key has already been generated (is the passphrase char array null), and if it has not been generated, we call the method we described previously, `makeSessionSecretDESPassPhrase()` to build the artifacts.

Listing 6-3. *Generating the Password Key, `makeSessionSecretDESKey()`*

```
private static void makeSessionSecretDESKey() throws Exception {
    // DES Pass Phrase is generated on server and passed to client
    if( null == sessionSecretDESPassPhraseChars )
        makeSessionSecretDESPassPhrase();
    paramSpec = new PBEParameterSpec( salt, iterationCount );
    KeySpec keySpec = new PBEKeySpec( sessionSecretDESPassPhraseChars, salt,
            iterationCount );
    // Try with recommended algorithm
    sessionSecretDESKey = SecretKeyFactory.getInstance(
        sessionSecretDESAlgorithm ).generateSecret( keySpec );
    // See what algorithm used
    sessionSecretDESAlgorithm = sessionSecretDESKey.getAlgorithm();
    cipherDES = Cipher.getInstance( sessionSecretDESKey.getAlgorithm() );
}
```

Using our secret passphrase key artifacts, we first instantiate `paramSpec`, which is a `static` class member. We use that `paramSpec` member in multiple methods, so we create it as a static class member. It will be available in future Java stored procedure calls to static methods, from the same Oracle session.

We also instantiate a `KeySpec` class, which is local to the method and is only used here. The `keySpec` member is used by the `SecretKeyFactory` to generate a key of the algorithm type described in the `sessionSecretDESAlgorithm` member. This algorithm type is subject to the bug, so we will actually get a key of the algorithm type supported by the Oracle JVM version. After that, we get the actual algorithm by calling `sessionSecretDESKey.getAlgorithm()`. We also get an instance of our DES `Cipher` based on that algorithm.

Remember that we have passed our RSA public key from the client to Oracle. Now, on Oracle database we are building our secret password key, artifacts of which we will encrypt with the client public key and pass back to the client. We will also pass back the actual algorithm to the client, so we establish a common algorithm on the client.

Encrypting with the Public RSA Key

We have seen this code before, in the last chapter. We encrypt clear data (on Oracle) using our server-built copy of the RSA public key. However, previously this was a public method that we were calling directly to demonstrate RSA public key encryption. Now we are using it as a utility method to encrypt artifacts of our secret password key to send to the client, so we've made it private. Look at the code in Listing 6-4.

Listing 6-4. *Encrypting with the RSA Public Key, `getRSACryptData()`*

```
private static final RAW getRSACryptData( String extRSAPubMod,
    String extRSAPubExp, String clearText ) throws Exception
{
    byte[] clearBytes = clearText.getBytes();
    return getRSACryptData( extRSAPubMod, extRSAPubExp, clearBytes );
}
```

```
private static final RAW getRSACryptData( String extRSAPubMod,
    String extRSAPubExp, byte[] clearBytes ) throws Exception
{
    if( ( null == extRSAPubKey ) ||
        ( !saveExtRSAPubMod.equals( extRSAPubMod ) ) )
        makeExtRSAPubKey( extRSAPubMod, extRSAPubExp );
    cipherRSA.init( Cipher.ENCRYPT MODE, extRSAPubKey, random );
    return new RAW( cipherRSA.doFinal( clearBytes ) );
}
```

Notice also that we have two methods now by the same name: `getRSACryptData()`. They are different, however, by taking different arguments: one takes its clear text in `String` form; the other, in byte array form. These methods are said to have different signatures. We call this method overloading when we have multiple methods with the same name but different signatures.

The first method takes the clear text `String` and creates a byte array. Then to keep from repeating any code, the first method simply calls the second method and returns what the second method returns: the `RAW` encrypted text. We will be calling each of these methods multiple times to encrypt all the artifacts of our secret password key.

Returning Secret Password Key Artifacts to the Client

You will see so much similarity between the following four methods, that we will only describe each aspect once. These four methods return the passphrase, algorithm, salt, and iteration count, which when combined can be used to generate the DES secret password key.

These methods will be called from Java stored procedures, so they are `public` and `static`. Notice that these methods take the RSA public key artifacts as parameters, which they will use to encrypt the secret password key artifacts. The methods return the key artifacts as `RAW` data types, so we maintain fidelity of the data between Oracle database and the client. Let's look at the first of the bunch in Listing 6-5. It returns the passphrase.

Listing 6-5. *Encrypt the Passphrase, `getCryptSessionSecretDESPassPhrase()`*

```
public static final RAW getCryptSessionSecretDESPassPhrase(
    String extRSAPubMod, String extRSAPubExp )
{
    RAW rtrnRAW =
        new RAW( "getCryptSessionSecretDESPassPhrase() failed".getBytes() );
    try {
        if( null == sessionSecretDESKey ) makeSessionSecretDESKey();
        byte[] sessionSecretDESPassPhraseBytes =
            charArrayToByteArray( sessionSecretDESPassPhraseChars );
        rtrnRAW = getRSACryptData( extRSAPubMod, extRSAPubExp,
            sessionSecretDESPassPhraseBytes );
    } catch( Exception x ) {

        java.io.CharArrayWriter errorText =
            new java.io.CharArrayWriter( 32767 );
        x.printStackTrace( new java.io.PrintWriter( errorText ) );
        rtrnRAW = new RAW( errorText.toString().getBytes() );

    }
    return rtrnRAW;
```

```
}
```

We've discussed a lot of the `Exception` handling already. This method is generally called first of the current foursome, and includes a bit more error reporting. Because we are returning a `RAW`, we can return up to 32,767 bytes of data. In the `catch` block, we instantiate a `CharArrayWriter` of size 32,767 characters in an array in memory. Then by stacked calls, we instantiate a `PrintWriter`, which points at the `CharArrayWriter`. Then we print the `Exception` *s*tack trace to that `PrintWriter`. As a result, we get the stack trace in a char array writer, `errorText`. We call `errorText.toString().getBytes()` to get a byte array of the stack trace, and then we instantiate a `RAW` from that, which we can pass back to the client.

At the client, if we have trouble deciphering the passphrase that we are seeking from the `RAW` that has been returned, we can read the `RAW` as a `String` and see the `Exception` stack trace. This is a handy troubleshooting practice in a client/server environment when you'd like to see the error that the server sees, not just the error at the client.

Listing 6-6 shows the second of this bunch of methods. This one returns the name of the actual algorithm that's in use on the Oracle database. In the `try` block, you'll see a feature common to all these methods. We test to see if the `sessionSecretDESKey` is `null`, and if it is, we call `makeSessionSecretDESKey()` (described earlier) to create the secret password key.

Listing 6-6. *Encrypt the Algorithm Name, `getCryptSessionSecretDESAlgorithm ()`*

```
public static final RAW getCryptSessionSecretDESAlgorithm(
    String extRSAPubMod, String extRSAPubExp )
{
    RAW rtrnRAW =
        new RAW( "getCryptSessionSecretDESAlgorithm() failed".getBytes() );
    try {
        if( null == sessionSecretDESKey ) makeSessionSecretDESKey();
        rtrnRAW = getRSACryptData( extRSAPubMod, extRSAPubExp,
            sessionSecretDESAlgorithm );
    } catch( Exception x ) {}
    return rtrnRAW;
}
```

The last common call in the `try` block is a call to the `getRSACryptData()` method described previously) to encrypt the secret password key artifact; in this case, the algorithm name. That generates a `RAW` data type containing the encrypted artifact, which will be returned to the client.

Please take note of what is returned if an exception is called. We still return `rtrnRAW`, but its value is the bytes of the string `"getCryptSessionSecretDESAlgorithm() failed"`. Having this reported back to the client can be helpful for troubleshooting. Also, note how we get the bytes of that `String`—we treat the value between quotation marks as if it were already *a* `String` object, calling the `getBytes()` method. Is that allowed? Yes!

The last two methods in this group, shown in Listings 6-7 and 6-8, return the salt and iteration count as encrypted `RAW` data types.

Listing 6-7. *Encrypt the Salt, `getCryptSessionSecretDESSalt ()`*

```
public static final RAW getCryptSessionSecretDESSalt( String extRSAPubMod,
    String extRSAPubExp )
{
    RAW rtrnRAW = new RAW( "getCryptSessionSecretDESSalt() failed".getBytes() );
    try {
        if( null == sessionSecretDESKey ) makeSessionSecretDESKey();
```

```
            rtrnRAW = getRSACryptData( extRSAPubMod, extRSAPubExp, salt );
        } catch( Exception x ) {}
        return rtrnRAW;
    }
```

Listing 6-8. *Encrypt the Itteration Count, `getCryptSessionSecretDESIterationCount ()`*

```
    public static final RAW getCryptSessionSecretDESIterationCount(
        String extRSAPubMod, String extRSAPubExp )
    {
        RAW rtrnRAW =
            new RAW( "getCryptSessionSecretDESIterationCount() failed".getBytes() );
        try {
            if( null == sessionSecretDESKey ) makeSessionSecretDESKey();

            byte[] sessionSecretDESIterationCountBytes =
                { ( byte )iterationCount };

            rtrnRAW = getRSACryptData( extRSAPubMod, extRSAPubExp,
                sessionSecretDESIterationCountBytes );
        } catch( Exception x ) {}
        return rtrnRAW;
    }
```

This last method for returning the encrypted iteration count creates a byte array of one `byte`. The only `byte` in the array is the iteration count `int`, cast as a `byte`. In that way, we can call the same method to encrypt the iteration count as a byte array. We will have to reverse that process when we decrypt it at the client.

Encrypting Data with Our Secret Password

We will call the same method on both the client and Oracle database to encrypt data using the secret password key. The syntax, in Listing 6-9, should be familiar by now. This method takes a `String` of clear text data and returns a `RAW` of encrypted data. Notice that initializing the `Cipher` using the `sessionSecretDESKey` is very similar to how we do it with the RSA keys, except that we also provide the `paramSpec`.

Listing 6-9. *Encrypt Data with Secret Password, `getCryptData()`*

```
    public static final RAW getCryptData( String clearData ) {
        if( null == clearData ) return null;
        RAW rtrnRAW = new RAW( "getCryptData() failed".getBytes() );
        try {
            if( null == sessionSecretDESKey ) makeSessionSecretDESKey();
            cipherDES.init( Cipher.ENCRYPT_MODE, sessionSecretDESKey, paramSpec );
            rtrnRAW = new RAW( cipherDES.doFinal( clearData.getBytes() ) );
        } catch( Exception x ) {}
        return rtrnRAW;
    }
```

Like the methods we described earlier for encrypting and returning our secret password key artifacts, this method tests whether the `sessionSecretDESKey` has been instantiated, and attempts to create it if not. This is good practice on the Oracle database, but is presumptuous on the client (where

we do not want to originate the secret password key.) It is mandatory that developers understand that the client MUST get the secret password key from the Oracle database first, and only then may use it to encrypt data for insert or update to Oracle. No harm will come if developers fail to follow that guideline, but their code will not work.

