```
    secret pass count :=
        appsec.app sec pkg.f get crypt secret count( ext modulus, ext exponent );
    secret pass :=
        appsec.app sec pkg.f get crypt secret pass( ext modulus, ext exponent );
    secret pass algorithm :=
        appsec.app sec pkg.f get crypt secret algorithm(ext modulus, ext exponent);

    OPEN resultset_out FOR SELECT

        employee id,
        first name,
        last name,
        email,
        phone number,
        hire date,
        job id,
        appsec.app sec pkg.f_get_crypt_data( TO CHAR( salary ) ),
        appsec.app sec pkg.f_get_crypt_data( TO CHAR( commission pct ) ),
        manager id,
        department id
    FROM employees;
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,
            'HR p select employees sensitive' );
END p select employees sensitive;
```

Filling the RESULTSET_TYPE

In the middle of the `p select employees sensitive` procedure, we open the `RESULTSET TYPE` to gather a `CURSOR` from a query. Note that we don't actually transmit all the data when we return to the client; rather, we provide the client with a handle for the `CURSOR` so that the client can gather and process rows of data, one at a time.

The query we are using selects all the columns from the `EMPLOYEES` table. Notice in Listing 7-10 that we encrypt the `SALARY` and `COMMISSION PCT` with these calls:

```
appsec.app sec pkg.f get crypt data( TO CHAR( salary ) ),
appsec.app sec pkg.f get crypt data( TO CHAR( commission pct ) ),
```

Our encryption methods require that we pass the data in for encryption using a `String`. Both `SALARY` and `COMMISSION PCT` are number columns, so we first convert them to `VARCHAR2` and then pass them to our Application Security Java Stored Procedure (function), `appsec.app sec pkg.f get crypt data`.

That function returns a `RAW` type that holds the encrypted data. The client will decrypt the data back to a clear-text `String`. And we will convert the data back to its original type (`Date`, number, etc.), as needed at the client.

You're probably asking, "But can't we encrypt non-`String` data?" The answer is yes. Actually, we can encrypt anything that can be represented as a `byte` array, which is really anything, after some conversion. However, if you can see the data on a screen or print it out, then you can also represent the data as a `String`, and it's often clearer when we convert to / from `String`s and often the case that we eventually want a `String` anyway.

■ **Note** At the end of this chapter, you will have a solid foundation to extend the encryption we have built here. You'll be able to extend what you've learned in order to encrypt objects or BLOBS, or other types of data.

Logging the Error Message

At the end of our procedure, we catch any Oracle exceptions and log the error. We call our new `p log error` procedure. We will log the error in the `appsec` schema so our application security manager can catch errors in the `appsec` structures, and can assist in debugging problems in individual applications using those structures. We will not strand the application developer but will provide assistance in the debugging effort.

Selecting All Data as a Single Sensitive String

`HR` has implemented another method that encrypts all the selected data, not as individual columns, but as one long, concatenated `VARCHAR2` per row. The only difference between this procedure, `p select employees secret` shown in Listing 7-11, and the last one we looked at is the definition of the `RESULTSET OUT`.

Listing 7-11. *Portion of Procedure to Encrypt All Data as Single String, `p select employees secret`*

```
OPEN resultset out FOR SELECT
    appsec.app_sec_pkg.f_get_crypt_data(
        TO CHAR( employee id ) ||', '||
        first name ||', '||
        last name ||', '||
        email ||', '||
        phone number ||', '||
        TO CHAR( hire date ) ||', '||
        job id ||', '||
        TO CHAR( salary ) ||', '||
        TO CHAR( commission pct ) ||', '||
        TO CHAR( manager id ) ||', '||
        TO CHAR( department id )
    )
FROM employees;
```

The double-pipe characters "||" are the symbol used in Oracle database to concatenate text. Notice that we call the `TO CHAR` function for columns that are not already of type `VARCHAR2`. After we've concatenated all of these columns together, we pass the resulting `VARCHAR2` to the `f get crypt data` function for encryption and we return a single `RAW` per row in `RESULTSET OUT`.

Note that at the client we may have to parse the data, once we decrypt it, to get the individual columns. We have used a comma as a separator between columns, but parsing on commas presumes that no commas exist in our data. Each application will have to plan their use of our Application Security structures, and the best approach to provide their data in a form appropriate for use by their client. This concatenated format would probably be better for a client that doesn't need individual elements of a record.

Selecting Sensitive Data for an Employee ID

We will explore one more example of procedures to select encrypted data from HR. This Oracle procedure, p select employee by id sens, is almost identical to the previous two procedures, except that it also takes a parameter representing a single EMPLOYEE ID. That is shown in Listing 7-12.

Listing 7-12. *Portion of Procedure to Select Sensitive Data by ID, p select employee by id sens*

```
m employee id    employees.employee id%TYPE
...
OPEN resultset out FOR SELECT
    employee id,
    ...
FROM employees
WHERE employee_id = m_employee_id;
```

The query for the resultset out parameter selects data where the EMPLOYEE ID equals that input parameter.

This procedure should only return one row of data.

Revising Procedure to Get Shared Passphrase

We saw the p get shared passphrase procedure in Chapter 6. We are embellishing it a bit in this chapter with error logging. Error logging can assist application security in supporting the application developer.

The biggest change is that we are pulling p get shared passphrase from the app sec pkg package into our individual application package, hr sec pkg. We have it in hr sec pkg now so that our client application, which may be running with hr view role, can execute the procedure. We allow HR to execute the app sec pkg structures, but we don't allow hr view to do so. So, hr view executes HR structures and HR structures execute appsec structures.

We call p get shared passphrase and follow up with a call to our new OracleJavaSecure.makeDESKey() method to complete a key exchange and build the shared secret password key. We must do this before attempting data updates.

Updating Sensitive Data Columns in EMPLOYEES

We are at the point now where we can implement encrypted updates to our data. We will define a procedure p update employees sensitive shown in Listing 7-13, in the package hr sec pkg to take data for all the columns of the EMPLOYEES table. For the sensitive columns, we will be submitting RAW types that encapsulate the encrypted data. The only IN parameters are the table column data, and the only OUT parameters are the error number and text. Notice what is missing here—there are no parameters representing our encryption keys. We have to assume that key exchanges have already taken place. If we have not already exchanged keys in the current Oracle session, then the user application is trying to submit unencrypted data in the fields that require encryption, or they are encrypting the data with keys from a different session; and the Oracle database will not be able to decrypt the data.

We are defining our parameter types using an anchored datatype form that refers to the definition of the original data. We anchor this datatype declaration to a previous definition. For example, in this declaration:

```
m employee id        employees.employee id%TYPE,
```

We are saying that the `m employee id` parameter is of the same type as the `EMPLOYEE ID` column in the `EMPLOYEES` table. We will use this form of "type specification by reference" whenever appropriate to further establish the relation between the data we are receiving and the table it is destined for. This practice is helpful for at least two reasons. The first is that our procedure will only accept data that is appropriate for the field into which it will be inserted or updated. This is a further protection against a SQL injection attack (see a detailed discussion in the next section). The second reason anchoring datatypes is good is that we can change the definition of that column in the table without having to also change this procedure.

Listing 7-13. *Update Sensitive Data in the Employees Table, `p update employees sensitive`*

```
PROCEDURE p update employees sensitive(
    m employee id        employees.employee id%TYPE,
    m first name         employees.first name%TYPE,
    m last name          employees.last name%TYPE,
    m email              employees.email%TYPE,
    m phone number       employees.phone number%TYPE,
    m hire date          employees.hire date%TYPE,
    m job id             employees.job id%TYPE,
    crypt_salary         RAW,
    crypt_commission_pct RAW,
    m manager id         employees.manager id%TYPE,
    m department id      employees.department id%TYPE,
    m err no         OUT NUMBER,
   m err txt        OUT VARCHAR2 )
IS
    test emp ct      NUMBER(6);
    v salary         VARCHAR2(15); -- Plenty of space, eventually a NUMBER
    v commission pct VARCHAR2(15);
BEGIN
    m err no := 0;
    v salary := appsec.app sec pkg.f get decrypt data( crypt salary );
    v commission pct :=
        appsec.app sec pkg.f get decrypt data( crypt commission pct );
    SELECT COUNT(*) INTO test emp ct FROM employees WHERE
        employee id = m employee id;
    IF test emp ct = 0
    THEN
        INSERT INTO employees
            (employees seq.NEXTVAL, first name, last name, email, phone number,
            hire date, job id, salary, commission pct, manager id, department id)
        VALUES
            (m employee id, m first name, m last name, m email, m phone number,
            m hire date, m job id, v salary, v commission pct, m manager id,
            m department id);
    ELSE
        -- Comment update of certain values during testing - date constraint
        UPDATE employees
        SET first name = m first name, last name = m last name, email = m email,
            phone number = m phone number,
            -- Job History Constraint -- hire date = m hire date, job id = m job id,
            salary = v salary, commission pct = v commission pct,
```

```
                manager id = m manager id
                -- Job History Constraint -- , department id = m department id
            WHERE employee id = m employee 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,
                'HR p update employees sensitive' );
    END p update employees sensitive;

END hr sec pkg;
/
```

Procedure Variables and Data Decryption

We cannot modify the `IN` parameters like we did with `OUT` parameters in previous example procedures, but we want to capture the output of our decryption, so we establish a couple procedure variables: `v salary` and `v commission pct`. We also define a numeric procedure variable named `test emp ct`:

```
    test emp ct      NUMBER(6);
    v_salary         VARCHAR2(15); -- Plenty of space, eventually a NUMBER
    v_commission_pct VARCHAR2(15);
```

Our procedure body, under the `BEGIN` header, includes two calls to the `f get decrypt data` Oracle function which will return a `VARCHAR2` data type representing the `SALARY` and the `COMMISSION PCT`. Note again that use of this procedure assumes that you have already done key exchange:

```
    m err no := 0;
    v salary := appsec.app sec pkg.f_get_decrypt_data( crypt salary );
    v commission pct :=
        appsec.app sec pkg.f_get_decrypt_data( crypt commission pct );
```

Inserting or Updating

I have found that a multifunction procedure is often the best choice for managing data inserts and updates. At most, we grant `SELECT` privilege to a view of the data and `EXECUTE` privilege to the management procedure. We regularly pass a transaction code (usually A, U, or D) that indicates whether we are going to be inserting (adding), updating, or deleting a record. For a simple, bi-functional procedure (insert or update), we do not need a transaction code, but can examine the data and 1) update an existing record or 2) insert a new record if no existing record matches the key columns.

