CHAPTER 12

Using Fine-Grained Access Control

In This Chapter

Using application security

Understanding application security contexts

Writing security policies

Using pre-8i alternatives

An exciting new Oracle8i feature is the ability to implement row-level security within the database. Oracle refers to this as *fine-grained access control*. It addresses a problem that has plagued application developers and security administrators ever since client/server computing came into vogue. The problem is that an impedance mismatch has existed between the typical relational database's implementation of security at the object level, and the application developer's need for security to be implemented at the row level. For a long time now, we have been able to define access rights to objects such as tables or views at the database level, but row-level security has almost always been forced down into the application. Oracle8i changes all this by implementing features that support fine-grained access control at the database level.

Using Application Security

Application security is loosely defined as the enforcement of limits on what a particular user can do using a given application. Take a payroll application, for example. Any payroll application is capable of adding new employees to the payroll, changing pay rates, and so forth. Even though all these features are available, you certainly don't want all users to avail themselves of them. Instead, you have a limited number of users who are allowed to see and change sensitive data such as an employee's pay rate. You might have a larger pool of users with "look only" access. You may have further restrictions; for instance, a manager is allowed to see only his or her own employees. All of these restrictions are typically enforced through some type of row-level security.

Implementing security in the application

Historically, implementing detailed security policies at the database level has been difficult. Sometimes you can do it using a combination of views and stored procedures, but the amount of human overhead needed to manage all that is high. Third-party development tools, such as Visual Basic and Powerbuilder, are designed around using tables and views, making it cumbersome to funnel all access through stored procedures. Because of these difficulties, row-level security is often implemented at the application level, as illustrated in Figure 12-1.

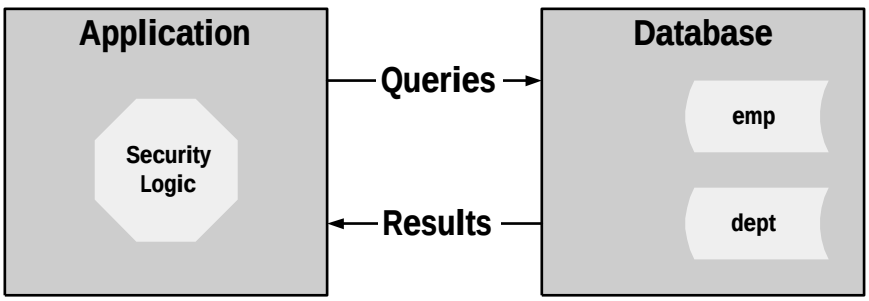


Figure 12-1: Row-level security is often implemented within the application.

When row-level security is implemented as shown in Figure 12-1, you typically run into these problems:

- The programmers don't always get it right.
- Implementation is inconsistent from one application to another.
- The application maintenance burden is greatly increased.
- There is no security when ad-hoc query tools are used.

Oracle8i now offers an alternative approach. Read on to learn about it.

Using application contexts and policies

With the release of Oracle8i, you now have two new database features that enable you to define row-level security within the database consistently, and in a way that is transparent to users and applications accessing the database: security policies and application contexts. Together, you can use these two features to move row-level security from the application into the database, as shown in figure 12-2.

You can use application contexts and security policies separately. Application contexts have uses other than for security. Security policies can be implemented without using an application context. However, any reasonably robust security scheme is likely to leverage both features.

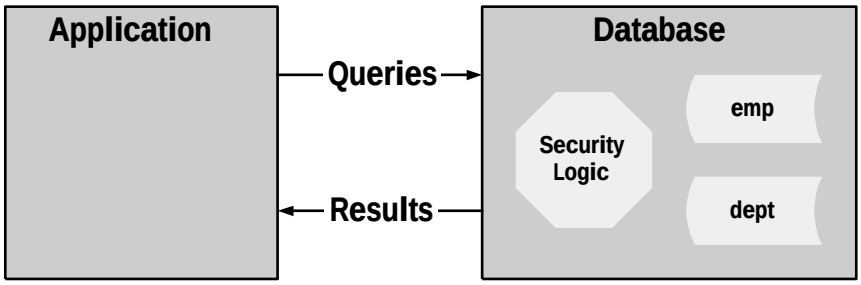


Figure 12-2: Oracle8i allows you to move application security into the database.