Oracle Structures for Secret Password Encryption

In the last chapter, we created an Oracle function and a procedure to demonstrate use of our RSA public key encryption in a client/server environment. Now we will create an Oracle package with multiple functions and procedures, including Java stored procedures, to handle our secret password encryption.

The package will be placed in the Application Security, `appsec` schema. As the `appsec` user, first set your role to the non-default role, `appsec role` with this command:

```
SET ROLE appsec role;
```

■ **Note** You can find a script of the following commands in the file named *Chapter6/AppSec.sql*.

As you read through this section, you can follow along in the referenced code file, and see in context the code that's discussed in this text. Also, you can execute the code to create the Oracle structures as you read about them. We will use these structures when we run tests at the end of the chapter.

Package to Get Secret Password Artifacts and Encrypted Data

A package in Oracle database is a set of functions and procedures that can be configured as a group. Permissions to access the functions and procedures in a package are granted by granting executable permission on the package. We will learn one additional benefit of Oracle packages when we get to Chapter 7: we can define new data types and use them in Oracle packages.

An Oracle package has two parts, the specification and the body. The specification provides the signature for each procedure or function, but the actual code is only included in the body. This two-part identity for packages allows PL/SQL programmers to share the functionality (the specification) without sharing the code (the body). You might do this for separation of duty, security through obfuscation, or simply to protect your intellectual property. C programmers will recognize this approach as being analogous to having a header file for each code file.

Application Security Package Specification

An Oracle package specification merely defines the functions and procedures, listing the expected parameters and the return type for functions. Listing 6-10 shows the `app sec pkg` package specification. Execute this as `appsec` user.

Listing 6-10. *Package Specification for Secret Password Encryption*

```
CREATE OR REPLACE PACKAGE appsec.app sec pkg IS

    -- For Chapter 6 testing only - move to app in later versions of this package
    PROCEDURE p get shared passphrase(
```

```
        ext modulus                VARCHAR2,
        ext exponent               VARCHAR2,
        secret pass salt       OUT RAW,
        secret pass count      OUT RAW,
        secret pass algorithm OUT RAW,
        secret pass            OUT RAW,
        m err no               OUT NUMBER,
        m err txt              OUT VARCHAR2 );

    -- For Chapter 6 testing only - remove in later versions of this package
    PROCEDURE p get des crypt test data(
        ext modulus                VARCHAR2,
        ext exponent               VARCHAR2,
        secret pass salt       OUT RAW,
        secret pass count      OUT RAW,
        secret pass algorithm OUT RAW,
        secret pass            OUT RAW,
        m err no               OUT NUMBER,
        m err txt              OUT VARCHAR2,
        test data                  VARCHAR2,
        crypt data             OUT RAW );

    FUNCTION f get crypt secret pass( ext modulus VARCHAR2,
        ext exponent VARCHAR2 ) RETURN RAW;

    FUNCTION f get crypt secret algorithm( ext modulus VARCHAR2,
        ext exponent VARCHAR2 ) RETURN RAW;

    FUNCTION f get crypt secret salt( ext modulus VARCHAR2,
        ext exponent VARCHAR2 ) RETURN RAW;

    FUNCTION f get crypt secret count( ext modulus VARCHAR2,
        ext exponent VARCHAR2 ) RETURN RAW;

    FUNCTION f get crypt data( clear text VARCHAR2 ) RETURN RAW;

    FUNCTION f get decrypt data( crypt data RAW ) RETURN VARCHAR2;

    -- For Chapter 6 testing only - remove in later versions of this package
    FUNCTION f show algorithm RETURN VARCHAR2;

END app sec pkg;
/
```

See the list of functions in the second half of the package specification. We have one function to return the encrypted `RAW` data for each of the secret password key artifacts: `f get crypt secret pass`, `f get crypt secret algorithm`, `f get crypt secret salt`, and `f get crypt secret count`. We also have functions to encrypt clear text and return an encrypted `RAW`, `f get crypt data`, and to decrypt a `RAW` and return clear text, `f get decrypt data` (we have not seen the Java side of that process yet).

Above our functions, we specify two procedures: `p get shared passphrase` and `p get des crypt test data`. Each of these procedures takes the client RSA public key modulus and exponent as input parameters, and returns the secret password key artifacts as `OUT` parameters. We are

handling errors as we described in the previous chapter with OUT parameters for the error number and error text. Additionally, the p get des crypt test data procedure takes a clear text input parameter and returns an encrypted RAW as an additional OUT parameter.

Both of these procedures are for testing in this chapter only and will be moved or removed from the package in future chapters. The last function, f show algorithm, is also for testing only in this chapter and will be removed later.

Application Security Package Body: Functions

The function and procedure definitions in our package specification must be duplicated exactly in our package body. The body contains not just the definitions, but also the code of the procedures and functions. You can execute this package body at this time; the package will be created in the Oracle database.

Here is one example function from the body listing for our consideration, in Listing 6-11. We pass our RSA public key modulus and exponent to the function f get crypt secret pass. It passes them along to the Java method named getCryptSessionSecretDESPassPhrase() (discussed above). That Java method returns a RAW, the secret password key passphrase, encrypted with the RSA public key. And the function returns the RAW value that the Java method returned to it.

Listing 6-11. *Function to Return the Secret Passphrase*

```
    FUNCTION f get crypt secret pass( ext modulus VARCHAR2,
                  ext exponent VARCHAR2 )

    RETURN RAW

    AS LANGUAGE JAVA
    NAME 'orajavsec.OracleJavaSecure.getCryptSessionSecretDESPassPhrase( java.lang.String,
java.lang.String ) return oracle.sql.RAW';
```

This same approach is taken to get each artifact of the secret password key, and to get encrypted and decrypted data. We also have a function that takes no input parameters and returns the algorithm string as an OUT parameter. That function, f show algorithm, is just for testing in this chapter.

Application Security Package Body: Procedures

The procedures in the package body are the locus of our primary efforts. These procedures are shown in Listings 6-12 and 6-13. Note that both of these procedures are for testing only in this chapter and will be removed and replaced in future chapters. The first procedure we will look at is p get shared passphrase, which returns the artifacts of a secret password key to the client. If the key does not yet exist, it will be created when we call f get crypt secret salt. Remember that our encryption keys are specific to an Oracle session, so we need to keep the session open in order to use the secret password key for encryption.

Listing 6-12. *Procedure to get the Shared Password Key, p get shared passphrase*

```
    PROCEDURE p get shared passphrase(
        ext modulus              VARCHAR2,
        ext exponent             VARCHAR2,
        secret pass salt      OUT RAW,
        secret pass count     OUT RAW,
```

```
    secret pass algorithm OUT RAW,
    secret pass           OUT RAW,
    m err no              OUT NUMBER,
    m err txt             OUT VARCHAR2 )
IS BEGIN
    m err no := 0;
    secret pass salt := f_get_crypt_secret_salt( ext modulus, ext exponent );
    secret pass count := f get crypt secret count( ext modulus, ext exponent );
    secret pass := f get crypt secret pass( ext modulus, ext exponent );
    secret pass algorithm :=
        f get crypt secret algorithm( ext modulus, ext exponent );
EXCEPTION
    WHEN OTHERS THEN
        m err no := SQLCODE;
        m err txt := SQLERRM;
END p get shared passphrase;
```

Observe the list of `IN` and `OUT` parameters in the Oracle procedure, `p get shared passphrase`, in Listing 6-12. We provide the RSA public key modulus and exponent, then get out the artifacts of the secret password key, and the error number and text. Each of the artifacts is acquired by a call to one of the Oracle functions, like this:

```
secret pass := f get crypt secret pass( ext modulus, ext exponent );
```

Each of the functions, like `f get crypt secret pass`, that are called from `p get shared passphrase` are Java stored procedures. We saw the code for one of those functions in Listing 6-11. All the substantial work of the Java stored procedures is done in our Java code, `OracleJavaSecure` on the Oracle database

A second procedure that we will look at here is `p get des crypt test data`. It is very similar to `p get shared passphrase` except that `p get des crypt test data` has two additional parameters, one `IN` and one `OUT`, as shown in Listing 6-13. These parameters will be used to submit clear text to Oracle database and to return that text in encrypted form—encrypted with the secret password key.

Listing 6-13. *Procedure to get Encrypted Data, `p get des crypt test data`*

```
PROCEDURE p get des crypt test data(
    ext modulus               VARCHAR2,
    ext exponent              VARCHAR2,
    secret pass salt      OUT RAW,
    secret pass count     OUT RAW,
    secret pass algorithm OUT RAW,
    secret pass           OUT RAW,
    m err no              OUT NUMBER,
    m err txt             OUT VARCHAR2,
    test_data                 VARCHAR2,
    crypt_data            OUT RAW )
IS BEGIN
    m err no := 0;
    secret pass salt := f get crypt secret salt( ext modulus, ext exponent );
    secret pass count := f get crypt secret count( ext modulus, ext exponent );
    secret pass := f get crypt secret pass( ext modulus, ext exponent );
    secret pass algorithm :=
        f get crypt secret algorithm( ext modulus, ext exponent );
    crypt_data := f_get_crypt_data( test_data );
```

```
EXCEPTION
    WHEN OTHERS THEN
        m err no := SQLCODE;
        m err txt := SQLERRM;
END p get des crypt test data;
```

We send clear text `test data` from the client to Oracle, and this procedure returns `crypt data` after encryption by a call to the `f get crypt data` function. That function is also a Java stored procedure.

Java Methods for Secret Password Decryption

Once we've called the appsec procedures to get the DES secret password key artifacts and encrypted data back to the client, we need to

1. decrypt the artifacts with the RSA private key
2. generate the DES secret password key
3. decrypt the data with the secret password key

As a rule, I try to limit the number of steps that I require of developers to accomplish work. Why make a developer call three methods when they can call a single method that accomplishes the other calls for them? The application developer's goal is to decrypt data, so we provide a method for them to do just that.

■ **Note** You can find this code in the file *Chapter6/orajavsec/OracleJavaSecure.java*.

Decrypting Data Using the Secret Password Key

After the client application has called the `p get des crypt test data` procedure, we have them call the method `getDecryptData()` shown in Listing 6-14.