In our management procedure body, we will populate `test emp ct` by using `SELECT INTO` syntax with a count of the number of employees whose `EMPLOYEE ID` matches the `m employee id` that is being passed in for update. There should never be more than one, so we expect a value of 0 or 1 from the count.

```
    SELECT COUNT(*) INTO test emp ct FROM employees WHERE
        employee id = m employee id;
```

We then test to see if `test emp ct` is 0 – if so, we do an `INSERT`; if not, an `UPDATE`:

```
    IF test emp ct = 0
```

```
THEN
    INSERT INTO employees
        ...
ELSE
    UPDATE employees
    SET first name = m first name, last name = m last name, email = m email,
        phone number = m phone number, hire date = m hire date,
        -- Job History Constraint -- job id = m job id,
        salary = v salary, commission pct = v commission pct,
        manager id = m manager id
        -- Job History Constraint -- , department id = m department id
    WHERE employee id = m employee id;
END IF IF;
```

Integrity Constraint on Employees Table

You will see in the previous code that we skip updating two of the columns: JOB ID and DEPARTMENT ID. The reason is that there is an existing trigger on the EMPLOYEES table that inserts a record in JOB HISTORY when either of those 2 columns in an EMPLOYEES' record is updated. The trigger code is shown in Listing 7-14.

Listing 7-14. An Existing Integrity Constraint on Employees Table, HR.update job history

```
CREATE OR REPLACE TRIGGER HR.update job history
  AFTER UPDATE OF job_id, department_id ON HR.EMPLOYEES FOR EACH ROW
BEGIN
  add_job_history(:old.employee id, :old.hire date, sysdate,
                  :old.job id, :old.department id);
END;
```

You can see the trigger in Listing 7-14 calls a procedure, add job history. All that procedure does is INSERT a record into the JOB HISTORY table. However, the JOB HISTORY table includes a UNIQUE index on (EMPLOYEE ID, **START_DATE**).

To summarize the problem: if you try to update an EMPLOYEES' JOB ID, or DEPARTMENT ID more than once a day, it fails because the trigger cannot insert another record in the JOB HISTORY table for that same user on the same day. This is a business rule that the developers of the HR sample schema are enforcing through a UNIQUE index—employees can't change jobs more than once a day.

UPDATE TRIGGER SYNTAX

I'd like to point out one aspect of the trigger syntax. Do you see the :old. prefix in Listing 7-14? That prefix indicates that we are using the value that already exists in the table. Because this is an AFTER UPDATE trigger, the value that exists in the table is the same value we submitted in the update. This runs after the data has been updated.

Often triggers can be used to test, filter, and manipulate data being submitted to a table before it is stored. For example, if I am updating the last name of an employee, I might say:

```
update employees set last name = 'coffin' where employee id = 700;
```

This may be a problem if we require all our `last name` entries to be in upper case! We can catch and correct this problem with a `BEFORE UPDATE OR INSERT` trigger. In the middle of our trigger, we might say:

```
:new.last name := upper(:new.last name);
```

This will capitalize the new value we are submitting for last_name. If we want to complain that the user is trying to update the last name with the same last name that already exists, we can compare the capitalized new value with the old value, like this:

```
IF :new.last name = :old.last name
THEN
     Raise Application Error(-20000, 'Same last name as before!');
```

A `BEFORE UPDATE` trigger has access to both the existing value in the database (`:old`) and the new value being submitted (`:new`). That ability is often utilized in triggers.

Avoiding SQL Injection

If a computer user saves the html source of a web page that submits data and is able to modify the web page to send data that is not normally allowed, that would be an example of cross-sight scripting (the user's own web page being one site, submitting data to the web server being another site.) For example, you might have a web page that submits a ZIP code for an address and only allows numeric data. I might maliciously modify the web page so my copy submits a web link (URL) in the ZIP code field. The only real prevention for cross-site scripting is to assume it will always happen and take steps on the server to catch and handle it.

Perhaps your web page submits data to Oracle database, and a malicious user modifies a copy of your web page to submit Oracle SQL or PL/SQL commands in the zip code field. That hacker might put this code in the field, "`11111;delete from employees;--`". If you are building a dynamic query that simply embeds the submitted data in a query, then instead of executing:

```
UPDATE EMPLOYEES SET ZIP=11111 WHERE EMPLOYEE ID=300;
```

You might execute this set of commands:

```
UPDATE EMPLOYEES SET ZIP=11111;delete from employees;-- WHERE EMPLOYEE ID=300;
```

That works out to three lines: an update for all employees, deleting all records from employees, and a comment. That is an example of SQL injection.

A typical SQL injection attack modifies a select statement by tacking on an extra where test that is true for all data. For example, if I accept user input of a last name to search employees and the user types "King' or 'a'='a", my dynamic SQL might look like this:

```
SELECT * FROM EMPLOYEES WHERE LAST NAME='King' or 'a'='a';
```

If this were a test for password matching a value stored in Oracle database, then the SQL injection might look like this:

```
SELECT count(*) FROM EMPLOYEES WHERE LAST NAME='King' and
    PASSWORD='whatever' or 'a'='a';
```

The select statement will return a number greater than 0 and the user might get access even though he doesn't know the password.

At the Oracle database, you can prevent SQL injection from happening by several methods. One traditional way is to filter the incoming data and/or escape the data (make it a sequence of individual

characters rather than text.) However, a better way is to always use parameterized input. We do that with our stored procedures that take parameters. We are not building dynamic queries but are pumping parameters into PL/SQL that has already been staged in Oracle database. The database *binds* the variables to our query/update framework.

We can also prevent SQL injection from the client with prepared statements in Java like the following. The value of `userInputEmpID` is bound to the query parameter at the question mark (?).

```
String query = "SELECT * FROM EMPLOYEES WHERE EMPLOYEE ID = ? ";
PreparedStatement pstmt = connection.prepareStatement( query );
pstmt.setString( 1, userInputEmpID );
```

Having a `PreparedStatement` accept our parameter and populate the query with it, keeps malicious code from being tacked onto the query. Once again, the `PreparedStatement` is staged in Oracle database and our parameters are set there, Oracle database binds them to the update/query.

If you need to place code for an Oracle database query in your application (java or other), use a `PreparedStatement`, as seen previously, rather than concatenating user input into a query string.

Demonstrating Failure to SQL Inject in Stored Procedure

I have included two more procedures in the `hr sec pkg` that will demonstrate attempts at SQL injection in stored procedures. I don't hail from Missouri, but I am from the "Show-Me State" of mind: trust, but verify. Let's try SQL injection when doing a select query on `LAST NAME`. In the procedure partially shown in Listing 7-15, `p select employee by ln sens`, we will pass in a tenth parameter, the `LAST NAME` and modify our select in the procedure to use it:

Listing 7-15. *Select Employees Data by Last Name and Attempt SQL Injection*

```
PROCEDURE p select employee by ln sens(
    ...
    m_last_name    employees.last name%TYPE )
IS BEGIN
    ...
    OPEN resultset out FOR SELECT
        ...
    FROM employees
    WHERE last_name = m_last_name;
```

Let's also see if we can sneak some SQL injection in by embodying it in a `RAW` (like we do with our encrypted data updates) and by casting the `RAW` to a `VARCHAR2` in the `WHERE` clause. We do that in a test procedure named `p select employee by raw sens` in Listing 7-16.

Listing 7-16. *Select Employees Data by RAW Value and Attempt SQL Injection*

```
PROCEDURE p select employee by raw sens(
    ...
    m last name    RAW )
IS BEGIN
    ...
    OPEN resultset out FOR SELECT
        ...
    FROM employees
    WHERE last_name = UTL_RAW.CAST_TO_VARCHAR2( m_last_name );
```

You will be happy to note that when we test this, these attempts at subterfuge are not successful. In both these cases, as is intended, Oracle database says, "give me something to plug into the test `WHERE LAST NAME = ?;`".

We will be giving Oracle database this string: "King' or 'a'='a", which one might imagine being wrapped in single quotes to become:

```
WHERE LAST NAME = 'King' or 'a'='a';
```

However, Oracle database sees our string as a *single data element* and checks to see if anyone's `LAST NAME` is (in escaped form): "King\' or \'a\'=\'a" or "{King' or 'a'='a}".

Executing the HR Package Specification and Body

Now that we have described the procedures in the `hr sec pkg` package, we will go ahead and execute the `CREATE` statements for the package specification and the package body. Execute the two blocks in the file named *Chapter7/HR.sql* to create the `hr sec pkg` package specification and body. After you have created `hr sec pkg`, you will need to grant execute on the package to the `hrview role` role

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

Inserting an EMPLOYEES Record: Update a Sequence

We need a fixed `EMPLOYEE ID`, number 300 for a record in the `EMPLOYEES` table in order for our example code to work. When the sample `EMPLOYEES` table is initially installed, there are about 100 records, with `EMPLOYEE IDs` from 100 to about 200. Generally, inserts into the `EMPLOYEES` table use the next value of the *sequence*, `EMPLOYEES SEQ` like this (do not execute this yet—this is for reference only):

```
INSERT INTO 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, 'David', 'Coffin', 'DAVID.COFFIN',
    '800.555.1212', SYSDATE, 'SA REP', 5000, 0.20, 147, 80);
```

Every time that `SEQUENCE.NEXTVAL` is called, the value is incremented. To see the current (next) value of `EMPLOYEES SEQ`, execute this command:

```
SELECT last number FROM user sequences WHERE sequence name='EMPLOYEES SEQ';
```

■ **Note** You can find a script of the commands in this section in the file named *Chapter7/HR.sql*.

There is no sanctioned way to manually set the `LAST NUMBER` for a sequence. However, we can adjust the increment value to get the desired effect. First, assure that the current `LAST NUMBER` returned in the command above is less than 300 (our example `EMPLOYEE ID`.) If it's not, you may have to substitute a number larger than the `LAST NUMBER` in our example code or update the data at `EMPLOYEE ID` 300.

To get set up to insert our example `EMPLOYEES` record at `EMPLOYEE ID` = 300, we need to get the `LAST NUMBER` of `EMPLOYEES SEQ` to be equal to 300 We are going to do that with an anonymous (unnamed)

PL/SQL block. This doesn't get saved to a named stored procedure but gets executed once to accomplish our plan.

■ **Note** We use `LAST_NUMBER` from the `user_sequences` view instead of the current value, `CURRVAL` of the sequence. We do this because we may not have a `CURRVAL` in this session. `CURRVAL` only exists after we have executed `NEXTVAL` on the sequence in this session. Then we can get the current value of the sequence.

