' ) = '127.0.0.1' )
    -- Requirements must be applicable to all applications - time may not be
    --AND TO CHAR( SYSDATE, 'HH24' ) BETWEEN 7 AND 18
    -- Assure that OS USER is a database user
    AND( appsec only pkg.f_is_user( just_os_user ) = 'Y' ) )
    THEN
        app sec pkg.p log error( 0, 'Success APPVER logon, ' || just os user );
    ELSE
        app sec pkg.p log error( 0, 'Problem getting APPVER logon, ' || just os user );
        --just_os_user := sys.f_get_off;
        -- This causes logon trigger to fail -- so not connected to Oracle
        RAISE_APPLICATION_ERROR(-20003,'You are not allowed to connect to the database');
    END IF;
END p appver logon;
/
```

We run this logon trigger procedure as `AUTHID CURRENT USER`; that is with invoker's rights. That's the only way we can accurately gauge the user's identity—similar to the way it is for secure application role procedures. So, we need to grant execute on this procedure to `PUBLIC`:

```
GRANT EXECUTE ON appsec.p appver logon TO PUBLIC;
```

Get Off Function

There is always a longing to have complete control. It would be wonderful to be able to spot a problem in our logon trigger and immediately kill the session. Do you see the commented line saying `just os user`

`:= sys.f get off` in our logon trigger procedure, `p appver logon`, Listing 10-14? It is a nice idea that won't work. The `f get off` function shown in Listing 10-15 could accomplish the task for us; however, Oracle Database does not allow us to kill the current session.

Listing 10-15. *Non-functional Kill Switch, f_get_off*

```
CREATE OR REPLACE FUNCTION sys.f get off
RETURN VARCHAR2
AS
    PRAGMA AUTONOMOUS TRANSACTION;
    p sid v$session.SID%TYPE;
    p serial v$session.serial#%TYPE;
BEGIN
    p sid := SYS CONTEXT( 'USERENV', 'SID' );
    SELECT serial# INTO p serial
    FROM v$session
    WHERE sid = p sid;
    EXECUTE IMMEDIATE 'ALTER SYSTEM KILL SESSION ''' ||
                p sid || ',' || p serial || '''';
    RETURN 'OFF';
END f get off;

GRANT EXECUTE ON sys.f get off TO appsec;
```

■ **Note** You can find Listing 10-15's function in the file named *Chapter10/Sys.sql*.

The heart of the function is in the `EXECUTE IMMEDIATE` command. An alternative along these lines that would work would be if we inserted the `SID` and `SERIAL#` to be killed into a table. Then using an independent, scheduled task, read the table and kill the sessions, then delete the records from the table.

Anyway, manhandling in the most direct way is not going to work. Just as well, because our call in `p appver logon` to `RAISE APPLICATION ERROR` does the same thing. Because we are in a logon trigger, when we raise an exception, the logon fails.

Function to Find Database User

There is a function that we have, for use in the `appver` logon trigger procedure, `f is user`, which tests whether the OS user is also a database user. This is an important test, because it enforces a portion of our SSO requirements—if we haven't created an Oracle user with the same name as this person's operating system user name, then he cannot use our applications. This function, Listing 10-16, will be added to the package, `appsec only pkg`.

Listing 10-16. *Function to Find Database User*

```
FUNCTION f is user( just os user VARCHAR2 )
RETURN VARCHAR2
AS
    return char VARCHAR2(1) := 'N';
    v count     INTEGER;
BEGIN
    SELECT COUNT(*) INTO v count
    FROM sys.all_users
    WHERE username = just os user;
    IF v count > 0 THEN
        return char := 'Y';
    END IF;
    RETURN return char;
END f is user;
```

■ **Note** Return to file *Chapter10/SecAdm.sql* for reference in this discussion.

Notice from where we get the indicator that the user is an Oracle user. We read from the Oracle data dictionary view, `SYS.ALL USERS`. That view is granted select to `PUBLIC`. I believe this is a bit of a security problem. If a hacker gets access to any Oracle user account, he can read the `ALL USERS` view and *get a list of all the user names* he might try to access. There are other views of the data dictionary that are likewise granted for select by `PUBLIC`, which I believe oversteps security. Another particular view is `SYS.ALL SOURCE`, which lists the entire body of every stored procedure in every schema, and other code to which a user is granted execute. Letting a hacker see our code (whether he's a legitimate but rogue user or an infiltrator) is inviting further compromise.

When we create a dedicated database for application authentication in Chapter 11, we will revoke select by `PUBLIC` for these particularly sensitive views. There are a few additional views that we will also remove from PUBLIC access.

Proxy Through Application Verification and Other Proxies

Finally, with regard to our application verification user, we need to permit every user to proxy through `appver`. Our logon trigger will not see the proxy, but when we execute the procedure to get the secure application role related to the current application, we will use our proxy test as part of SSO. For example:

```
ALTER USER osuser GRANT CONNECT THROUGH appver;
```

Do not forget to create an Oracle user for every OS user that you want to give access to your applications. And grant that each user may proxy through `appver`. Also for each OS user, grant that her account may proxy through the role associated with the specific application.

For example, to give an OS user named "coffin" access to the `hrview` application, you'd need to execute these commands:

```
CREATE USER coffin IDENTIFIED EXTERNALLY;
GRANT CREATE SESSION ROLE TO coffin;
```

```
ALTER USER coffin GRANT CONNECT THROUGH APPVER;
-- APPUSR is the account that gets access to HRVIEW ROLE
ALTER USER coffin GRANT CONNECT THROUGH APPUSR;
```

Additionally, for user coffin to pass two-factor authentication, he will need entries in both `HR.EMPLOYEES` and `HR.emp mobile nos`.

Auditing Application Verification

It would be nice to know everything that the `appver` user does—there should only be a select number of statements he makes. However, we anticipate he will get called many, many times, and we do not want to audit all the legitimate calls he makes. The first clue that `appver` is doing something unofficial will probably be when he tries to select data. So we will use the audit log, by the uncommented commands in Listing 10-17 to watch for that.

Listing 10-1., *Auditing Application Verification*

```
--AUDIT ALL STATEMENTS BY appver BY ACCESS; -- WHENEVER SUCCESSFUL;
AUDIT SELECT TABLE BY appver BY ACCESS;

AUDIT EXECUTE PROCEDURE
    BY appver
    BY ACCESS
    WHENEVER NOT SUCCESSFUL;
```

Also, we might catch some application errors and attempted abuse by auditing when `appver` fails to execute procedures. So we audit when those calls are `NOT SUCCESSFUL`. We wouldn't want to audit successful procedure calls, because we know he will be calling procedures, and we hope he succeeds.

Structures for Application Authorization

We discussed one of the tasks that `appver` accomplishes—comparing an inner class object provided by the application with one that has been registered. Registration occurs when the inner class gets inserted on first sight (when there is no existing registry entry for the application). That's right: the first time you (a new application) show up, we write your name in the guest book and save the inner class you provided. Your application had to leap several obstacles to get this far, and we give you credit for that. However, at this point, it's not much more than a mug shot.

Your application has to come back with the same identity in order to register your connection strings. You also have to show up with the same identity from now on to get those connection strings returned to your application.

More Space for Application Security

Because we are committing to store a couple objects per application that use these services, we should give a bit more space to the `appsec` schema to use for storage. Execute this `ALTER USER` command to accomplish that:

```
-- Increase quota to hold app verification data
ALTER USER appsec DEFAULT TABLESPACE USERS QUOTA 10M ON USERS;
```

Application Connection Registry Table

We will create a table (see Listing 10-18) to hold the object each application provides, as a RAW data type. It needs to be less than 2K to be stored as a RAW, so developers shouldn't expand on the template class that we provide to them. Along with each object, we will store a binary large object (BLOB) data type that holds a list of the associated connection strings.

It is difficult to index and select on RAW or BLOB types, as you might imagine. So we are going to index on the class name and class version as VARCHAR2 data types. These identifiers can be garnered from the object, so we never pass them: passing the object is sufficient for us to find out who the application is claiming to be.

Setting the table of Oracle connection strings (the BLOB) is not a one-time occurrence; they can be updated. So we include an update dt column in our table to keep track of that. We use the empty_blob() directive to allocate a BLOB locator address by default, pointing at no particular BLOB, but an address nonetheless—not a null.

Listing 10-18. *Application Connection Registry Table*

```
CREATE TABLE appsec.t app conn registry
(
    class name      VARCHAR2(2000) NOT NULL,
    class version   VARCHAR2(200) NOT NULL,
    class instance  RAW(2000),
    update dt       DATE DEFAULT SYSDATE,
    connections     BLOB DEFAULT EMPTY BLOB()
);
```

■ **Note** You can find this script in the file named *Chapter10/AppSec.sql*.

We create an index and primary key on class name and class version, not shown. We also create a view of this table for regular reference.

A Set of Connection Strings for an Application

On the client side, we are going to handle our table of connection strings as a HashMap, which we will call connsHash. You must assure that it is marked private so only the OracleJavaSecure class can see it. Here is the declaration:

```
private static HashMap<String, RAW> connsHash = null;
```

We will have exchanged keys with the Oracle database, so we can receive this table of connection strings encrypted with the shared password key. We will only decrypt them as needed, creating a connection and then freeing the connection string for garbage collection (retaining no class member reference to it.) Note that until the garbage collector runs (automatically, self-scheduling), the clear-text connection string will be in the machine memory, but not easily retrieved, even with a debugger.

A HashMap is a Collection class that has a key and value relationship. It is like a two-column table with unique keys and associated values. Both the keys and values are Java objects, not primitives. In this case, our keys are Strings, and our values are RAWs. Specifically, our RAW values are the encrypted form of