Listing 6-14. *Build Secret Password and Decrypt Data, `getDecryptData()`*

```
public static final String getDecryptData( RAW cryptData,
    RAW cryptSecretDESPassPhrase, RAW cryptSecretDESAlgorithm,
    RAW cryptSecretDESSalt, RAW cryptSecretDESIterationCount )
{
    String rtrnString = "getDecryptData() A failed";
    try {
        if( ( null == sessionSecretDESKey ) || testAsClientAndServer ) {
            decryptSessionSecretDESPassPhrase( cryptSecretDESPassPhrase,
                cryptSecretDESAlgorithm, cryptSecretDESSalt,
                cryptSecretDESIterationCount );
            makeSessionSecretDESKey();
        }
        rtrnString = getDecryptData( cryptData );
    } catch( Exception x ) {
```

```
            x.printStackTrace();
        }
        return rtrnString;
    }
```

The first thing we do in the `try` block is to test whether the `sessionSecretDESKey` has already been instantiated. If not, then we call two methods: `decryptSessionSecretDESPassPhrase()` (discussed in the next section) and `makeSessionSecretDESKey()`. We discussed `makeSessionSecretDESKey()` earlier in this chapter—it is the same method we called to build to the secret password key initially on the Oracle database. We are calling it again on the client to build an identical key.

When we test whether we already have the `sessionSecretDESKey`, we also test the `boolean testAsClientAndServer`. The `testAsClientAndServer boolean` is always `false`, unless we are testing the `OracleJavaSecure` class from its `main()` method. In the `main()`, when we set this `boolean` to `true`, we can replace a locally generated DES secret password key with one coming from Oracle database at different stages of testing. We will examine the code of the `main()` method a bit later in this chapter.

The `getDecryptData()` method is overloaded with a version that assumes the secret password key has been built and does the decryption. It takes a `RAW` and returns the clear text as a `String`. The first `getDecryptData()` method (shown previously) calls this second `getDecryptData()` method, see Listing 6-15.

Listing 6-15. *Decrypt Data with Existing Secret Password, `getDecryptData()`*

```
    public static final String getDecryptData( RAW cryptData ) {
        if( null == cryptData ) return null;
        String rtrnString = "getDecryptData() B failed";
        try {
            cipherDES.init( Cipher.DECRYPT MODE, sessionSecretDESKey, paramSpec );
            rtrnString = new String( cipherDES.doFinal( cryptData.getBytes() ) );
        } catch( Exception x ) {
            //x.printStackTrace();
            //rtrnString = x.toString();
        }
        return rtrnString;
    }
```

This same, second `getDecryptData()` method is also called to decrypt data on the Oracle database for encrypted data inserts and updates coming from the client. There on the Oracle database, we presumably know that we already have our DES secret password key.

Decrypting the DES Passphrase using RSA Private Key

The `decryptSessionSecretDESPassPhrase()` method uses the client's RSA private key to decrypt all the artifacts of the server DES secret password key. The code is presented in Listing 6-16.

Listing 6-16. *Decrypt Secret Password Key Artifacts, `decryptSessionSecretDESPassPhrase()`*

```
    private static void decryptSessionSecretDESPassPhrase(
        RAW cryptSecretDESPassPhrase, RAW cryptSecretDESAlgorithm,
        RAW cryptSecretDESSalt, RAW cryptSecretDESIterationCount )
        throws Exception
    {
        cipherRSA.init( Cipher.DECRYPT MODE, locRSAPrivKey );
```

```
        byte[] cryptBytes;
        cryptBytes = cryptSecretDESPassPhrase.getBytes();
        sessionSecretDESPassPhraseChars =
            byteArrayToCharArray( cipherRSA.doFinal( cryptBytes ) );
        cryptBytes = cryptSecretDESAlgorithm.getBytes();
        sessionSecretDESAlgorithm = new String( cipherRSA.doFinal( cryptBytes ) );
        cryptBytes = cryptSecretDESSalt.getBytes();
        salt = cipherRSA.doFinal( cryptBytes );
        cryptBytes = cryptSecretDESIterationCount.getBytes();
        iterationCount = cipherRSA.doFinal( cryptBytes )[0];
        //System.out.println( "\n" + new String( sessionSecretDESPassPhraseChars ) );
        //System.out.println( sessionSecretDESAlgorithm );
        //System.out.println( new String( salt ) );
        //System.out.println( iterationCount );
    }
```

For each artifact, we convert the encrypted `RAW` into a byte array and pass it to the `cipherRSA` member to be decrypted. We use the byte array coming back from the `Cipher` to populate our static class members with an appropriate data type. Keep the value as a byte array for `salt`, char array for `sessionSecretDESPassPhraseChars` by calling the `byteArrayToCharArray()` method, `String` for `sessionSecretDESAlgorithm` by instantiating a `new String()`, and a single `byte`, automatically cast as an `int` for `iterationCount`.

If you are interested in observing these session-specific, random artifacts and the negotiated algorithm as they arrive at the client, you can uncomment the `System.out.println()` calls at the end of the method. However, you should only do this temporarily: the `System.out.println()` calls have been removed from the code in the next chapter.

Ancillary Methods for Array Conversion

In two places in the preceding code, we called some ancillary array conversion methods that we have defined in the `OracleJavaSecure` code, shown in Listing 6-17. One takes a byte array and converts it to a char array. The other does the opposite. We call `byteArrayToCharArray()` from `decryptSessionSecretDESPassPhrase()` (see Listing 6-16) when we get `sessionSecretDESPassPhraseChars`, and we call `charArrayToByteArray()` from `getCryptSessionSecretDESPassPhrase()` (see Listing 6-5) when we are encrypting `sessionSecretDESPassPhraseChars`.

Listing 6-17. *Array Conversion Methods, `byteArrayToCharArray()` and `charArrayToByteArray()`*

```
    static char[] byteArrayToCharArray( byte[] bytes ) {
        char[] rtrnArray = new char[bytes.length];
        for ( int i = 0; i < bytes.length; i++ ) {
            rtrnArray[i] = ( char )bytes[i];
        }
        return rtrnArray;
    }

    static byte[] charArrayToByteArray( char[] chars ) {
        byte[] rtrnArray = new byte[chars.length];
        for ( int i = 0; i < chars.length; i++ ) {
            rtrnArray[i] = ( byte )chars[i];
```

```
        }
        return rtrnArray;
    }
```

We can normally convert these array types back and forth just by casting, as in the following example. However, we need to be aware of the implications of narrowing and widening conversions. We must restrict the char values in our array to standard ASCII characters, not 16-bit Unicode characters, in order to not lose information in the conversions.

```
byte[] bAr = new byte[10];
char[] cAr = (char[])bAr;
bAr = (byte[])cAr;
```

That kind of array casting is not supported from within the JDeveloper IDE (and possibly elsewhere), so we will instead rely on our ancillary methods. JDeveloper is nice because it is free, and it is highly tailored for working with Oracle databases; it handles Oracle views better than any other IDE. You can find JDeveloper on the Oracle corporate web site at `www.oracle.com`.

You may wonder why we maintain the passphrase as a char array. That is the format we need when we build the `PBEKeySpec`.

Method Used to Show Actual Algorithm

Listing 6-18 presents the `showAlgorithm()` method. This is actually duplicative functionality. Take a look at the code for the `decryptSessionSecretDESPassPhrase()` method (shown previously) and you will see that we get `sessionSecretDESAlgorithm` from Oracle database to a `String` on the client that we could simply print out.

The only additional assurance in selecting this directly from the Oracle database (via the function `f show algorithm`) is that there is no mix up during transfer. We have already built the function in `app sec pkg` that will call this method to return the algorithm name. We can also call this method from the client (before calling the server) and compare the algorithms used.

Listing 6-18. *Display the Secret Password Algorithm Name in Use,* `showAlgorithm()`

```
    public static final String showAlgorithm() {
        String rtrnString = "showAlgorithm failed";
        try {
            rtrnString = sessionSecretDESKey.getAlgorithm();
        } catch( Exception x ) {
            rtrnString = x.toString();
        } finally {
            return rtrnString;
        }
    }
```

This is a temporary method for testing only in the chapter, and we will remove it, and the Oracle function that calls it, from the code in future chapters.

Testing DES Encryption on the Client Only

Once again, we are going to do our client-side-only testing by calling our methods from the `main()` method of `OracleJavaSecure`. The code for the first part of `main()` is shown in Listing 6-19. Start out by

getting our client public key modulus and exponent, which in the process generates the RSA Public/private key pair, if not existing.

Listing 6-19. *Code for Client-Only Testing, from* `main()`

```
String clientPubModulus = getLocRSAPubMod();
String clientPubExponent = getLocRSAPubExp();

// Emulates server actions
RAW mCryptSessionSecretDESPassPhrase =
    getCryptSessionSecretDESPassPhrase( clientPubModulus,
    clientPubExponent );
RAW mCryptSessionSecretDESSalt =
    getCryptSessionSecretDESSalt( clientPubModulus,
    clientPubExponent );
RAW mCryptSessionSecretDESAlgorithm =
    getCryptSessionSecretDESAlgorithm( clientPubModulus,
    clientPubExponent );
RAW mCryptSessionSecretDESIterationCount =
    getCryptSessionSecretDESIterationCount( clientPubModulus,
    clientPubExponent );
RAW cryptData = getCryptData( "Monday" );

testAsClientAndServer = true;

// As client
System.out.println( getDecryptData( cryptData,
    mCryptSessionSecretDESPassPhrase, mCryptSessionSecretDESAlgorithm,
    mCryptSessionSecretDESSalt, mCryptSessionSecretDESIterationCount ) );
System.out.println( showAlgorithm() );
```

Next, we emulate the server, having received the modulus and exponent, getting the DES secret password key artifacts from `getCryptSessionSecretDESPassPhrase()` and other methods. In the process, a copy of the public key is built from the artifacts, the DES secret password key is generated and each of the secret password key artifacts is encrypted using the RSA public key.

We also emulate the server, encrypting some data with the secret password key:

```
RAW cryptData = getCryptData( "Monday" );
```

Our next step in testing assumes we are back on the client, and having received all the encrypted secret password key artifacts, we generate a copy of the secret password key from those artifacts. To do that, we set `testAsClientAndServer` to be `true` in order to overwrite the secret password key that we just created while emulating the server (even though the key will be identical):

```
testAsClientAndServer = true;
```

Now again as the client, we call `getDecryptData()` with all the DES secret password key artifacts, and the encrypted data. This will create a new DES secret key based on the artifacts, then decrypt the data using that key. We will print out the decrypted data, which should be the same as what we encrypted earlier. Also, we will print out the DES algorithm name that we used for secret password encryption.