See Listing 7-17. We have a `NUMBER`, `offset` into which we select the value (300 – `LAST_NUMBER`) from our sequence. If our `LAST_NUMBER` is currently 207 for example, the value of `offset` will be 300 – 207 or 93. We concatenate a command string, `alter_command` to `ALTER` the sequence to set the `INCREMENT BY` value to that `offset`. We pass that `ALTER` command to `EXECUTE IMMEDIATE`. Then the next time we call `EMPLOYEES_SEQ.NEXTVAL`, we will get the value of 207 + 93 = 300. To finish this plan, we set the `INCREMENT BY` value for the sequence back to 1.

Execute all of the commands in Listing 7-17 at this time. You will create our test user as `employee_id` = 300. Feel free to insert your own personal data in the `INSERT` command at the end.

Listing 7-17. *Anonymous PL/SQL Block to Reset Sequence*

```
DECLARE
    offset NUMBER;
    alter_command VARCHAR2(100);
    new_last_number NUMBER;
BEGIN
    SELECT (300 - last_number) INTO offset FROM user_sequences
        WHERE sequence_name='EMPLOYEES_SEQ';

    alter_command := 'ALTER SEQUENCE employees_seq INCREMENT BY ' ||
        TO_CHAR(offset) || ' MINVALUE 0';
    EXECUTE IMMEDIATE alter_command;

    SELECT employees_seq.NEXTVAL INTO new_last_number FROM DUAL;
    DBMS_OUTPUT.PUT_LINE( new_last_number );

    EXECUTE IMMEDIATE 'ALTER SEQUENCE employees_seq INCREMENT BY 1';
END;
/

SELECT last_number FROM user_sequences WHERE sequence_name='EMPLOYEES_SEQ';

INSERT INTO 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, 'David', 'Coffin', 'DAVID.COFFIN',
    '800.555.1212', SYSDATE, 'SA_REP', 5000, 0.20, 147, 80);
```

```
COMMIT;
```

A brute-force way to increment a sequence without inserting records is to `SELECT SEQUENCE.NEXTVAL` a sufficient number of times. You can also set the `INCREMENT BY` value to a negative number to reduce the value of `LAST NUMBER` in a sequence.

We can see the new `LAST NUMBER` setting in `EMPLOYEES SEQ` by selecting it again. Assure that it is 300, and then insert our example record for `EMPLOYEE ID` = 300. Finally `COMMIT` the updates. See our new entry by selecting:

```
SELECT * FROM employees WHERE employee id=300;
```

I should mention that there is an existing `INSERT/UPDATE/DELETE` trigger on the `EMPLOYEES` table named `SECURE DML`. This trigger limits changing the `EMPLOYEES` data to weekdays from 8 AM to 6 PM. It is similar to the restrictions we enforced in Chapter 2. However, by default this trigger is disabled.

Demonstrations and Tests of Encrypted Data Exchange

We are going to execute a separate Java class, `TestOracleJavaSecure` to emulate a client application of the `HR` schema. Our client application will call the stored procedures that we defined within the `hr sec pkg`, doing several queries and a few updates.

We will only explore several small and large snippets of the application code. You should have the *`TestOracleJavaSecure.java`* `file open to refer to while we go through this section.`

■ **Note** You can find this code in the file *Chapter7/TestOracleJavaSecure.java*.

Some Preliminary Steps

Before we get into the big individual demonstrations and tests, we want to get our bearings. The next few subsections set us up for the demonstrations and tests that follow.

The main() Method and Method Members

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.

The first thing we do in the `main()` method is establish an Oracle connection, shown in Listing 7-18. Edit the connection string to use the password you assigned to the Application User, `appusr`, and with your specific server name and port.

Listing 7-18. *Beginning of Code to Test Encryption in Transit,* `TestOracleJavaSecure` *Class*

```
public class TestOracleJavaSecure {
    public static void main( String[] args ) {
        Connection conn = null;
        try {
            private static String appusrConnString =
                "jdbc:oracle:thin:AppUsr/password@localhost:1521:Orcl";
```

```
Class.forName( "oracle.jdbc.driver.OracleDriver" );
conn = DriverManager.getConnection( appusrConnString );
```

Getting Ready for Encryption

We do not need the `Connection` in `OracleJavaSecure` because we won't be calling Oracle database directly from that class. The only function of `OracleJavaSecure` on the client (in this chapter) is to build keys and encrypt/decrypt data. See Listing 7-19.

Listing 7-19. *Get Ready for Encryption*

```
//OracleJavaSecure.setConnection( conn );
String locModulus = OracleJavaSecure.getLocRSAPubMod();
String locExponent = OracleJavaSecure.getLocRSAPubExp();
```

We get our RSA key pair, and get the public key exponent and modulus to pass to Oracle database.

Setting Non-Default Role

The `appusr` user has the privilege to execute the `appsec.p check hrview access` procedure (refer back to Chapter 2), which will set the Secure Application Role, `hrview role`. We execute the procedure as shown in Listing 7-20.

Listing 7-20. *Set Non-Default Role*

```
stmt = ( OracleCallableStatement )conn.prepareCall(
    "CALL appsec.p_check_hrview_access()" );
// Comment next line to see Exception when non-default role not set
stmt.executeUpdate();
```

We need to execute the statement in order to acquire the role. If you want to assure yourself that access without the role will fail, comment the line to `executeUpdate()` and run `TestOracleJavaSecure`. Be sure to uncomment that line after you run that test, so you can run our primary tests.

Reusing a Callable Statement

Inasmuch as an `OracleCallableStatement` is an interface that implements `Statement`, we can use it like a regular `Statement`. A regular `Statement` can be used over and over again to execute queries and updates. In my experience, though, if you have `OUT` parameters from a procedure called from your `OracleCallableStatement`, then you should not reuse it—just get a new `OracleCallableStatement`.

That first call to set our role in Listing 7-20, leaves us with an `OracleCallableStatement` that we can reuse for getting a count of the rows in our non-sensitive view of `EMPLOYEES`. We will count the rows two ways, both shown in Listing 7-21: once by iterating through the `ResultSet` of all the rows, incrementing our count, `cnt` as we go; and once by selecting the `count(*)` of all rows. Selecting `count(*)` is a much more efficient way:

Listing 7-21. *Get a Count of Rows in the Public View of Employees*

```
rset = stmt.executeQuery(
    "SELECT * FROM hr.v employees public" );
int cnt = 0;
while( rset.next() ) cnt++;
System.out.println( "Count data in V EMPLOYEES PUBLIC: " + cnt );

rset = stmt.executeQuery(
    "SELECT COUNT(*) FROM hr.v employees public" );
if( rset.next() ) cnt = rset.getInt(1);
System.out.println( "Count data in V EMPLOYEES PUBLIC: " + cnt );

if( null != stmt ) stmt.close();
```

Selecting Encrypted Data from EMPLOYEES

Here is our textbook process for selecting encrypted data from Oracle database, Listing 7-22. It resembles the test procedures we used in the last chapter. We pass the public key modulus and exponent to Oracle database and receive back the encrypted artifacts of our DES secret password key. In addition, we have an `OUT` parameter that is of type `OracleTypes.CURSOR`. We will draw on the `OracleTypes.CURSOR` (`ResultSet` in Java) to read our data.

Listing 7-22. *Java Code to Select Sensitive Data from Employees, from `p select employees sensitive`*

```
stmt = ( OracleCallableStatement )conn.prepareCall(
    "CALL hr.hr sec pkg.p_select_employees_sensitive(?,?,?,?,?,?,?,?,?)" );
stmt.registerOutParameter( 3, OracleTypes.RAW );
stmt.registerOutParameter( 4, OracleTypes.RAW );
stmt.registerOutParameter( 5, OracleTypes.RAW );
stmt.registerOutParameter( 6, OracleTypes.RAW );
stmt.registerOutParameter( 7, OracleTypes.CURSOR );
stmt.registerOutParameter( 8, OracleTypes.NUMBER );
stmt.registerOutParameter( 9, OracleTypes.VARCHAR );
stmt.setString( 1, locModulus );
stmt.setString( 2, locExponent );
stmt.setNull(   3, OracleTypes.RAW );
stmt.setNull(   4, OracleTypes.RAW );
stmt.setNull(   5, OracleTypes.RAW );
stmt.setNull(   6, OracleTypes.RAW );
// This must go without saying - unsupported type for setNull
//stmt.setNull( 7, OracleTypes.CURSOR );
stmt.setInt(    8, 0 );
stmt.setNull(   9, OracleTypes.VARCHAR );
stmt.executeUpdate();

errNo = stmt.getInt( 8 );
if( errNo != 0 ) {
    errMsg = stmt.getString( 9 );
    System.out.println( "Oracle error 1) " + errNo +
```

```
        ", " + errMsg );
} else {
    System.out.println( "Oracle success 1)" );
    sessionSecretDESSalt = stmt.getRAW( 3 );
    sessionSecretDESIterationCount = stmt.getRAW( 4 );
    sessionSecretDESAlgorithm = stmt.getRAW( 5 );
    sessionSecretDESPassPhrase = stmt.getRAW( 6 );
    rs = ( OracleResultSet )stmt.getCursor( 7 );
    //while( rs.next() ) {
    // Only show first row
    if( rs.next() ) {
        System.out.print( rs.getString( 1 ) );
        System.out.print( ", " );
        System.out.print( rs.getString( 2 ) );
        System.out.print( ", " );
        System.out.print( rs.getString( 3 ) );
        System.out.print( ", " );
        System.out.print( rs.getString( 4 ) );
        System.out.print( ", " );
        System.out.print( rs.getString( 5 ) );
        System.out.print( ", " );
        System.out.print( rs.getString( 6 ) );
        System.out.print( ", " );
        System.out.print( rs.getString( 7 ) );
        System.out.print( ", " );
        System.out.print( OracleJavaSecure.getDecryptData(
            rs.getRAW( 8 ), sessionSecretDESPassPhrase,
            sessionSecretDESAlgorithm, sessionSecretDESSalt,
            sessionSecretDESIterationCount ) );
        if ( null != rs.getRAW( 8 ) )
            System.out.print( " (" + rs.getRAW( 8 ).stringValue() +
                    ")" );
        System.out.print( ", " );
        // Most initial commissions in database are null
        System.out.print( OracleJavaSecure.getDecryptData(
            rs.getRAW( 9 ), sessionSecretDESPassPhrase,
            sessionSecretDESAlgorithm, sessionSecretDESSalt,
            sessionSecretDESIterationCount ) );
        if ( null != rs.getRAW( 9 ) )
            System.out.print( " (" + rs.getRAW( 9 ).stringValue() +
                    ")" );
        System.out.print( ", " );
        System.out.print( rs.getString( 10 ) );
        System.out.print( ", " );
        System.out.print( rs.getString( 11 ) );
        System.out.print( "\n" );
    }
}
if( null != rs ) rs.close();
if( null != stmt ) stmt.close();
```

Our error handling is identical to what we did in the last chapter: we hand back the error number and message through two of the OUT parameters. If there is no error, we continue by getting the secret password key artifacts into local method members, and we also get the CURSOR into a ResultSet so we can walk through the data. Since there are about 100 entries in the EMPLOYEES table, we will only show the first one (by reading the ResultSet.next() in an if block rather than in a while block).