Running the Code

We assume you followed the procedure to run the code in the previous chapter. This chapter will use an identical procedure. In a command prompt, change directories to *Chapter6*. Compile the code with this command:

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

If you have any problems, refer to Chapter 3 for directions on compiling at the command prompt and setting your environment `CLASSPATH` to include *ojdbc6.jar*. Then run the code from that same directory with this command:

```
java orajavsec.OracleJavaSecure.OracleJavaSecure
```

Observing the Results

The following two lines will be printed as a result of the commands issued in the preceding section:

```
Monday
PBEWithSHA1AndDESede
```

We encrypted the string "Monday" using the DES secret password key when we emulated the server, and we passed the encrypted data back to the client, along with the secret password key artifacts, encrypted with the client RSA public key. Back at the client, we built a duplicate DES key using the artifacts, and decrypted the encrypted data. We printed out the decrypted data and saw "Monday" at the command prompt. Then we printed the negotiated algorithm. If you are using JDK 1.6 or later on your workstation, you will see `PBEWithSHA1AndDESede`; however, if you are using JDK 1.5, you will see `PBEWithMD5AndDES`.

Coding to Test Client/Server Secret Password Encryption

The next line is at the top of our Java code in *OracleJavaSecure.java*. Uncomment it and copy the entire code into your Oracle client.

```
CREATE OR REPLACE AND RESOLVE JAVA SOURCE NAMED appsec."orajavsec/OracleJavaSecure" AS
```

For security, scroll down to the class body and assure you do not have a valid password in the connection string. If you do, remove the password from the connection string before executing this command in Oracle.

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

Execute the script in your Oracle client (e.g., SQL*Plus) to load the Java code into the Oracle database. This command, as we have seen, loads the Java code into the Oracle database and compiles it.

Setting the Code to Test Server as well as Client

To compile and execute `OracleJavaSecure` on your client, that first line that we uncommented to run on the Oracle database, needs to be commented:

```
//CREATE OR REPLACE AND RESOLVE JAVA SOURCE NAMED appsec."orajavsec/OracleJavaSecure" AS
```

Scroll down to the class body and set the password in the connection string. Also correct any of the other addresses and names for your connection string.

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

Also set the `testingOnServer` `boolean` to `true`:

```
private static boolean testingOnServer = true;
```

Save the file.

You may have already executed the `app sec pkg` package specification and body on Oracle, from earlier in this chapter. If you haven't done so, do that now. This will create the Oracle structures we need to do secret password encryption.

Consider the Server Portion of the main() Method

This time when we run through the `main()` method of `OracleJavaSecure`, we will pass the `testingOnServer` test, so we will execute the remainder of `main()`, as shown in Listing 6-20. We declare a couple member variables to hold the error number and error message coming back from Oracle, `errNo` and `errMsg`.

Because we are running from the client (not on Oracle database), we need to load the Oracle-specific driver (assuming we might not be using JDK 1.6 or later). And we will set up the Oracle connection for use: note that we will be connecting as `appsec` user.

We will be using an Oracle-specific `OracleCallableStatement`, which allows us to retrieve `OUT` parameters back from Oracle, and to transfer Oracle-specific data types.

Listing 6-20. *Code for Client/Server Testing, from `main()`*

```
if( testingOnServer ) {
    int errNo;
    String errMsg;
    // Since not on the Server, must load Oracle-specific Driver
    Class.forName( "oracle.jdbc.driver.OracleDriver" );
    // This will set the static member "conn" to a new Connection
    conn = DriverManager.getConnection( appsecConnString );
    OracleCallableStatement stmt;
```

Getting the DES Secret Password from Oracle

Our first procedure call, in Listing 6-21, is to `p get shared passphrase`. This will simply test the exchange of RSA and DES keys between our client and Oracle. We hand the procedure our RSA public key modulus and exponent, and in return get the DES secret password key artifacts, encrypted by the Oracle database using the public key. Notice that we register the `OUT` parameters and either set or `setNull` all our parameters.

Listing 6-21. *Get Shared Passphrase, from `main()`*

```
stmt = ( OracleCallableStatement )conn.prepareCall(
    "CALL app sec pkg.p_get_shared_passphrase(?,?,?,?,?,?,?,?)" );
stmt.registerOutParameter( 3, OracleTypes.RAW );
stmt.registerOutParameter( 4, OracleTypes.RAW );
```

```
stmt.registerOutParameter( 5, OracleTypes.RAW );
stmt.registerOutParameter( 6, OracleTypes.RAW );
stmt.registerOutParameter( 7, OracleTypes.NUMBER );
stmt.registerOutParameter( 8, OracleTypes.VARCHAR );
stmt.setString( 1, clientPubModulus );
stmt.setString( 2, clientPubExponent );
stmt.setNull(   3, OracleTypes.RAW );
stmt.setNull(   4, OracleTypes.RAW );
stmt.setNull(   5, OracleTypes.RAW );
stmt.setNull(   6, OracleTypes.RAW );
stmt.setInt(    7, 0 );
stmt.setNull(   8, OracleTypes.VARCHAR );
stmt.executeUpdate();

errNo = stmt.getInt( 7 );
if( errNo != 0 ) {
    errMsg = stmt.getString( 8 );
    System.out.println( "Oracle error " + errNo +
        ", " + errMsg );
    System.out.println( (stmt.getRAW( 3 )).toString() );
} else {
    mCryptSessionSecretDESSalt = stmt.getRAW( 3 );
    mCryptSessionSecretDESIterationCount = stmt.getRAW( 4 );
    mCryptSessionSecretDESAlgorithm = stmt.getRAW( 5 );
    mCryptSessionSecretDESPassPhrase = stmt.getRAW( 6 );
}
if( null != stmt ) stmt.close();
```

After executing the statement, we get the error number, `errNo` coming back from Oracle database and assure it's 0. If not, we report the error.

If there was no error, then we, as the client application, set a method member equal to each of the encrypted artifacts, like the `mCryptSessionSecretDESSalt`. Now we have everything we need to build an exact copy of the secret password key that was generated on the Oracle database. At this point, we have fully exchanged keys; however, we have not yet built a copy of the DES secret password key on the client. Later in this code, we will decrypt each artifact using our private key, then build the secret password key and use it to exchange encrypted data with Oracle.

Because we are done with this statement, we close it. It is a matter of consistency and diligence that before we close the statement we assure that it is not `null`. In reality, if `stmt` were `null`, we would have thrown an exception in the earlier code with the first `registerOutParameter()` call.

Seeing the Negotiated Algorithm for Password-Based Encryption

We call the function, `f show` algorithm in order to display the algorithm name that the Oracle database selected (see Listing 6-22). Because the Oracle 11g JVM is based on the standard JVM version 1.5, we will see that Oracle database manifests the aforementioned bug in this regard and selects `PBEWithMD5AndDES` as the protocol, instead of what we requested, `PBEWithSHA1AndDESede`. This will serve as the common algorithm negotiated with the client.

Listing 6-22. *Show Algorithm, from* `main()`

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

```
    "{? = call app sec pkg.f_show_algorithm}" );
stmt.registerOutParameter( 1, OracleTypes.VARCHAR );
stmt.executeUpdate();
System.out.println( stmt.getString(1) );
if( null != stmt ) stmt.close();
```

Of note here is the syntax for calling an Oracle function, rather than a procedure. The function always returns a value. The leading "? =" represents the return value, and our statement parameter (1) is that value. Each question mark in a prepared callable statement is a parameter, whether it comes before or after the procedure or function name. The parameters are always numbered from left to right with 1 (one) being the first.

The format used in this function call, with the open and close curly brackets, is referred to as SQL92 syntax. That is a reference to the international standard for SQL that was adopted in 1992. Another form of syntax that can be used for calling stored procedures and functions is the PL/SQL Block syntax (with `begin` and `end` statements). I have found that with much older versions (e.g., jdbc14.jar) of the Oracle drivers (although not as old as SQL92) the PL/SQL Block syntax will work when the SQL92 syntax will not. Here is the same `prepareCall()` method call from Listing 6-22 in PL/SQL Block syntax:

```
stmt = ( OracleCallableStatement )conn.prepareCall(
    "begin ? = call app sec pkg.f show algorithm; end;" );
```

Calling Oracle Database to get Encrypted Data

Next we want to demonstrate getting encrypted data back from Oracle database and decrypting it on the client with a local copy of the DES secret password key. In Listing 6-23, you can see we call the procedure, `p get des crypt test data`. Again, we pass our public key artifacts and retrieve the encrypted secret password key artifacts. Because this process is something we will see repeated throughout the remainder of this book, I will just call this "key exchange". We just returned from another procedure that retrieved the secret password key artifacts, so we don't bother to set our method members again—those lines are commented. Note that all these calls are occurring in the same Oracle session, so existing keys are used—there is no additional key generation.

Listing 6-23. *Get DES Crypt Test Data, from* `main()`

```
stmt = ( OracleCallableStatement )conn.prepareCall(
    "CALL app sec pkg.p_get_des_crypt_test_data(?,?,?,?,?,?,?,?,?,?)" );
stmt.registerOutParameter( 3, OracleTypes.RAW );
stmt.registerOutParameter( 4, OracleTypes.RAW );
stmt.registerOutParameter( 5, OracleTypes.RAW );
stmt.registerOutParameter( 6, OracleTypes.RAW );
stmt.registerOutParameter( 7, OracleTypes.NUMBER );
stmt.registerOutParameter( 8, OracleTypes.VARCHAR );
stmt.registerOutParameter( 10, OracleTypes.RAW );
stmt.setString( 1, clientPubModulus );
stmt.setString( 2, clientPubExponent );
stmt.setString( 9, "Tuesday" );
stmt.setNull(   3, OracleTypes.RAW );
stmt.setNull(   4, OracleTypes.RAW );
stmt.setNull(   5, OracleTypes.RAW );
stmt.setNull(   6, OracleTypes.RAW );
stmt.setInt(    7, 0 );
stmt.setNull(   8, OracleTypes.VARCHAR );
```

```
stmt.setNull(  10, OracleTypes.RAW );
stmt.executeUpdate();
errNo = stmt.getInt( 7 );
if( errNo != 0 ) {
    errMsg = stmt.getString( 8 );
    System.out.println( "Oracle error " + errNo +
        ", " + errMsg );
    System.out.println( (stmt.getRAW( 10 )).toString() );
} else {
    //mCryptSessionSecretDESSalt = stmt.getRAW( 3 );
    //mCryptSessionSecretDESIterationCount = stmt.getRAW( 4 );
    //mCryptSessionSecretDESAlgorithm = stmt.getRAW( 5 );
    //mCryptSessionSecretDESPassPhrase = stmt.getRAW( 6 );
    cryptData = stmt.getRAW( 10 );

    System.out.println( getDecryptData( cryptData,

        mCryptSessionSecretDESPassPhrase,
        mCryptSessionSecretDESAlgorithm, mCryptSessionSecretDESSalt,
        mCryptSessionSecretDESIterationCount ) );
}
if( null != stmt ) stmt.close();
```

Along with key exchange, we send the string "Tuesday" to the `p get des crypt test data` procedure to be encrypted on Oracle database using the secret password key. So, after executing the `Statement`, we retrieve the encrypted data, and then decrypt the data by calling `getDecryptData()` locally on the client and print out the decrypted `String`. Notice that the `getDecryptData()` method takes all the encrypted secret password key artifacts.