Most of the columns are clear text, so we just print them out. But the SALARY and COMMISSION_PCT values are handed to us by Oracle database as RAW encrypted data. (Note that the encrypted SALARY is in the 8th position in the ResultSet, and the ResultSet is in the seventh position in the Statement. Those element numberings are independent.) We will send those RAW values to the getDecryptData() method, along with the secret password key artifacts. If we haven't yet built our local copy of the secret password key, we will build it there; in any case, we will decrypt the data and return it as a String. We will also print that out.

For our demonstration purposes only, we are also going to print out the actual RAW (if not null), within parentheses, "()". If we want to track that across multiple runs through this code, we will see that it is different each time—owing to using different secret password keys in different Oracle sessions.

At the end of each call to our stored procedures for data encryption, we will close the OracleCallableStatement. We may also close the ResultSet, even though it is unnecessary—it is inherently closed when we close the Statement, because it was acquired through the Statement. However, it is good practice to explicitly close each ResultSet, especially because it is common practice to reuse a Statement—you may open multiple ResultSets during the life of a single Statement, and to assure that you are freeing up Oracle resources, you should close each ResultSet when you are done with it.

Selecting All Columns in Encrypted String

Our second example procedure for selecting encrypted data from Oracle database, p select employees secret does not select and encrypt individual columns, but rather selects all the columns and concatenates them into a single, comma-delimited VARCHAR2 to be encrypted. In this case, none of the data is sent in clear-text form. At the client, if individual columns of the data are needed, you will need to parse the decrypted string to acquire individual data elements.

Also in this example, we are using a comma as a delimiter between fields. This assumes that there are no commas in the data—not often a valid assumption. You can use a different delimiter that is less likely to occur in the data, like a caret (^)or tilde (~). This is an example only, and you would need to evaluate your specific requirements for the data before building a procedure like this for your application.

We call the procedure, decrypt the data and print the results as in partial Listing 7-23.

Listing 7-23. *Encrypt All Data Selected from Employees, from* `p select employees secret`

```
stmt = ( OracleCallableStatement )conn.prepareCall(
    "CALL hr.hr_sec_pkg.p_select_employees_secret(?,?,?,?,?,?,?,?,?)" );
    ...
    if( rs.next() ) {
        System.out.print( OracleJavaSecure.getDecryptData( rs.getRAW( 1 ),
            sessionSecretDESPassPhrase,
            sessionSecretDESAlgorithm, sessionSecretDESSalt,
            sessionSecretDESIterationCount ) );
        if( null != rs.getRAW( 1 ) )
            System.out.print( " (" + rs.getRAW( 1 ).stringValue() +
                ")" );
```

```
        System.out.print( "\n" );
```

Again, we print the String value of the encrypted RAW in parentheses beside the data. In this case there is a single RAW that represents the entire row of concatenated data.

Sending Encrypted Data to Oracle Database for Insert/Update

I wouldn't say that this next example procedure bears no resemblance to our previous examples, but you will notice there are none of the trappings of key exchange in either direction. This is an update, and in order to encrypt the data at the client, we would already have had to exchange keys. Listing 7-24 shows the code to call our encrypted data update procedure, `p update employees sensitive`.

Listing 7-24. Update Sensitive Data in Employees, call `p update employees sensitive`

```
stmt = ( OracleCallableStatement )conn.prepareCall(
    "CALL hr.hr sec pkg.p_update_employees_sensitive(?,?,?,?,?,?,?,?,?,?,?,?,?,?)" );
stmt.registerOutParameter( 12, OracleTypes.NUMBER );
stmt.registerOutParameter( 13, OracleTypes.VARCHAR );
stmt.setInt(    1, 300 );
stmt.setString( 2, "David" );
stmt.setString( 3, "Coffin" );
stmt.setString( 4, "DAVID.COFFIN" );
stmt.setString( 5, "800.555.1212" );
stmt.setDate(   6, new Date( ( new java.util.Date() ).getTime() ) );
stmt.setString( 7, "SA REP" );
// Note - may not have locModulus, locExponent,  at this time!
stmt.setRAW(    8, OracleJavaSecure.getCryptData( "9000.25" ) );
stmt.setRAW(    9, OracleJavaSecure.getCryptData( "0.15" ) );
stmt.setInt(   10, 147 );
stmt.setInt(   11, 80 );
stmt.setInt(   12, 0 );
stmt.setNull(  13, OracleTypes.VARCHAR );
stmt.executeUpdate();

errNo = stmt.getInt( 12 );
if( errNo != 0 ) {
    errMsg = stmt.getString( 13 );
    System.out.println( "Oracle error 3) " + errNo + ", " + errMsg );
}
else System.out.println( "Oracle success 3)" );
if( null != stmt ) stmt.close();
```

For our encrypted data, we first call the `getCryptData()` method. Then we set a RAW parameter in our Oracle procedure. When we execute this statement, both the clear-text and encrypted data values get sent to Oracle database for Insert or Update, as determined by the procedure.

A TALE OF TWO DATE CLASSES

As in our previous examples, here in Listing 7-24 we are setting our parameters, except that this time, we are setting data instead of key artifacts. In our sixth parameter, we are instantiating a `java.util.Date`

class. The constructor with no arguments creates a `Date` with the current date and time. Frequently with Oracle database, we will be using instances of `java.sql.Date` (notice the package *java.sql* instead of *java.util*), which does not have such a constructor; however, we can construct a `java.sql.Date` with the long (milliseconds) coming from the `java.util.Date.getTime()` method. We could also use the same statement used by `java.util.Date()` constructor to instantiate a `java.sql.Date` object, like this:

```
java.sql.Date( System.currentTimeMillis() );
```

I have one observation about the syntax we use in our code. The parentheses around our instantiation of a `( new java.util.Date() )` allow us immediate access to it as an object, to its `getTime()` method, like this:

```
stmt.setDate(  6, new Date( ( new java.util.Date() ).getTime()));
```

Okay, so why doesn't `java.sql.Date` have a `Date()` constructor, didn't it inherit one by extending `java.util.Date`? Oh, excellent question! The short answer is that constructors are not considered members of a class, so they are not inherited by the subclass. You can, however, get access to the parent constructors by a call to the `super()` method as the first line in any subclass constructor. There can be multiple `super()` methods with differing signatures, one for each constructor in the parent class.

We will be coming back to the subject of *java.sql* and *java.util* `Dates` again in Chapter 9. We need a standard practice for exchanging dates, and I will propose one there.

Selecting a Single Row from EMPLOYEES

Perhaps we don't want to select the entire table, but only records that meet a certain criteria. We can provide criteria to be made part of the `WHERE` clause in our query. In this procedure, `p select employee by id sens`, Listing 7-25, we are providing a tenth parameter, the `EMPLOYEE ID`.

Listing 7-25. *Select by ID, Sensitive Data from Employees, from* `p select employee by id sens`

```
stmt = ( OracleCallableStatement )conn.prepareCall(
    "CALL hr.hr sec pkg.p select employee by id sens(?,?,?,?,?,?,?,?,?,?)" );
...
stmt.setInt(   10, 300 ); // Employee ID 300
stmt.executeUpdate();
```

The remainder of the procedure call is identical to our previous query examples, except that we only expect one or zero records to be returned (the `EMPLOYEE ID` is a `UNIQUE` key for this table.) So we can handle the `ResultSet` in an `if( rs.next() )` block instead of `while( rs.next() )`.

Selecting EMPLOYEES Data by Last Name: Try SQL Injection

We can also query all records meeting some other criteria that is perhaps not `UNIQUE`. One example that we will show queries `EMPLOYEES` by `LAST NAME` using the `p select employee by ln sens` procedure, in partial Listing 7-26. There are two entries with the `LAST NAME` "King". So this would return two rows.

Listing 7-26. *Select by Last Name, Sensitive Data from Employees, from* `p select employee by ln sens`

```
stmt = ( OracleCallableStatement )conn.prepareCall(
```

```
    "CALL hr.hr sec pkg.p select employee by ln sens(?,?,?,?,?,?,?,?,?,?)" );
...
stmt.setString(   10, "King" ); // Employees Janette and Steven King
...
while( rs.next() ) {
```

Iterate through the `ResultSet` using a `while` block to see all returned rows.

We can attempt a SQL injection here by replacing our previous parameter 10 setting with the following:

```
stmt.setString(   10, "King' or 'a'='a" );
```

What we will see is that *no data is returned,* because no `EMPLOYEES` have the `LAST NAME` {King' or 'a'='a}.

Selecting EMPLOYEES Data by RAW: Try SQL Injection

Perhaps, one might think, we can break through and accomplish SQL injection if we hand our data back as a `RAW` and only convert it to a `VARCHAR2` at the point of making our selection (refer to the Oracle procedure, `p select employee by raw sens` which we described earlier and in Listing 7-16.) This procedure call, partial Listing 7-27, attempts that strategy.

Listing 7-27. *Select by RAW Value, Sensitive Data from Employees, from* `p select employee by raw sens`

```
stmt = ( OracleCallableStatement )conn.prepareCall(
    "CALL hr.hr sec pkg.p select employee by raw sens(?,?,?,?,?,?,?,?,?,?)" );
...
stmt.setRAW(   10, new RAW("King' or 'a'='a".getBytes()) );
```

You will see that once again, our attempt at SQL injection in a stored procedure fails. It seems that passing parameters, as opposed to embedding user-provided text in dynamic SQL, is quite resistant to SQL injection.

Also note how we are treating the value between quotation marks as if it were already a `String` object, calling the `getBytes()` method. We saw that first in Chapter 6.

Testing Encryption Failure with New Client Keys

Perhaps you need to see this for yourself, or perhaps not. In any case, you can test calling procedures with keys on the client that don't match what's on Oracle database: it will fail. Take note that in the Java code thus far in the `TestOracleJavaSecure.main()` method, we have exchanged keys, and those would continue to work. We will take a split-second to remove or disable the keys on the client. We do that with a call to the `resetKeys()` method (see Listing 7-28).

Listing 7-28. *Test Encryption with Mixed Keys*

```
OracleJavaSecure.resetKeys(); // Method for Chapter 7 testing only
locModulus = OracleJavaSecure.getLocRSAPubMod();
locExponent = OracleJavaSecure.getLocRSAPubExp();
```

We also establish new keys on the client by calling the `getLocRSAPubMod()` method, and we get the public modulus and exponent. If we make no effort to pass these public key artifacts to Oracle database

and retrieve new secret password key artifacts in return, then an update (which doesn't exchange keys) will fail. This is what we will test.

Note that it is unlikely that we will accidentally reset our keys, and calling the `getLocRSAPubMod()` method or other methods of `OracleJavaSecure` multiple times will not create new keys if they already exist, but rather will return the existing keys. The keys are `static`, so they will not disappear on their own as long as the Java Virtual Machine continues to run. So, this test is really just for fun: answering a "what if" question.

Testing Failure with New Oracle Connection

Another test that we can demonstrate is the scenario wherein we close the `Connection` to Oracle database and establish a new one. The server-side keys will disappear—both the client public key and the secret password key. We can reset the `Connection` and prepare to call our encryption procedures by setting our Secure Application Role to `HRVIEW ROLE` as in Listing 7-29.

Listing 7-29. *Reset Connection and Re-Set Secure Application Role*

```
if ( null != conn ) conn.close();
conn = DriverManager.getConnection( appusrConnString );
OracleJavaSecure.setConnection( conn );
stmt = ( OracleCallableStatement )conn.prepareCall(
    "CALL appsec.p check hrview access()" );
stmt.executeUpdate();
```

Again, in the case of doing an encrypted Oracle update, we will see a failure because Oracle database has neither the public key nor the secret password key. This code is initially commented.

Some Closing Remarks

I've covered all the big demonstrations and tests in the preceding sections. Now, it's your turn to modify the tests and try some of your own. Feel free to comment and uncomment those sections of test code that we just described. When you recompile and run those sections of `TestOracleJavaSecure` you will see a message on an attempted data update that indicates we have "Failed where expected—OK."

If You Need To Close/Open Your Connection

Resetting your Oracle connection poses another potential problem when we do a select of encrypted data, for certainly we will pass our public key to Oracle database, and the database will build the secret password key and hand it back. However, we still retain an old copy of the secret password key on the client and will run into problems decrypting the data. We can remedy that after getting and decrypting the secret password key artifacts on the client with an immediate call to the `OracleJavaSecure.makeDESKey()` method, before we try to decrypt the data.

I think the best rule of thumb, and an examination on your code review checklist, is to retain the Oracle connection for the duration of your queries and updates. If there are long pauses and you need to close the Oracle connection, when you open a new connection, pass your public key artifacts to Oracle database via the `p get shared passphrase` procedure (described in the following) and build a replacement secret password key on the client by calling the `makeDESKey()` method.

Running Basic Key Exchange Without Data Encryption

For those times when all we want to do is submit encrypted data updates to Oracle database, or we want to be prepared to update before doing any selects, we need to assure that we have exchanged keys beforehand. We can do that by calling the `p get shared passphrase` procedure (in our current design, this procedure must be included in every individual application package, like `hr sec pkg`). Listing 7-30 presents the fundamentals of basic key exchange from a Java client.

Listing 7-30. *Basic Key Exchange*

```
    stmt = ( OracleCallableStatement )conn.prepareCall(
        "CALL hr.hr sec pkg.p get shared passphrase(?,?,?,?,?,?,?,?)" );
...
    OracleJavaSecure.makeDESKey( sessionSecretDESPassPhrase,
        sessionSecretDESAlgorithm, sessionSecretDESSalt,
        sessionSecretDESIterationCount );
```

We will get and decrypt each artifact of our secret password key, and pass that to the `makeDESKey()` method. At that point, we have completed our key exchange and are ready to exchange encrypted data and do decryption on both the client and Oracle database.

After we call the `p get shared passphrase` procedure and call `OracleJavaSecure.makeDESKey()`, the `TestOracleJavaSecure` class will again try to do an encrypted data update, and will succeed.

Executing the Demonstrations and Tests

We are going to run our demonstrations and tests now. To do that, we will again edit our code, as needed, and then compile and run it. In a command prompt, change directories to *Chapter7*. Edit *TestOracleJavaSecure.java* if not already, placing the correct password for `appusr` and correct host and port number in the Oracle connection string, near the top.

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

Compile the code with these commands, or just the second which will automatically compile the first (be sure the first line of *OracleJavaSecure.java* is commented, the line to `CREATE` the Java structures in Oracle database).

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

Then run the code from that same directory with this command:

```
java TestOracleJavaSecure
```

Observing the Results

When you execute `TestOracleJavaSecure` (as distributed), all of the tests listed previously will run straight through from top to bottom. The results will look like this:

```
Count data in V EMPLOYEES PUBLIC: 108
Count data in V EMPLOYEES PUBLIC: 108
Oracle success 1)
```

```
198, Donald, OConnell, DOCONNEL, 650.507.9833, 2007-06-21 00:00:00, SH CLERK, 2600
(E27811A8C7C9D9F3), null, 124, 50
Oracle success 2)
198, Donald, OConnell, DOCONNEL, 650.507.9833, 21-JUN-07, SH CLERK, 2600, , 124, 50
(F7EA4E97B2F39E036AF6E880B2E5CA3EB78332BF8CE82B7585A4CBC7B340FEBDE4862830927
D118D27A1DDE3304478D9A463EBA9BC78E3188217884D5F5EA92F54A6EA2FB62598D1419F003295D
F1C076E48BC6D07058E3B)
Oracle success 3)
Oracle success 4)
300, David, Coffin, DAVID.COFFIN, 800.555.1212, 2010-08-30 00:00:00, SA REP, 9000.25, .15,
147, 80
Oracle success 5) No data on failed SQL Injection
Oracle success 6) No data on failed SQL Injection
Failed where expected - OK. Need key exchange.
Oracle success 8)
Oracle success 9)
```

Demonstrating Scenarios

Here is a list in relatively plain English of the scenarios we have demonstrated. There is a lot of code to accomplish all these different scenarios. The code for each scenario is very similar to some of the other scenarios, with modifications for the specific demonstration.

- We queried the `EMPLOYEES` table and got the `SALARY` and `COMMISSION PCT` columns back in encrypted form. For both of these, we print out the decrypted `String`, and in parentheses, the `stringValue()` of the encrypted `RAW` (unless null). We only show the first row of the `ResultSet`.
- We queried the table and got all the columns back in one concatenated `String`, in encrypted form. We print the decrypted data, and in parentheses, the `stringValue()` of the encrypted `RAW`. Again, we only show the first row.
- We do an insert or update to the `EMPLOYEES` table, inserting `EMPLOYEE ID = 300`. If it already exists, we do an update. At that point, the salary is 9000.25 (now I'm dreaming).
- We select a single row from `EMPLOYEES`, requesting data `WHERE EMPLOYEE ID = 300`.
- We attempt to query `EMPLOYEES` through our procedures with a sample SQL injection string. This fails, and no data is returned.
- We attempt again to query `EMPLOYEES` through our procedures with a sample SQL injection string, this time transmitted as a `RAW` and only converted when we do the `SELECT`. This also fails, and no data is returned.
- We can alternately compile `TestOracleJavaSecure` to test resetting the client keys or resetting the Oracle connection. After that, our attempt to send encrypted data to Oracle database for insert/update fails, as expected.
- We successfully call the `p get shared passphrase` procedure and run the `makeDESKey()` method to complete key exchange.

- Again, we send encrypted data to Oracle database for insert/update, and we succeed. Now the salary is 9700.75 (now my wife is dreaming).

Querying Employees to See Updates

You can see the status of the record we inserted / updated by connecting to Oracle database as the HR user and executing the following command:

```
SELECT * FROM employees WHERE employee id=300;
```

Packaging Template to Implement Encryption

I know that folks find the cost of security, like the cost of quality, to be unpalatable; but the potential cost of insecurity is inordinately higher. Our security structures are only valuable if put to use. We will add some value in the next few chapters, but as it stands now, getting folks to use our security structures may be like pulling teeth. This is not one of those "if you build it, they will come" moments.

We are going to lower the threshold of entrance into these security structures by providing application developers with templates that they can use to quickly implement the back end Oracle structures for their application, and the front end Java calls to those structures.

Template for Oracle Application Security Structures

The first file that we will give to developers provides the code for an Oracle security package that they will need to implement in their application schema. In the file, there are generic names for the application schema, the package, procedures, tables/views and columns. Developers should search and replace those names with the actual names they will use.

■ **Note** You can this file at *Chapter7/AppPkgTemplate.sql*.

Your security administrator, secadm, will need to grant execute on the appsec.app sec pkg to the developers' application schema user so the application can use the app sec pkg procedures and functions.

In order to create the application package, the developer will need the CREATE PROCEDURE system privilege. Also, in order for the application user(s) to use the application package, all application users will need the EXECUTE object privilege on the package granted to them.

We have four template procedures in the package:

```
p get shared passphrase
p select APPTABLE sensitive
p select APPTABLE by COLUMN1 sens
p update APPTABLE sensitive
```

There is nothing new being introduced to you here—you have seen all these structures used in the HR schema.

Template for Java Calls to Application Security

We will also provide our application developers with a template Java file. Each developer will search and replace the generic names for schema, table, procedure etc. to whatever names are appropriate in their application.

■ **Note** You can find the Java code for the `AppAccessSecure` template class in the file *Chapter7/AppJavaTemplate.java*.

This file bears strong resemblance to our test code for the structures in the `HR` schema, *TestOracleJavaSecure.java*. It consists of a `main()` method that establishes an Oracle connection and calls the application structures that the developer will define in the application schema.

Perhaps this will be the most intimidating code for your application developers because of the bulk required to do key exchange. You are the expert now, so you will do well to give your application developers some assistance. In fact, it will save you some headaches down the road if you can assist the developers in implementing this correctly, and avoiding any security weaknesses, like we did with the `HR.EMPLOYEES` table.

Java Archive for Use by Applications

In addition to providing your developers with the two template files, you need to give them the *orajavsec/OracleJavaSecure.class* file. I recommend that you do not give developers the *OracleJavaSecure.java* code file, just to assure yourself that no modified version of the class is in use in your organization.