If the secret password key has not yet been built, then `getDecryptData()` calls `makeSessionSecretDESKey()`. When we call `getDecryptData()` we pass sufficient parameters, the artifacts of the secret password key, to build the key; but if it has already been built, we do not repeat that effort. We may call `getDecryptData()` multiple times for multiple pieces of encrypted data, but the effort to build the secret password key will only be undertaken once.

Testing Oracle Database Encrypt and Local Decrypt Data

The next test in Listing 6-24 is even more concise, if unrealistic. We will call the temporary function, `f get crypt data` to get encrypted data representing a clear text `String`, "Wednesday". We will get the encrypted data `RAW` back from the statement and call the `getDecryptData()` method locally to decrypt it, printing the result.

Listing 6-24. *Get Crypt Data, from `main()`*

```
stmt = ( OracleCallableStatement )conn.prepareCall(
    "{? = call app sec pkg.f_get_crypt_data(?) }" );
stmt.registerOutParameter( 1, OracleTypes.RAW );
stmt.setString( 2, "Wednesday" );
stmt.executeUpdate();
cryptData = stmt.getRAW( 1 );

System.out.println( getDecryptData( cryptData,

    mCryptSessionSecretDESPassPhrase,
```

```
        mCryptSessionSecretDESAlgorithm, mCryptSessionSecretDESSalt,
        mCryptSessionSecretDESIterationCount ) );
    if( null != stmt ) stmt.close();
```

Normally we wouldn't call Oracle database to encrypt clear text data from the client, and then decrypt it at the client to obtain the clear text. Nevertheless, that is precisely what we did here.

Sending Encrypted Data to Oracle

In our last test in Listing 6-25, we will encrypt clear text data, "Thursday" on the client. We encrypt it with the copy of the DES secret password key, based on the artifacts we got from Oracle. Then we submit the encrypted data to Oracle database by calling the `f get decrypt data` function. Oracle database will decrypt the data using the original secret password key, and then our client will read the clear text `String` that is returned as parameter 1. We print that result and close the `Statement`.

Listing 6-25. *Get Decrypt Data, from* `main()`

```
cryptData = getCryptData( "Thursday" );
stmt = ( OracleCallableStatement )conn.prepareCall(
    "{? = call app sec pkg.f_get_decrypt_data(?) }" );
stmt.registerOutParameter( 1, OracleTypes.VARCHAR );
stmt.setRAW( 2, cryptData );
stmt.executeUpdate();
System.out.println( stmt.getString( 1 ) );
if( null != stmt ) stmt.close();
```

This is another unlikely scenario where we are calling an Oracle function to decrypt data for use on the client. Don't worry about exposing unneeded functionality to the client; we will be sorting this out as we continue. In fact, we already have! Our client applications will not be connecting as `appsec` user, and only `appsec` will be able to execute procedures and functions in the `app sec pkg` package.

Testing Our Secure Client/Server Data Transmission

In a Command Prompt, Change directories to *Chapter6*. Compile the code with this command:

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

Again, if you have any problems, refer to Chapter 3 for directions on compiling at the command prompt and setting your environment `CLASSPATH` to include *ojdbc6.jar*. Then run the code from that same directory with this command:

```
java orajavsec.OracleJavaSecure
```

The following six lines will be printed as a result:

```
Monday
PBEWithSHA1AndDESede
PBEWithMD5AndDES
Tuesday
Wednesday
Thursday
```

The first two lines show the same client-only tests that we performed earlier in this chapter. After that, because `testingOnServer` is `true`, the `main()` method continues to get the DES secret password key from Oracle. Then we call `f show algorithm` and display the negotiated algorithm at the command prompt, which will likely be `PBEWithMD5AndDES`.

After that, we send the string "Tuesday" to Oracle database for encryption by `p get des crypt test data` procedure. We read the encrypted data, decrypt it with a copy of the secret password key, and display it, "Tuesday".

We also call the function `f get crypt data` directly, handing it the clear text string "Wednesday". We read the encrypted data returned by the function, decrypt it again with the copy of the secret password key and display it, "Wednesday". By this time, it is obvious that the session of our Oracle connection is retaining and reusing the same RSA and DES keys for multiple queries.

Finally, we encrypt the string, "Thursday" on the client and submit it to the Oracle function `f get decrypt data`. Oracle database decrypts the data with the original secret password key. We read the returned clear text string and print it, "Thursday."

Chapter Review

We find ourselves at the end of another chapter. Looking back, let's see what ground we've covered.

- We learned about DES secret password encryption. In particular, we learned about the various artifacts that make up the secret password key: passphrase, salt, iteration count, and the algorithm. We also devised a way to observe and report and work around a bug in JCE, even coding to accommodate an eventual JVM upgrade that will not have the bug.
- We found out how to generate the secret password key on Oracle, do key exchange, then build an identical key on the client. With identical keys (a shared password), we can exchange encrypted data. We also saw how to use the RSA public key to encrypt the DES key so that we can exchange it between Oracle database and the client, but still keep it secret.
- We made extensive use of the `SecureRandom` class in order to generate a random passphrase, salt, and iteration count for each session.
- We explored Oracle packages, something we will depend on for organization and security.

Figures 6-1 and 6-2 illustrate the secret password encryption processes we have covered in this chapter. These processes will be used throughout the remainder of this book. In Figure 6-1, at the top, you will see that we refer to block [**A**] of Figure 5-1—that is where we saw the standard process of generating an RSA public/private key pair on the client. We pass the public key exponent and modulus to Oracle database when we call `p get shared pass phrase`. You can see in Figure 6-1, on the far right, that an equivalent public key is built on Oracle database, represented by the key image labeled RSA. Another key image on the far right, labeled DES depicts the shared password key that we create on the Oracle database.

Each of the artifacts of the shared password key is encrypted on the Oracle database with the public key and returned to the client. Each encrypted artifact is placed in one of the `OUT` parameters of the `p get shared pass phrase` procedure. We have outlined this whole process and labeled it as block [**B**] in Figure 6-1. This is our standard process for exchanging the shared password key, and we will refer to it in future Figures.

At the bottom of Figure 6-1, we illustrate how the client calls methods to decrypt the secret password key artifacts and to build an equivalent secret password key. The key image at the bottom represents the secret password key on the client.

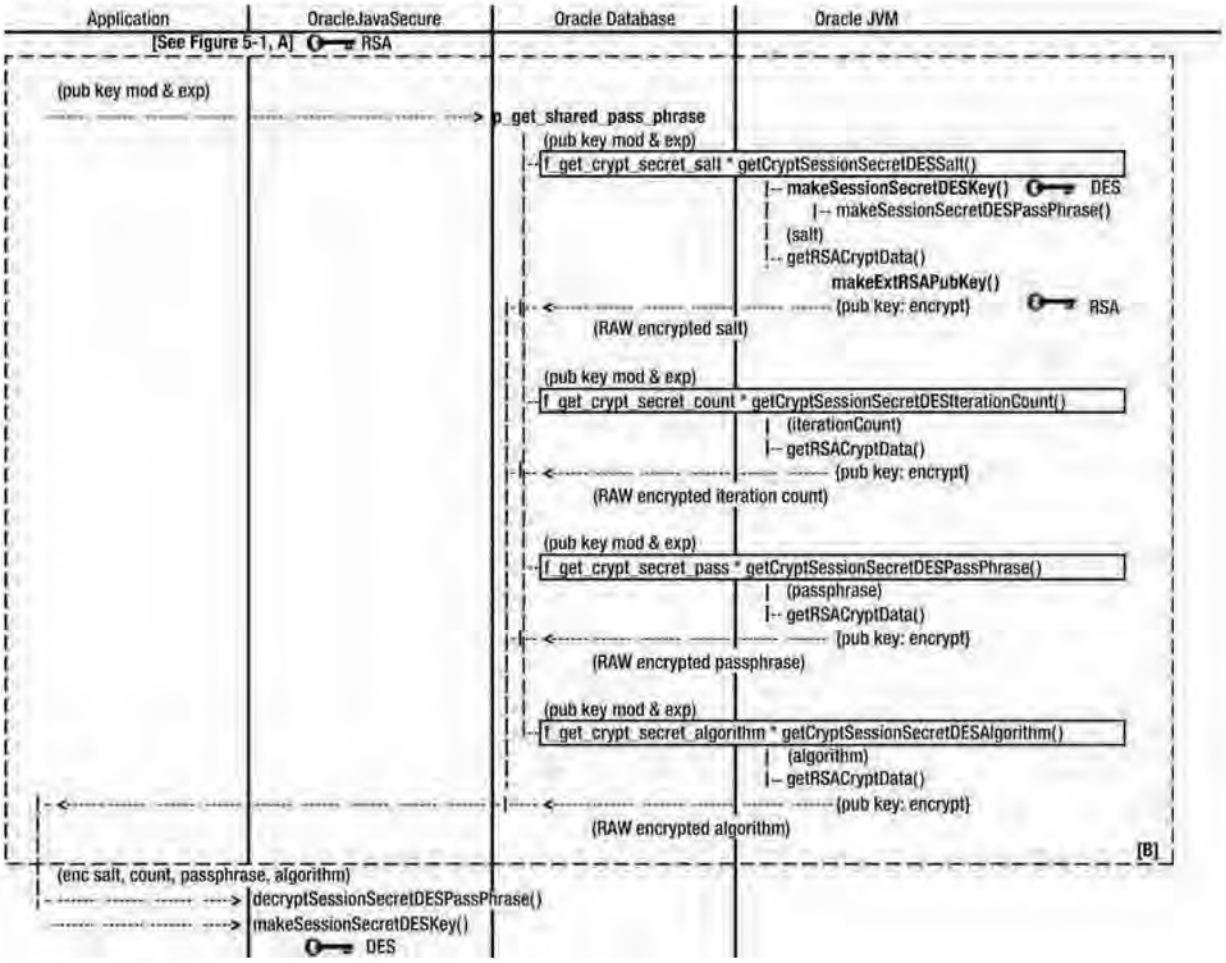


Figure 6-1. *Key exchange*

In Figure 6-2 you can see an illustration of the example processes for exchanging encrypted data between the client and Oracle database. In the top one third of Figure 6-2, you see the process of doing key exchange and encrypting data on the Oracle database. That encrypted data is returned to the client in the `OUT` parameters of the `p get des crypt test data` procedure. Here we refer to block **[B]** of Figure 6-1, where we saw the secret password key being built on Oracle database and returned to the client. In Figure 6-2, in addition to building the secret password key, we use that key to encrypt data that is returned to the client in encrypted form.

In the middle section of Figure 6-2, by a call to the `getDecryptData()` method (version **A** of two methods with that name), we see an illustration of how the client decrypts data coming from Oracle database. We outline this section of Figure 6-2 and label it as block **[C]**—it illustrates a standard process for decrypting data. You can see that an equivalent secret password key, labeled DES, is built on the client (as needed) during the decryption process.

The last section of Figure 6-2, at the bottom shows the mirror image of this process. In that illustration, the client uses the equivalent secret password key that has already been built to encrypt

data. The encrypted data is then sent to the Oracle database by a call to the `f get decrypt data` function. The Oracle database then decrypts the data using the original secret password key. In our example code, that decrypted, clear-text data is returned to the client for display.

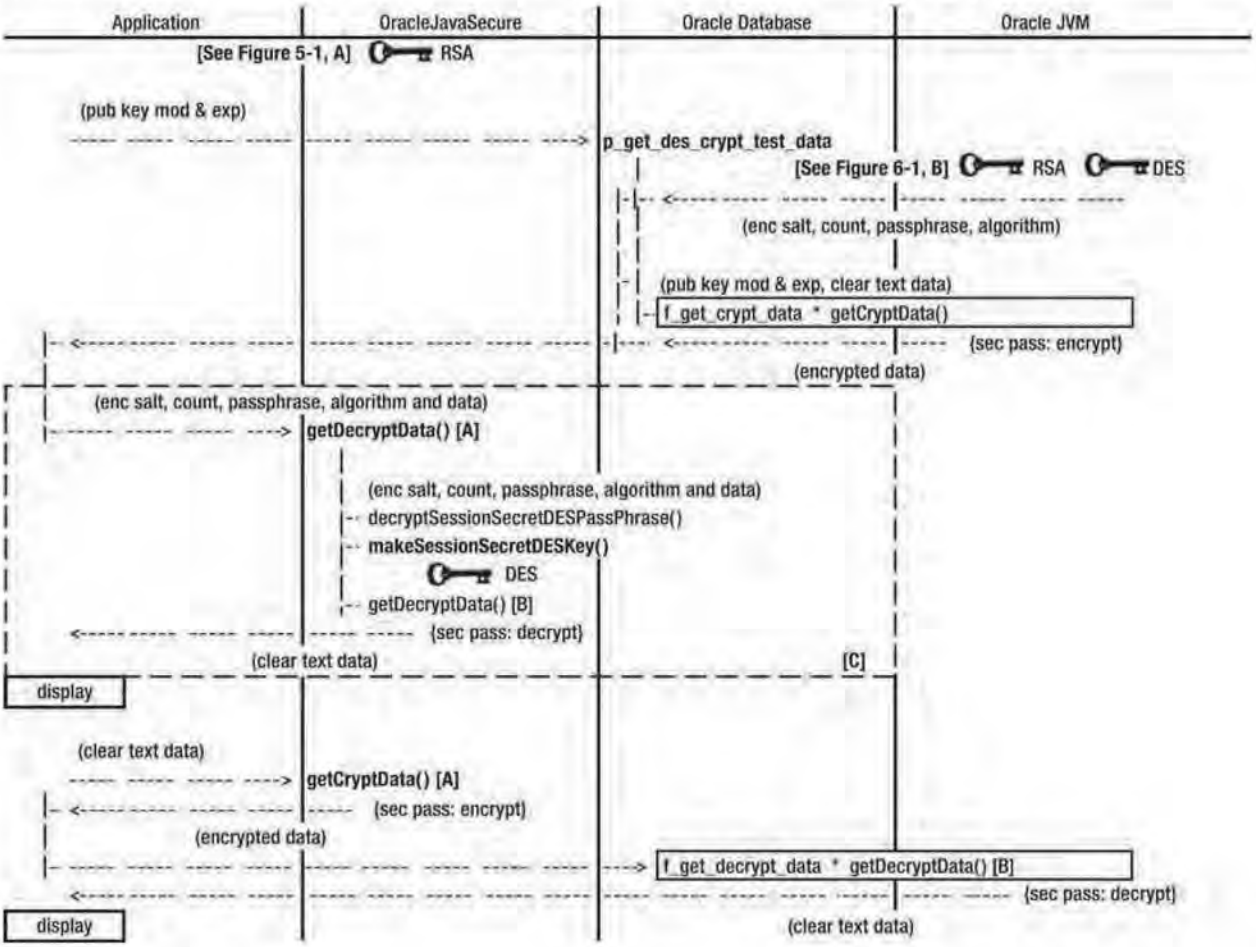


Figure 6-2. *Encrypted data exchange example*

CHAPTER 7

Data Encryption in Transit

In Chapter 6, we laid the foundation for data encryption between Oracle database and a Java client. We proved that we can exchange keys securely and then send encrypted data back and forth, successfully decrypting the data at the recipient, and at the Oracle database.

In this chapter, we will complete the foundation for encryption, which we will continue to build in the application security, `appsec` Oracle schema. Then we will take on the role of an application developer, using the `appsec` structures to secure access to our data. Specifically, we will secure access to data in the `HR` example schema.

Consider the last chapter and observe that we have been both building and testing our application security structures and code as the `appsec` user. It is not the intent to have every application run as `appsec`. Rather, we will allow each application that needs our security to execute our security functions, and we will be demonstrating that in this chapter.

Also, we need to provide developers access to our Java structures to include with their desktop applications. We will discuss that at the end of this chapter.

Security Administrator Activities

Our security administrator, `secadm`, needs to provide a few more permissions. Some of the permissions are system privileges, and some are grants to packages in the `appsec` and `HR` schemas.

■ **Note** You can find a script of the following commands in the file named *Chapter7/SecAdm.sql*.

Connect to Oracle database as SECADM user, and acquire the secure application role, `secadm role`.

```
EXECUTE sys.p check secadm access;
```

We are going to be creating a table in the `appsec` schema for logging errors. We are also going to create a trigger associated with that table. Our trigger is like a procedure that runs when certain events occur—in our case, our trigger will run when a record is inserted into our table.

We want to have a central table for application errors, because the error messages will be returned to, potentially, dozens or hundreds of applications. How will we, as the application security administrator, get reports from all these sources? If our application developers are conscientious, they will let us know what problems they are seeing, but we are not going to count on that happening. We will watch for errors from our remote listening post—the error table.

Granting More System Privileges to the Application Security User

In order to succeed at setting up the error logging table, we need to allow appsec to store data in a tablespace. The default tablespace is "USERS," and that will suffice for us. We need to specify how much space appsec may use, a *quota*. We'll start out permitting two megabytes of space. Execute the following as appsec:

```
ALTER USER appsec DEFAULT TABLESPACE USERS QUOTA 2M ON USERS;
```

Additionally, for appsec to be able to create a trigger, we need to grant the CREATE TRIGGER system privilege. We will grant it to her non-default role (she'll only need it occasionally):

```
GRANT CREATE TRIGGER TO appsec role;
```

Permitting Users to Execute Packages in Other Schemas

We want HR to execute the appsec security structures. We'd like to create a role to which we might grant execute on a package, and then grant that role to whomever needs it. However, let's examine why this approach does not always work. Consider the following statements, which you should not execute:

```
--CREATE ROLE appsec user role NOT IDENTIFIED;
--GRANT EXECUTE ON appsec.app sec pkg TO appsec user role;
--GRANT appsec user role TO hr;
```

Specifically, the approach illustrated here does not work when procedures, functions, and packages call procedures, functions, and packages in other schemas. To get a bit ahead of ourselves, we will be creating procedures in the HR schema that we want to have execute the app sec pkg package (you know, we want HR to call functions to encrypt data).

The problem is that the HR procedures, functions, and packages cannot gain privileges from a role. This is a restriction (based on the dependency model) designed to keep the HR procedures from getting invalidated every time we logout or set role. We remedy the restriction by granting execute on the app sec pkg package directly to HR user. For example, execute the following code:

```
GRANT EXECUTE ON appsec.app sec pkg TO hr;
```

By way of stark contrast, our application user, appusr, and other application users will be calling the procedures, functions, and packages in the HR schema directly, as needed. We do not imagine appusr calling our procedures from their own procedures. So we can grant access to our (yet to be configured) HR Security Package, hr sec pkg, to a role that appusr has, the hrview role. The following is the GRANT statement, which we will execute later after creating the hr.hr sec pkg package:

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

Later we will execute a script named HR.sql. This script creates the hr.hr_sec_pkg package, and also executes the preceding GRANT statement.

Application Security User Activities

We are going to create an error log table, and an insert trigger, and we will also be adding procedures to do logging to app sec pkg package.

■ **Note** You can find a script of the following commands in the file named *Chapter7/AppSec.sql*.

Connect to Oracle database as appsec user, and set your role to the non-default role, appsec role:

```
SET ROLE appsec role;
```

Creating a Table for Error Logging

Next, create a table for error logging. Probably, you want to pull out your DBA skills or get a DBA to help you define this table, setting its performance parameters and estimating initial storage and growth plans, none of which is defined here. Execute the code in Listing 7-1 to create the table using the defaults.

Listing 7-1. *Create the Application Security Error Log Table,* `t appsec errors`

```
CREATE TABLE appsec.t appsec errors (
    err no     NUMBER,
    err txt    VARCHAR2(2000),
    msg txt    VARCHAR2(4000) DEFAULT NULL,
    update ts  DATE DEFAULT SYSDATE
);
```

We are going to capture the Oracle error number, `err no`, and text, `err txt`, and provide ourselves with another field, `msg txt`, for helpful information (e.g., method name or stack trace). We will also capture the time of the error, `update ts`, which helps in two ways: first, we want to know when things happened, or what is happening; second, we want to throw away log records when they are too old to be helpful.

Note that the last two column definitions in the listing specify default values using the keyword `DEFAULT`. To insert a record, you only need to insert the first two fields. The third will default to `NULL`, and the fourth will default to the current date and time on the Oracle database, `SYSDATE`. In fact, we do not want to insert a date in `UPDATE TS`. We want to accept the default. Because appsec is the only one who will be entering data in this table (this is an error log for application security use, not for general use), we do not need to enforce the default `UPDATE TS`.

In order to accomplish our sorting and selection of records by date, we are going to set up an index on the `UPDATE TS` column. Execute the code in Listing 7-2 to create the index.