Probably the best form to use to distribute the class file is in the form of a Java archive (JAR) file. To create an appropriate jar file for distribution, you can run the JAR Tool that comes with the JDK. If your `PATH` and `CLASSPATH` are still set as described in Chapter 2, you can get a command prompt window and change directory to the *Chapter7* directory. From there, execute this command:

```
jar cvf orajavsec.jar orajavsec/OracleJavaSecure.class
```

This will create a file named *orajavsec.jar* in the current directory. Distribute this file and instruct your application developers to place this file name in their `CLASSPATH` both during development and when their application code is run.

Don't Stop Now

With templates in hand, we are at a point where we can bring Oracle application developers into the encrypted data over the network fold, and you would do well to take one of your applications and configure it to use these structures and approaches, to blaze the way. But this is only a half-way house for security respite. We are going to introduce some powerful concepts in the following chapters that will entice your application developers to hang with the program. Stepping into secure application development and operation is like getting your life back. Security is a sense of well-being. It requires work, but with the work, you determine your own secure computing destiny.

Chapter Review

We took our key exchange and data encryption concepts that we learned in Chapter 6 and applied them to a specific Oracle application, the HR schema. Is this one of those great moments in science? Perhaps not, but it is a milestone, and a springboard for us into bigger and better things in regard to our application security. We have successfully encrypted our sensitive application data as it traversed the network.

We built structures for error logging, specifically to track errors and events involving our application security processes, no matter what client application is currently using them. Additionally, we built a trigger to automatically manage the logs, discarding records over 45 days old.

There was a lot of code, both PL/SQL and Java, presented in this chapter. The goal of this code is fairly straightforward—to use our existing data encryption processes when retrieving sensitive data and records from the HR schema, and when updating sensitive data and records.

Along the way, I introduced the following concepts:

- Using indexes for performance
- Triggers, autonomous transactions, and `COMMIT`
- Use and manipulation of sequences
- Default procedure parameters
- Use of the `CURSOR TYPE`
- Use of type specification by reference, anchored declaration of a datatype
- Use of `java.util.Date` and `java.sql.Date`

We spent significant time evaluating and testing our protections against SQL injection attacks.

Finally, we presented templates that can be delivered to application programmers so that they can implement these security constructs in their applications.

Please study Figures 7-1 and 7-2 for a visual overview of secure application data query and update processes. Figure 7-1 illustrates the process of querying Oracle database for sensitive data and retrieving it in encrypted form. At the top, you see a reference to Figure 5-1 [**A**] which is where we saw the RSA private/public key pair created on the client. Then in Figure 7-1, we see the client calls `p check hrview access` to set the role for access to the sensitive HR data.

At that point, we have the execute privilege to call the `hr sec pkg` package. We call `p select employees sensitive` to get both public and sensitive data from the `EMPLOYEES` table. The first order of business is to accomplish key exchange. You can see to the right of Figure 7-1 that we create an equivalent RSA public key on the Oracle database, and we create the secret password (DES) key which we will return to the client. We saw the details of that process in Figure 6-1 [**B**].

The next order of business of the procedure on Oracle database is to query the `EMPLOYEES` table and encrypt both the `SALARY` and `COMMISSION PCT` fields. Finally the procedure returns the data, both clear text fields and encrypted forms of the sensitive fields in a `ResultSet` to the client.

Back at the client (the left side of Figure 7-1), we build an equivalent secret password key based on the artifacts that the Oracle database returned. See the key image at the bottom of the illustration. Then we display the data from the `ResultSet`, using the secret password key to decrypt the sensitive data components.

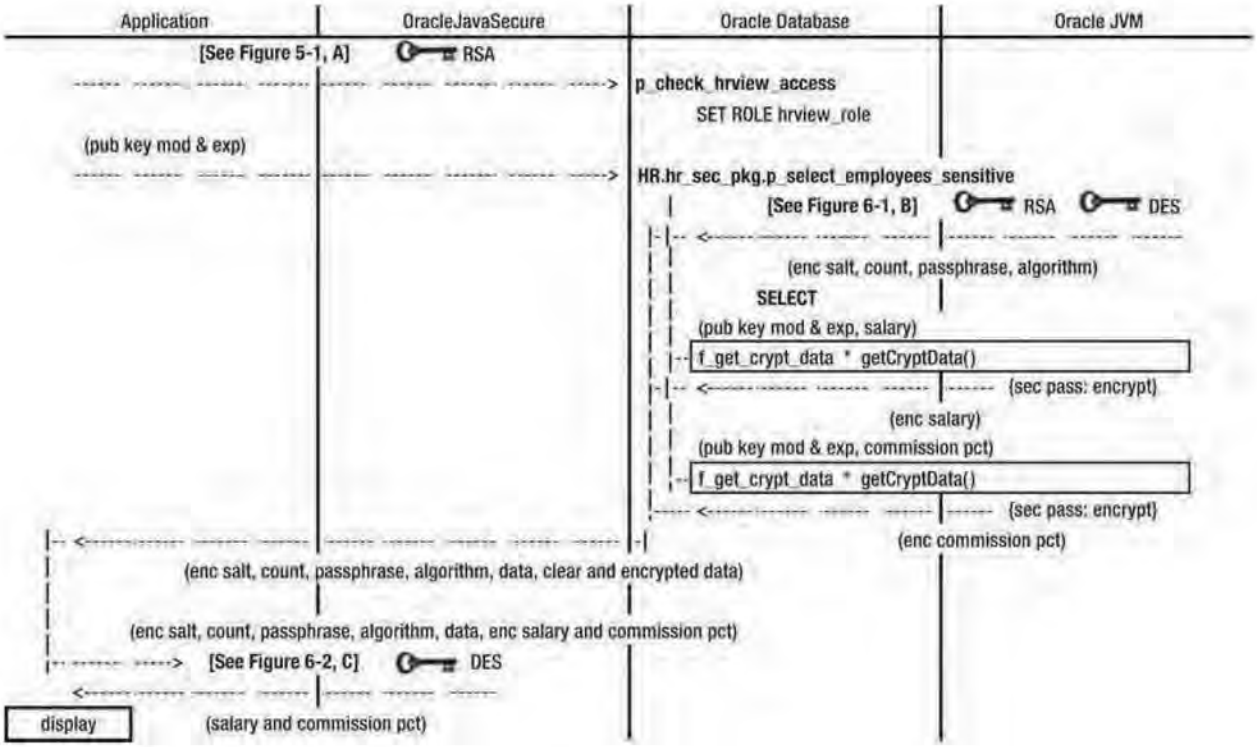


Figure 7-1. *Secure application data query*

Figure 7-2 illustrates the process for updating sensitive data. For that process we encrypt the sensitive data on the client, using the secret password key. We do that by calling the `getCryptData()` method. Then we submit both clear text fields and encrypted sensitive fields to Oracle database by calling the `p update employees sensitive` procedure.

The key exchange must have already taken place, so that the Oracle database can use the equivalent (original) secret password key to decrypt the data. Then the procedure executes an `INSERT` or `UPDATE` command to store the data.

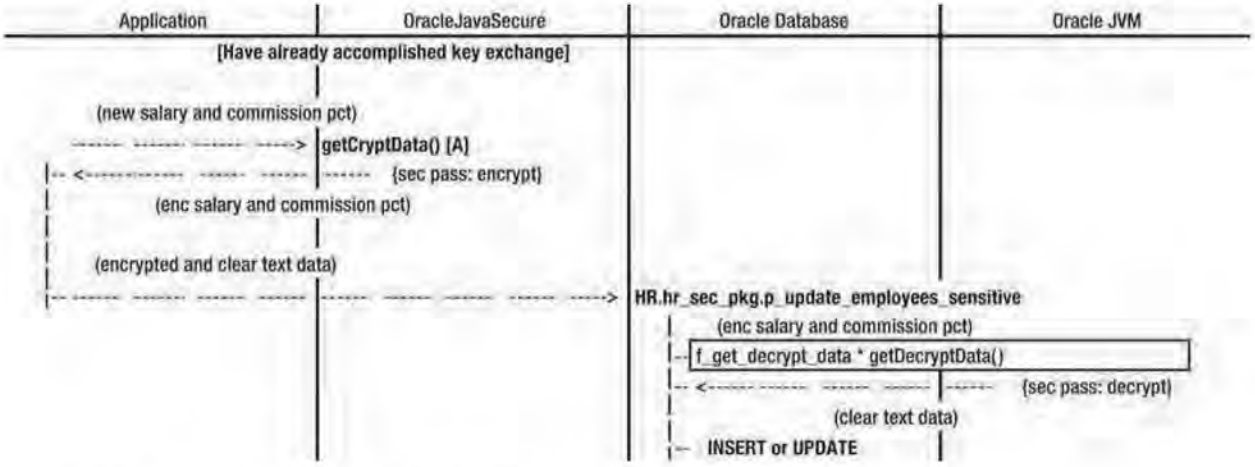


Figure 7-2. *Secure application data update*

CHAPTER 8

Single Sign-On

Single sign-on (SSO) is a relatively simple concept. Did you log in when you turned on your computer, before you were able to start working? If so, then we ought to be able to find out who you logged in as and assume that your identity is the same when you use our applications.

Part of the mystery here revolves around assurance and trust. Can I trust that initial log in to be secure enough so that only the actual user can authenticate as himself? Can I be sure that no one else has that username and password, and that no one can pretend to be that user (spoof the user)? This assurance comes not only from the power of the encryption and the strength of the password storage protection, but also from the computer security code of conduct: frequency of password change, password composition and strength rules, screen-lock rules, rules against password sharing and posting, education about malware and viruses, and social engineering. This assurance also comes from network access control (NAC). Do we assure that each client computer has the latest updates and anti-virus software? Do we assure that client computers are correctly configured with security options, like our password-protected screen lock? Do we protect mobile computers with hard disk encryption, firewall software, and virtual private network (VPN) when they communicate with our company over public networks? And do we use NAC to do all this *before* we allow those computers to connect to our corporate network?

If I am sure of all these things, and anything else that is required to combat a computer security threat, then I can trust the original log in to be good enough for any further identity requirements. There is a large foundation that we require before we can continue building our application security with SSO.

Another Layer of Authentication?

We could always add another authentication in our application—a chance for the user to reenter the username and password (possibly a different password), but beyond frustrating the user, have we improved on security? (In the next chapter, we will try to improve on security with 2-Factor Authentication.)

I'm not saying that extra authentications are a bad thing. Sometimes you do not have the trust and assurances I listed previously, so SSO is not valid. The problems come when you have ten or tens of passwords. At that point, two things happen: your authentication support systems get heavily taxed because of lost or forgotten passwords and innumerable password resets, and your organization's (and personal) security decreases. Can it be true that security decreases with increased quantity of passwords? Yes, because now you have placed users in the position where they have too many passwords, changing too often to remember—they must be written down.