Listing 7-2. *Index for the Application Security Error Log Table,* `t appsec errors`

```
CREATE INDEX i appsec errors00 ON appsec.t appsec errors (
       update ts
);
```

Indexes, once created, are maintained automatically as you insert or update rows. Also, they are automatically used when you select records from the table. Which index is used is a logical choice made by Oracle database that can be overridden by a hint, if desired. One thing to be aware of is that a select query that does not mention the leading column of an index will not benefit directly from the index. For example, if we selected all rows from the `t appsec errors` table where the `msg txt` column contains the string "Exception," we would not directly use the index we just created. If we created an index on columns (`update ts, msg txt`), that index would not benefit our query directly either. To get the direct

benefits of an index when we select from `msg txt`, we would want to create an index mentioning `msg txt` as the first column, for example `( msg txt, update ts )`.

The Oracle database optimizer may actually use any index to improve the performance of a query by doing a *skip scan*. A skip scan can improve performance in simple queries, potentially reducing the reliance on full-table scans (those cause very bad performance). You can also use a skip scan hint to explicitly name an index for use. It may take some trial and error and a bit of engineering to get the best performance from a skip scan hint. Here is an example, taken from the Oracle documentation. The comment section (/* */) with the plus symbol and hint name serve as a hint to the optimizer.

```
SELECT /*+ INDEX_SS(e emp_name_ix) */ last name
    FROM employees e
    WHERE first name = 'Steven';
```

Notice that the skip scan hint, `INDEX SS` directs the optimizer to use the index `emp name ix` for our query. Even though we're selecting records where the `first name` column is 'Steven,' we ask to benefit from the index on `last name, first name`. For a better understanding of skip search optimization, I recommend doing an Internet search for examples.

You might want to sort or select from `t appsec errors` table based on different columns, but unless there is a frequent query that requires this column to be ordered, you will not need an index. Because this table is really only intended for troubleshooting after an error, we don't expect to have other indexes—we will always select the most recent records (based on `update ts`).

We will create a view of the table that we are prepared to grant select on; however, we don't have anyone in mind at present who may need to view it. Perhaps down the road we will have a savvy app developer who wants to help debug his application's use of the `appsec` packages. We may arrange for her to select from our view. Execute the following:

```
CREATE OR REPLACE VIEW appsec.v appsec errors AS SELECT * FROM appsec.t appsec errors;
```

Creating a Table for Managing Our Error Log Table

Remember, we only gave our application security, `appsec`, user two megabytes of space in the `USERS` tablespace. It would be inconsiderate, if not negligent, to create a table, especially a log table, without any provision for regular cleaning and maintenance.

Now to let you in on a secret: we are building a robot. Not a mechanical contraption to fetch our coffee, but a software sentinel to help us manage the error log table, especially when we are not looking. We are going to do automatically remove old records from our table with a trigger that runs whenever we insert a record into our table.

The tricky thing is, there is some effort involved in managing our table, so we want to minimize how frequently the management tasks occur. In fact, we only want to manage our error log table once a day. We also won't mind knowing when the table was last managed. The best way to accomplish both those goals is to create another table to store the date whenever our error log table is managed. Execute the code in Listing 7-3 to create the error log maintenance table.

Listing 7-3. *Creating the Application Security Error Log Table and Index,* `t appsec errors maint`

```
CREATE TABLE appsec.t appsec errors maint (
    update ts DATE DEFAULT SYSDATE
);
CREATE UNIQUE INDEX i appsec errors maintOO ON appsec.t appsec errors maint (
       update ts );
```

Once again, selecting by date is important for performance, so we will create an index on the UPDATE TS column, the only column.

This time, we made it a UNIQUE index, meaning that we will only have one entry with a specific timestamp. On our error log table, the index was not UNIQUE, because we may have multiple errors and entries in the table at the exact same time.

The t appsec errors maint table is for internal use only, so we won't create a view and don't anticipate ever granting privileges on the table.

Creating an Error Log Management Procedure

Our table management task is instigated by a trigger, but before we can define the trigger, we need to define our procedure that accomplishes the management task. Our management procedure will be named p appsec errors janitor, and it has no parameters.

We want it to run independently; therefore, we define it with the modifier PRAGMA AUTONOMOUS TRANSACTION. This allows the procedure to do inserts and deletes and commit the changes, even if the program that called this transaction does not commit. Without this modifier, if we issue a commit here, we are asking Oracle database to commit every update, insert, or delete we've made in the current session. When we're dealing with an error, we specifically want to avoid committing anything except the insert of an error message in our log and cleanup of old entries. Execute the script in Listing 7-4 to create the procedure.

Listing 7-4. *Procedure to Manage to Error Log Table, p appsec errors janitor*

```
CREATE OR REPLACE PROCEDURE appsec.p appsec errors janitor
AS
    PRAGMA AUTONOMOUS TRANSACTION;
    m err no NUMBER;
    m err txt VARCHAR2(2000);
BEGIN
    INSERT INTO t appsec errors maint ( update ts ) VALUES ( SYSDATE );
    COMMIT;
    -- Remove error log entries over 45 days old
    DELETE FROM t appsec errors WHERE update ts < ( SYSDATE - 45 );
    COMMIT;
    INSERT INTO t appsec errors
        ( err no, err txt, msg txt ) VALUES
        ( 0, 'No Error', 'Success managing log file by Janitor' );
    COMMIT;
EXCEPTION
    WHEN OTHERS
    THEN
        m err no := SQLCODE;
        m err txt := SQLERRM;
        INSERT INTO t appsec errors
            ( err no, err txt, msg txt ) VALUES
            ( m err no, m err txt, 'Error managing log file by Janitor' );
        COMMIT;
END;
/
```

Our first step (after the BEGIN header) is to insert the current date in our management table. That will prevent others from also trying to manage the error log table. We commit it, which is only allowed here because we are an autonomous transaction.

The second step is to delete records in the error log that are over 45 days old. Notice that we do some date arithmetic involving SYSDATE, in this case SYSDATE - 45, which is equivalent to 45 days ago. We will use similar date arithmetic in our trigger. We also commit this deletion.

The last thing under the BEGIN header is to insert a "Success" message in the error log, and commit it. Why not? That seems like a good place.

As in our other procedures we've seen so far, we are going to catch errors. In this case, we will insert the error into our error log table (once again, why not? It'll be nice to have all our troubleshooting messages in one place.) And commit it.

Creating a Trigger to Maintain the Error Log Table

The maintenance procedure we defined above will work every time you call it, and however you call it. You could hire someone to manually run that procedure once a day. Oracle database has a scheduler (the DBMS SCHEDULER PL/SQL package or the older DBMS JOB PL/SQL package) that you could alternatively use to run it once a day.

Instead, we are going to make the table self-governing by adding a trigger. A trigger has a lot of similarities to a procedure, so it is consistent with the syntax we've been discussing. Execute the code in Listing 7-5 to create and enable a trigger on the t appsec errors table that will run after each row (log entry) is inserted in the table.

Listing 7-5. *Insert Trigger on Error Log Table,* `t appsec errors iar`

```
CREATE OR REPLACE TRIGGER appsec.t appsec errors iar
    AFTER INSERT ON t appsec errors FOR EACH ROW
DECLARE
    m log maint dt DATE;
BEGIN
    SELECT MAX( update_ts ) INTO m_log_maint_dt FROM t appsec errors maint;
    -- Whenever T APPSEC ERRORS MAINT is empty, M LOG MAINT DT is null
    IF( ( m log maint dt IS NULL ) OR
        ( m log maint dt < ( SYSDATE - 1 ) ) )
    THEN
        p appsec errors janitor;
    END IF;
END;
/
ALTER TRIGGER appsec.t appsec errors iar ENABLE;
```

This trigger runs after each insert, AFTER INSERT; however, we only want our procedure to run once a day. To accomplish this, we get the MAX(update ts) of the last time our procedure was run from the t appsec errors maint table, and store that date in m log maint dt. (Note this example of SELECT INTO syntax—selecting a value into a variable.) Then we check to see if m log maint dt is NULL (whenever t appsec errors maint table is empty) or m log maint dt is earlier than 24 hours ago (< SYSDATE – 1). If it is, then we run our procedure, p appsec errors janitor.

Testing the Trigger

While you are connected to Oracle database as appsec user, you can test the trigger. First execute the following lines to insert an error log entry and commit it:

```
INSERT INTO appsec.v appsec errors (err no, err txt ) VALUES (1, 'DAVE' );
COMMIT;
```

Note that our autonomous procedure can only deal with data that exists independently. Our inserts and updates do not exist in the database independently until we COMMIT the data.

Note also that we are depending on the default values for msg txt and update ts—those columns are not part of our insert statement.

Query each of our tables, the error logs and the maintenance record, and observe that our previous insert succeeded and that the Janitor procedure ran. Here's an example:

```
SELECT * FROM appsec.v appsec errors ORDER BY update ts;
SELECT * FROM appsec.t appsec errors maint ORDER BY update ts;
```

Now insert an error log entry that pretends to be 60 days old (note the arithmetic with SYSDATE):

```
INSERT INTO appsec.v appsec errors (err no, err txt, msg txt, update ts)
    VALUES (2, 'DAVE', 'NONE', SYSDATE - 60 );
COMMIT;
```

Again query each of our tables to assure that our insert worked and that our janitor procedure did not run again (because it has already run on this day):

```
SELECT * FROM appsec.v appsec errors ORDER BY update ts;
SELECT * FROM appsec.t appsec errors maint ORDER BY update ts;
```

Now change the data of our last janitor maintenance run date to yesterday (actually, 24 hours ago), and assure that the change was effective (if you have more than one entry in this table, this UPDATE won't work. The index on UPDATE TS in this table is a UNIQUE index):

```
UPDATE appsec.t appsec errors maint SET update ts = SYSDATE-1;
COMMIT;
SELECT * FROM appsec.t appsec errors maint ORDER BY update ts;
```

And submit another record from today (default, SYSDATE):

```
INSERT INTO appsec.v appsec errors (err no, err txt ) VALUES (3, 'DAVE' );
COMMIT;
```

For the last time in these tests, query each of our tables to assure that our insert worked and that our janitor procedure ran a second time, and that the mock-old record (the one with err no = 2) was deleted:

```
SELECT * FROM appsec.v appsec errors ORDER BY update ts;
SELECT * FROM appsec.t appsec errors maint ORDER BY update ts;
```

Updating the Application Security Package

In Chapter 6, we had two procedures (p get shared passphrase and p get des crypt test data) and one function (f show algorithm) that we described as being "temporary". They were used for testing only in Chapter 6, and we will remove them from app sec pkg in this chapter. The remainder of the

functions in app sec pkg will remain and have not been changed. Look at the file, *Chapter7/AppSec.sql* to see the full listing. We have one new procedure that we introduce in this chapter: p log error.