Perhaps your users are savvy enough to keep a list of all the places they have passwords, and when they change them one place, they change them all places. I'm not sure you can count on that, even in a computing organization with less than 10 people. And there are always exceptions—passwords that change more frequently and passwords that have different composition rules.

Additionally, are your users able to distinguish the list of secure, corporate sites where the single work password can be synchronized from all other insecure or non-work-related sites that should not have access to their work password? Work passwords that grant access to sensitive information should never be used on external systems—who knows what security those external systems maintain?

In the end, we need to strive for SSO. First, we need to establish the foundation of trust and assurance with NAC. Then we should push for adoption of SSO everywhere we can achieve it. Our Java and Oracle applications are one of those places.

Who Is Logged-In on the Client?

If you are on a Microsoft Windows client or a UNIX client, the operating system (OS) retains knowledge of the identity you claimed when you authenticated or logged in. That identity is published in your environment settings for easy access from scripts, but the environment can be changed by those scripts so that, judging from your environment settings alone, you can spoof a different user.

To see this spoofing, bring up a command prompt window (on Windows) and type `SET` to see your settings. Perhaps you have a setting called `USERNAME` near the end. Observe what it is. In the same window, type:

```
set USERNAME=coffin
```

Now type `SET` again and observe what the value of `USERNAME` is. Note that this doesn't change your identity in the operating system, but it could change your identity for any script that uses the current environment as its source.

Because the environment is so malleable and transient, it is not a trustworthy source of identity. Do not read a user's identity from the environment—if you are doing that, change your ways!

Find a Better Source of OS User Identity

Windows is willing to tell our applications who the logged-in user is. Specifically, the Active Directory authenticated user ID is available to all who know how to read it. The JRE for Windows has a class that can acquire the user ID. That class is named `NTSystem`, and it is in the *com.sun.security.auth.module* package—it is automatically available to the Java applications on Windows.

There is an alphabet soup of acronyms that will help us define what `NTSystem` does. `NTSystem` calls functions in a dynamic load library (DLL) named *jaas_nt.dll*—part of the Java authentication and authorization system (JAAS) package. That DLL is included in the JRE (and JDK) for Windows. `NTSystem` makes native code calls using Java Native Interface (JNI) to the DLL. I suspect that `NTSystem` was named for the precursor to Active Directory, NT Directory Service. For Microsoft, the Windows NT (marketed in 1993 as New Technology) server operating system was a source of many names, including NT Domains.

Use NTSystem or UnixSystem to Get Identity

When you use `NTSystem`, you are not relying on the environment or other intermediary, but are going directly to the OS to get the user identity. Listing 8-1 shows the code for that functionality.

Listing 8-1. *Use NTSystem to Get Windows User Identity*

```
import com.sun.security.auth.module.NTSystem;

NTSystem mNTS = new NTSystem();
String name = mNTS.getName();
```

Now, wasn't that easy?

If you have UNIX clients, you can use a similar JAAS component through the `UnixSystem` class, as in Listing 8-2.

Listing 8-2. *Use UnixSystem to Get UNIX User Identity*

```
import com.sun.security.auth.module.UnixSystem;

UnixSystem mUX = new UnixSystem();
String name = mUX.getUsername();
```

These classes are not both available on either client, and you can only compile the code shown above on the system where the class exists. So you should only include one or the other calls to `NTSystem` or `UnixSystem` in your application code. You can include each of these and comment one to compile the other for distribution to those clients where appropriate.

There are other ways to use Java to read the current user identity from Windows Active Directory, for example as a lightweight directory access protocol (LDAP) service; however, security and configuration will be very specific to your corporate environment. I will not cover that approach here.

Do Cross-Platform-Specific Coding with Reflection

That approach of "pick your client platform" is not very satisfying, nor does it meet the Java goal of "run anywhere." But we have to deal with the fact that `NTSystem` and `UnixSystem` classes are not only platform-specific, but also not available cross-platform. There is an approach that can solve this problem: *reflection*. With reflection, we can forget those pesky either/or `import` statements and commenting the platform-inappropriate code.

With reflection we can write code that is compiled with possibilities rather than specifics. It is possible we will be running on a Windows platform, and in that case we want to use `NTSystem`. But it is also possible we are running on a UNIX platform, in which case we want `UnixSystem` to run.

With reflection, we will load a platform-appropriate class, sight-unseen (like going on a blind date), and then we will use the resources of that class. Reflection uses runtime type introspection to find and use properties of a specific class.

Listing 8-3 shows the code to get the user identity from the Windows OS using reflection. The code for UNIX is similar but uses the `UnixSystem` class rather than the `NTSystem` class.

■ **Note** You will find the code in Listing 8-3 in the file *Chapter8/PlatformReflectTest.java.*

Listing 8-3. *Use Reflection to Get OS User Identity*

```
//import com.sun.security.auth.module.NTSystem;
import java.lang.reflect.Method;

//NTSystem mNTS = new NTSystem();
Class mNTS = Class.forName( "com.sun.security.auth.module.NTSystem" );

//String name = mNTS.getName();
Method classMethod = mNTS.getMethod( "getName" );
String name = ( String )classMethod.invoke( mNTS.newInstance() );
```

Notice first in this code that we no longer import the Windows specific class, `NTSystem`. We would not be able to compile code with that `import` statement on a UNIX platform. Instead, we are importing the reflection class, `Method`. `Method` can represent any specific method in a class.

Next, notice that we do not instantiate an `NTSystem` class, as we had done before. Now we get an `NTSystem` class by using the `Class.forName()` method and giving the fully qualified name of `NTSystem`. Where have we seen this before? Oh that's right; we used this syntax when we load the `OracleDriver`. Using this syntax, the compiler has no trouble—it sees the fully qualified name as a `String`, so even if `NTSystem` doesn't exist on a UNIX box, you can compile this code there.

Next, we know we need to access a method named `getName()`, so we pass the name of the method to `Class.getMethod()`, which returns a `Method` class, `classMethod` representing the `getName()` method.

We do not yet have an instance of `NTSystem`, but we have a handle, `mNTS`, to what might amount to a `static` class. Our next step requires that we call the `getName()` method on an instance of `NTSystem`. To call the method, we call `classMethod.invoke()`, but we need a real instance (object) of `NTSystem`, so we instantiate the object by calling `Class.newInstance()`.

At this point, our instance of `NTSystem` returns the OS user name from the `getName()` method. However, because we are calling this through the `Method` class, we will get an `Object` type returned, which we need to cast as a `String`.

We will do more with reflection when we get to Chapter 10. There we use reflection to restore classes and objects from storage in the Oracle database and from transmission over the network. In the end, we will read their members and call their methods.

Assure More Stringent OS Identity

What would we like to know before we accept the `NTSystem` report of the user ID? We'd like to know first that we are on a Windows client. One possible pursuit of spoofing our code would be to run on a UNIX client with an imposter class named *com.sun.security.auth.module.NTSystem* found in the client `CLASSPATH`. There would be problems trying to accomplish that, but we will avoid the issue by simply assuring we are on a Windows machine. See Listing 8-4.

Knowing we are on a Windows client also informs us about which JAAS source to use: `NTSystem` instead of UnixSystem.

Listing 8-4. *Get OS User Identity, getOSUserID()*

```
    private static String expectedDomain = "ORGDOMAIN";

    //System.getProperties().list(System.out);
    if( ( System.getProperty("os.arch").equals("x86") ||
        System.getProperty("os.arch").endsWith("64")) &&
```

```
    System.getProperty("os.name").startsWith("Windows") )
{
    // Using reflection
    Class mNTS = Class.forName( "com.sun.security.auth.module.NTSystem" );

    Method classMethod = mNTS.getMethod( "getDomain" );
    String domain = ( String )classMethod.invoke( mNTS.newInstance() );
    domain = domain.toUpperCase();

    classMethod = mNTS.getMethod( "getName" );
    String name = ( String )classMethod.invoke( mNTS.newInstance() );
    name = name.toUpperCase();

    System.out.println( "Domain: " + domain + ", Name: " + name );
    if ( ( name != null ) && ( !name.equals( "" ) ) &&
        ( domain != null ) &&
        domain.equalsIgnoreCase( expectedDomain ) )
    {
        rtrnString = name;
    } else {
        System.out.println( "Expecting domain = " + expectedDomain );
        System.out.println( "User " + name + " must exist in Oracle"  );
    }
}
```

The `if` statement tests two properties of `System` to assure our OS architecture (`os.arch` system property) and OS name (`os.name` system property) are consistent with a Windows client.

To see all the properties of `System`, you can uncomment the top line of code. Calling the `list()` method of a `Properties` object will "print" the properties to an output stream—`System.out`, in our case.

The Expected Domain

In our identity code in Listing 8-4, we also get the Windows domain name from the `NTSystem.getDomain()` method. This must match the `expectedDomain` that we have hard coded.

Assume that our application code needs to get to resources on our organizational network, like Oracle databases; we should have a high threshold that client machines must cross before being allowed access to our corporate network. We do this with a NAC system. Part of NAC supervision is assuring that our clients are connected to our corporate domain services (Active Directory). The user would have to be logged into our domain to get network access.

If our network is not protected by NAC that assures our domain, then another avenue of attempted spoofing might be available. A hacker might set up her own domain with an imposter user identity (she may be posing as one of us), and have our code get the spoofed ID from her domain by `NTSystem`.

We avoid that potential issue, even if we have NAC, by requiring that the client computer be attached to our corporate domain. That is, `NTSystem` must return our expected domain name, or we don't accept the claim of user identity.

■ **Note** On a stand-alone system, the domain name may equal the system name alone.

USING UNIXSYSTEM

If you are going to use `UnixSystem` for your clients, then you will want to assure that your clients and the Oracle database are using the same naming services. This information is not provided by the `UnixSystem` class.

There is something you could use. `UnixSystem` provides the user identifier, `uid`, which is a numeric value representing the user. The `uid` is not necessarily unique, but in a single naming system, a specific user will have a specific `uid`.

To use the `uid`, your clients could transmit what they see for `uid` to Oracle database, and the database could assure that it sees the same `uid` for that user in its naming service. The level of assurance provided by this check is quite low, so I'm not advising it.

On the contrary, I recommend you use NAC to assure that all your UNIX and Linux clients are using the required naming service before they can get access to the network and the Oracle Database. I like NAC! That sounds like a campaign slogan, but that was before my time.