Creating an Error Logging Procedure

The p log error procedure takes a NUMBER and one or two VARCHAR2 (text) parameters. The err txt field is limited to 2,000 characters, but a VARCHAR2 column may have up to 4000 characters; so we truncate the m err txt parameter to 2,000 characters, if needed, to fit our err txt column.

Note that this procedure (the package), and the table being updated are in the appsec schema, but the one calling this procedure may be an application in another schema (like HR). We have already granted execute on app sec pkg package to the HR user, and we need to grant execute to any other application user who needs our Application Security processes.

If you will, think back to when we defined the t appsec errors table. Recall that we set the msg txt and update ts columns to be nullable, and to have default values (NULL and SYSDATE). That allowed us to do data inserts by just providing data elements for the first two columns. We could insert data without even mentioning the last two columns. In fact, we said that we don't want to insert a value for update ts; but rather, allow the Oracle database to assign the current default value of SYSDATE.

Well, now we are creating a procedure (shown in Listing 7-6) for various applications to call in order to insert error records into our table, and the procedure accounts for those default values. First of all, the procedure doesn't accept a value for update ts; rather, the default SYSDATE will be used. Second, the value for msg txt has a default value of NULL so that application users can call this procedure with or without a value for msg txt.

Listing 7-6. *Procedure to Insert Log Entries, p log error*

```
PROCEDURE p log error( m err no NUMBER, m err txt VARCHAR2,
    m msg txt VARCHAR2 DEFAULT NULL )
IS
    l err txt VARCHAR2(2000);
BEGIN
    l err txt := RTRIM( SUBSTR( m_err_txt, 1, 2000 ) );
    INSERT INTO v appsec errors ( err no, err txt, msg txt )
        VALUES ( m err no, l err txt, m msg txt );
    COMMIT;
END p log error;
```

We use the substring function, SUBSTR, to get only the first 2,000 characters of the error text. Then we use the right trim function, RTRIM to remove any spaces at the right end of the remaining text. If m err txt is NULL, SUBSTR returns a NULL, and RTRIM returns a NULL.

At the end of the p log error procedure, we simply insert the error data into our error log table and COMMIT.

Executing Package Specification and Body

Execute the two blocks in the file named *Chapter7/AppSec.sql* to replace the app sec pkg package specification and body. You can see that both those blocks begin with the command CREATE OR REPLACE. Because we already have a package named app sec pkg, this command will replace it. The absolute best thing about this command is that we can execute it on a running Oracle database, and the applications that use the package will not fail. That is, if the package specification does not need to change. Consider another option: if we had to DROP and then separately CREATE these structures, we would have to wait

until the Oracle database was off-line, or at least until the dependent applications were not running; otherwise, applications would fail during the interim between the `DROP` and the `CREATE`.

Methods for Using and Testing Encryption in Transit

Our working model will not be tested from the `main()` method of `OracleJavaSecure`; rather, we are going to demonstrate how we come in as a separate application and use the structures of our Application Security package, `app sec pkg`. We are going to add two more methods to the `OracleJavaSecure` class: one just for testing, `resetKeys()`; and another to prepare the client to encrypt data for updates / inserts to Oracle database, `makeDESKey()`.

We want the ability to do data updates from our client application with a minimum of effort. The least effort requires these steps:

1. Generate RSA keys on client and pass public key to Oracle.
2. Generate DES secret password key on Oracle database, encrypt the artifacts with RSA public key, and pass back to the client.
3. Build a copy of the DES key on the client.
4. Encrypt data with DES key and send to Oracle database for decrypt and update.

We have already demonstrated an Oracle procedure, `p get shared passphrase`, which allowed us to combine steps 1 and 2 into a single step. However, step 4 requires a second Oracle statement. Therefore, we are going to have to make at least two calls to Oracle database to do the first update. Within the same Oracle session, we can do additional updates, each in a single call. We only need to do the combined steps 1, 2 and 3 once (key exchange); and then after the keys have been established, we can do as many updates and inserts as we'd like using existing keys.

Method to Build the Secret Password Key

In Chapter 6, we used the `p get shared passphrase` Oracle procedure to get all the DES secret password key artifacts to the client; however, we didn't build the secret password key until we received encrypted data from Oracle database that we wanted to decrypt on the client.

In this chapter, we are going to need the DES secret key even without data to decrypt. We are going to do data encryption on the client and send it to Oracle database as an independent task. So, we need a Java method that independently builds the secret password key. Listing 7-7 shows the code for that method.

Listing 7-7. *Method to Call to Build Secret Password Key, `makeDESKey()`*

```
public static final void makeDESKey(
    RAW cryptSecretDESPassPhrase, RAW cryptSecretDESAlgorithm,
    RAW cryptSecretDESSalt, RAW cryptSecretDESIterationCount )
{
    try {
        decryptSessionSecretDESPassPhrase( cryptSecretDESPassPhrase,
            cryptSecretDESAlgorithm, cryptSecretDESSalt,
            cryptSecretDESIterationCount );
        makeSessionSecretDESKey();
    } catch( Exception x ) {
        x.printStackTrace();
    }
}
```

Within the `try` block is most of the body of our previous `getDecryptData()` method, sans the call to actually decrypt data. This provides us an opportunity to do some refactoring, improving the design of our code. Because our new method does most of what we did in `getDecryptData()`, let's rewrite `getDecryptData()` to call the new method, as in Listing 7-8.

Listing 7-8. *Decrypt Data with the Secret Password Key, `getDecryptData()`*

```
public static final String getDecryptData( RAW cryptData,
    RAW cryptSecretDESPassPhrase, RAW cryptSecretDESAlgorithm,
    RAW cryptSecretDESSalt, RAW cryptSecretDESIterationCount )
{
    String rtrnString = "getDecryptData() A failed";
    try {
        if( ( null == sessionSecretDESKey ) || testAsClientAndServer ) {
             makeDESKey( cryptSecretDESPassPhrase, cryptSecretDESAlgorithm,
                 cryptSecretDESSalt, cryptSecretDESIterationCount );
        }
        rtrnString = getDecryptData( cryptData );
    } catch( Exception x ) {
        x.printStackTrace();
    }
    return rtrnString;
}
```

The bold text, our call to `makeDESKey()`, is where we previously had the code that we have moved into the body of `makeDESKey()`.

Temporary Method to Reset All Keys

The second method we are adding to `OracleJavaSecure` is `resetKeys()`. The `resetKeys()` method is only for testing in this chapter (however, we will resurrect it in Chapter 10). Later we will describe several test scenarios, one of which will emulate starting a new connection/session on the client (by running this method) and trying to use existing keys on Oracle database. This scenario will fail, but we will do the test just to demonstrate that scenario.

In `resetKeys()`, Listing 7-9, we set back to `null` those static members that we set to `null` initially. Recall that they were `null` in order to facilitate testing those variables for `null` and/or for testing comparison to other members. We needed to set them to `null` initially in order to compile past the "may not have been initialized" error message.

We also reset the value of `sessionSecretDESAlgorithm` to its pre-negotiated value.

Listing 7-9. *Reset all Keys, `resetKeys()`*

```
public static final void resetKeys() {
    locRSAPubMod = null;
    saveExtRSAPubMod = null;
    extRSAPubKey = null;
    sessionSecretDESPassPhraseChars = null;
    sessionSecretDESKey = null;
    sessionSecretDESAlgorithm = "PBEWithSHA1AndDESede";
}
```

Loading Updated OracleJavaSecure Class into Oracle

Connect or remain connected to Oracle database as application security, `appsec`, user with the non-default role, `appsec role` and copy/paste the code from the file *Chapter7\orajavsec\OracleJavaSecure.java* into your Oracle client. Uncomment the first line then run the script to replace our Java class in Oracle database.

```
CREATE OR REPLACE AND RESOLVE JAVA SOURCE NAMED appsec."orajavsec/OracleJavaSecure" AS
```

Security Structures for the HR User

Our working model of application encryption will consist of reading data from the `HR` schema, with sensitive columns being encrypted as they are transmitted across the network. Part of the responsibility falls to the *application* developers who must assure that sensitive data is only available to the client in encrypted form. Our application security schema, `appsec`, can provide the tools, but our application developers, like `HR`, will need to implement them.

Let's explore what `HR` does to encrypt his data first. Then we'll look at what the Application Security manager can provide to all application developers as a template for implementing this.

Exploring Privileges That Enable HR Tasks

`HR`, as provided by Oracle, is a sample schema and already has a variety of system privileges. `HR` has the default role, `RESOURCE`, and via that role, has this list of privileges:

```
CREATE SEQUENCE,
CREATE TRIGGER,
CREATE CLUSTER,
CREATE PROCEDURE,
CREATE TYPE,
CREATE OPERATOR,
CREATE TABLE,
CREATE INDEXTYPE
```

All application schemas that intend to implement our application security will need the CREATE PROCEDURE system privilege.

Recall that to have access to the application security structures, we also need to grant each application schema the object privilege to execute the app sec pkg package, like we did (already as secadm) for HR:

```
GRANT EXECUTE ON appsec.app sec pkg TO hr;
```

Creating the HR Security Package

HR will have a package of his own procedures and functions that provide access to HR tables, but return any sensitive columns in encrypted form only. Let's examine the package and then create it at the end.

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

```
CREATE OR REPLACE PACKAGE hr.hr sec pkg IS
    TYPE RESULTSET_TYPE IS REF CURSOR;
```

Within the Specification of our package, we will define a TYPE. We will name it RESULTSET TYPE, and it will represent a CURSOR, also known as a ResultSet in Java. When we call procedures to get our encrypted HR data, we are going to hand back from Oracle database, a number of OUT parameters. Many of the OUT parameters will be artifacts of our secret password key, as we have seen, and one might also be a RESULTSET TYPE, which will hold *multiple rows of encrypted data.*

Selecting Sensitive Data Columns from EMPLOYEES

The code in Listing 7-10 is the body of an Oracle procedure named p select employees sensitive. You should be very familiar with this format. The list of parameters and the code used to set the secret password key artifacts look like what we've seen before. We do have an OUT parameter named resultset out that will hold a RESULTSET TYPE (rows of data):

Listing 7-10. *Procedure to Select Sensitive Data from Employees Table, p select employees sensitive*

```
    PROCEDURE p select employees sensitive(
        ext modulus               VARCHAR2,
        ext exponent              VARCHAR2,
        secret pass salt      OUT RAW,
        secret pass count     OUT RAW,
        secret pass algorithm OUT RAW,
        secret pass           OUT RAW,
        resultset_out         OUT RESULTSET_TYPE,
        m err no              OUT NUMBER,
        m err txt             OUT VARCHAR2 )
    IS BEGIN
        m err no := 0;
        secret pass salt :=
            appsec.app sec pkg.f get crypt secret salt( ext modulus, ext exponent );
```