Case Sensitivity

You will notice in our code that we uppercase both the user name and domain. We also test the domain using the `equalsIgnoreCase()` method. That case-insensitive test of domain is just in case someone implements this code and neglects to type the `expectedDomain` in all caps. No matter how the domain name is cased coming from `NTSystem`, if the spelling is the same, it is the same domain.

In Java, we can do case-insensitive tests, but in Oracle database, we are always case sensitive. In the Windows/NT domains (Active Directory), depending on how user IDs are entered, you will find mixed cases. Windows domains are case insensitive: the user COFFIN is the same as Coffin or coffin.

There is a possibility that you or your application developers will use an `equals()` method instead of an `equalsIgnoreCase()` method in Java when dealing with the user ID. There is also a possibility (especially if you follow this book) that you will send the user ID to Oracle database and save it there or test it for existence in a database. For these possibilities, we will assure that our data is consistently cased where case-sensitivity is an issue. We will handle user IDs in upper case.

Access Oracle Database as Our Identified User

Oracle JDBC transmits a number of identity characteristics from the client to the server. Among these are the OS user ID, the IP address and in certain cases the *terminal* (client computer) name. We can query these items and use them for validation. Additionally, we can pass identity information to Oracle database, and we can assume a valid alternate identity and use the connected identity as a proxy.

All of these aspects of identity, when appropriately set, allow us to authorize access, and as importantly, to audit access to data. We want to know, monitor, and report who did what.

Examine the Oracle SSO Options for Programmers

Let's examine some of our options at this point. I am going to limit the options for doing single sign-on with Oracle database to the following:

- Have a standard connection to Oracle database as `appusr` and pass the OS user identity for authorization (perhaps) and auditing (certainly).
- Have a proxy session to Oracle database. The Oracle user will be named the same as our OS user, and will proxy through `appusr`. This requires a minimal Oracle user for each OS user, with the same name as the OS user. (This will be the default approach used throughout the chapters following this one. But if you are using another scheme at present, you will be glad to know that we can achieve SSO in that approach as well.)
- Have a connection pool for the app with Oracle users named the same as our OS users, all proxying through `appusr`. We will examine both lightweight (thin) connection pools and heavyweight (Oracle call interface, or OCI) connection pools. We will also implement the newest connection pool technology from Oracle database: universal connection pool (UCP).

Set a Client Identifier

The client identifier is an identity trait we can set for each Oracle connection. It can be used for a number of things, but for our purposes, we will set it equal to the user identity we get from `NTSystem` or `UnixSystem`.

Using an `OracleConnection` class (which extends the standard `Connection` class), we can set the client identifier using the code in Listing 8-5.

Listing 8-5. *Set the Client Identifier, doTest1()*

```
userName = OracleJavaSecure.getOSUserID();

String metrics[] =
    new String[OracleConnection.END_TO_END_STATE_INDEX_MAX];
metrics[OracleConnection.END_TO_END_CLIENTID_INDEX] = userName;
conn.setEndToEndMetrics( metrics, ( short )0 );
```

The last line is a call to set the end-to-end metrics for the connection. That call takes a `String` array, `metrics`, and an index of type `short` (a smaller integer) equal to 0—we cast the value 0 as a `short`. We set the size of the `String` array equal to the constant member of `OracleConnection` named `END TO END STATE INDEX MAX`, and we place the user identity in the array at the constant index `END TO END CLIENTID INDEX`.

Later, when we want to see what the client identifier is set to, we will examine it on the Oracle database by querying `SYS CONTEXT('USERENV','CLIENT IDENTIFIER')`. Oracle database provides the facility for creating and using application contexts in addition to the session context, `SYS CONTEXT`. Contexts are a convenience feature for storing information in the session as opposed to storing that data in a database table. Often application contexts are mentioned along with the security topic of Fine-Grained Access (FGA) control (See Chapter 12), but contexts by themselves do not provide security—just another storage place for information.

Prepare to Access HR Data

In all cases, we want to access data in the `HR` schema, so we can do a couple things to prepare for that. First, we will call the `appsec.p check hrview access` procedure to acquire our secure application role, `hrview role`. Then, we can set our current schema to the `HR` schema. This has no effect on access, but

allows us to call on views and procedures in the HR schema without prefixing each call with "HR." The code in Listing 8-6 does these things.

Listing 8-6. *Prepare to Access HR Data*

```
stmt.execute("CALL appsec.p check hrview access()");
stmt.execute("ALTER SESSION SET CURRENT_SCHEMA=hr");
rs = stmt.executeQuery( "SELECT COUNT(*) FROM v_employees_public" );
```

Notice there is no "HR." prefix on the view name, `v employees public`.

Update p_check_hrview_access Procedure, Non-Proxy Sessions

We will be making a couple sweeping changes to the `appsec.p check hrview access` procedure: one to handle regular connections, and one to handle proxy sessions. Once you decide which approach you will implement, you may comment or remove one or the other blocks of code. In the body of `appsec.p check hrview access`, we will place this code, Listing 8-7 for non-proxy sessions.

Listing 8-7. *Verify Non-Proxy Sessions*

```
IF( ( SYS CONTEXT( 'USERENV', 'IP ADDRESS' ) LIKE '192.168.%' OR
    SYS CONTEXT( 'USERENV', 'IP ADDRESS' ) = '127.0.0.1' )
AND TO CHAR( SYSDATE, 'HH24' ) BETWEEN 7 AND 18
AND SYS_CONTEXT( 'USERENV', 'SESSION_USER' ) = 'APPUSR'
AND SYS CONTEXT( 'USERENV', 'CLIENT_IDENTIFIER' ) = just_os_user )
THEN
    --DBMS_SESSION.SET_ROLE('hrview role');
    EXECUTE IMMEDIATE 'SET ROLE hrview role';
END IF;
```

■ **Note** You can find this script in the file named *Chapter8/AppSec.sql*.

The first two tests in the `if` statement, regarding `IP ADDRESS` and `SYSDATE` time constraints, are the same as what we implemented in Chapter 2. The third test assures that the `SESSION USER` is 'APPUSR'; that is, that the `appusr` user is connected to Oracle database. Prior to this point, we have limited the execution privilege for `appsec.p check hrview access` to `appusr`, but now with proxy sessions, we need to permit any Oracle user to execute our procedure, and assure that they are connected as `appusr` after the fact. We grant execute to `PUBLIC` (everybody) like this:

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

Remember, the goal of all these checks is to eventually set the role to `hrview role`, if everything checks out. There are at least the following two ways to set the role:

- Call to `DBMS SESSION.SET ROLE`.
- Immediately execute the `SET ROLE` command.

Both are valid, but we will continue to use the second form. It is more generic, and we will use the EXECUTE IMMEDIATE syntax in additional situations.

Assure Client Identifier and OS_USER

The fourth test in our procedure, shown Listing 8-7, assures that the OS user identity that we are passing to Oracle database as the client identifier is equal to the `OS USER` connection trait that JDBC passes to the database. This is merely a second check for us to assure that the client identifier set by the application, which represents the OS user identity, is the same as the OS user identity sensed by the Oracle Database. We assure that the application has not set client identity to something other than the OS user identity.

We get the OS user that was sensed by Oracle database, `just os user` through the JDBC client using the code in Listing 8-8. Really, it is just another session context environment setting; however, we need to massage the value.

Listing 8-8. *Getting the OS User Sensed by Oracle*

```
    just os user    VARCHAR2(40); -- Windows users are 20, allow 20 for domain
    backslash place NUMBER;
BEGIN
    -- Upper case OS USER and discard prepended domain name, if exists
    just os user := UPPER( SYS CONTEXT( 'USERENV', 'OS_USER' ) );
    -- Back slash is not an escape character in this context
    -- Negative 1 indicates count left from the right end, get last backslash
    backslash place := INSTR( just os user, '\', -1 );
    IF( backslash place > 0 )
    THEN
        just os user := SUBSTR( just os user, backslash place + 1 );
    END IF;
```

Note that we need to upper-case the `OS USER` sent by JDBC to Oracle Database using the `UPPER` function so that we can match by case the uppercase user identity we set in client identifier. In some cases, the domain name is prepended on the `OS USER` with a backslash separator like this: *ORGDOMAIN\OSUSER*. We use the `INSTR` (in string) function to find the place of the backslash, if any. Then we remove the domain name and backslash using the `SUBSTR` (substring) function. Looking back at Listing 8-7, it is this massaged `OS USER`, `just os user` that we compare to the value we set in the client identifier.

Testing that all these user identity traits are identical doesn't really buy us a lot of additional security, but sets another hurdle to trip up would-be hackers. You might be surprised at how often that one extra hurdle is all you needed to prevent a break-in, and how often a simple added hurdle is sold as a commercial security solution.

Audit Activity with Client Identifier Set

The following query will show you audit trail entries for connections where the client identifier is set. You may not have any just yet, but we will see these connections after our tests.

```
SELECT * FROM sys.dba audit trail
WHERE client id IS NOT NULL
ORDER BY TIMESTAMP DESC;
```

■ **Note** You can find this script in the file named *Chapter8/SecAdm.sql*.

Proxy Sessions

The idea behind proxy sessions is that we can connect as our application user but do work as an identified person user. This allows us to securely connect as our application user (no individual person passwords are used, nor indeed are they needed), and to audit the activities of individual persons. These goals do not differ radically from what we accomplish with setting our client identifier to the OS user identity, as in the previous section. The primary differences (in our efforts) are:

- With proxy sessions, we are not doing work as our application user, but as our individual user.
- With proxy sessions, an Oracle user must exist for each individual person who is going to connect.

Create Individual Person Users in Oracle

Why would we want to go to the effort of setting up individual person users in Oracle database when our application account is all we need to do the work? That is a good question, and the answer may give you reasons to choose to take either of the two routes we are describing in this section:

1. Using standard connections and placing the OS user identity in the Oracle client identifier.
2. Using proxy sessions.

Individual (human) users require some administrative activity; however, it can be minimal. Let's create one example user named `OSUSER` (feel free to substitute your user ID for `OSUSER` in these commands):

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

For each individual user, you only need to execute these commands. Each user must have the Create Session system privilege because proxying requires creation of an additional session. You will have one connection (as the application user) and two sessions—one as the application user and one as the proxying user.

You could script user creation from a list of all the users in your organization and quickly create an Oracle user for each person. The ease with which these users can be created is not an argument in their favor, as there is still an effort required. There is also an administrative effort required to remove or disable users when employees are no longer around.

Another argument against individual users in Oracle database is that some access and allocation is provided for each user. Each user will have an associated schema. This schema will not consist of much, but it will exist. And when you scroll through the list of schemas (in an IDE) to get to the one you want, you may be scrolling through dozens, hundreds, or thousands belonging to individual persons.