Using Security Policies

A *security policy* is a set of rules, attached to an object such as a table or a view, that defines who is allowed to see data from the object, what rows they are allowed to see, and whether those rows can be updated or deleted. You implement security policies by writing PL/SQL code to dynamically modify the `WHERE` clauses of queries against the object being secured.

Figure 12-3 illustrates how a security policy actually works. A query has been issued against the `EMP` table in an attempt to select all rows from that table. When the query is issued, Oracle automatically invokes the security policy for the `EMP` table. The security policy is nothing more than a PL/SQL package that looks up some information about the user, and uses that information to modify the query's `WHERE` clause. All this is done transparently. The user never sees the modified query. In this case, the end result is that a manager will be allowed to see only his or her own employees.

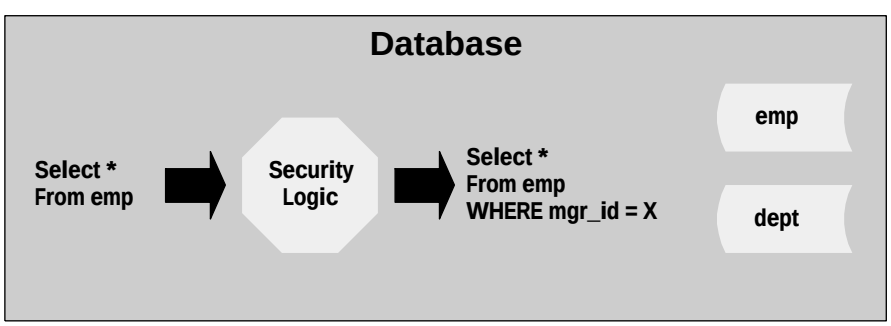


Figure 12-3: Security policies dynamically rewrite the WHERE clause of a query.

For performance reasons, if nothing else, you will want to use application contexts in a row-level security scheme. If a security policy needs to retrieve information from the database and needs to retrieve that information each time it is invoked, performance will suffer.

Using Application Contexts

An *application context* is a scratchpad area in memory that you can use to store bits of information that security policies need. For example, when a user logs on, you might store his or her username, ID number, department number, and other information in an application context. That information can then be referenced by the code that enforces the security policies on objects that the user accesses. Figure 12-4 shows the complete relationship between security policies, application contexts, and the object being secured.

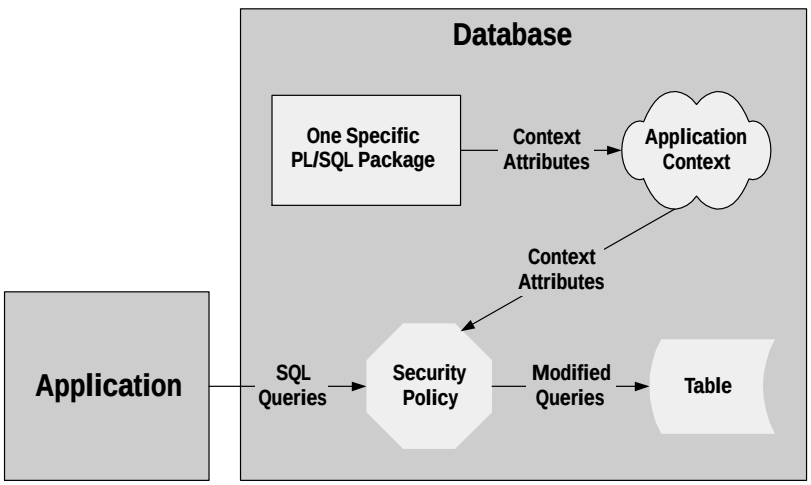


Figure 12-4: Security policies and application contexts work together to secure an object.

Application contexts are restricted to prevent users from arbitrarily updating their context data. When you create an application context, you also need to create a PL/SQL package that you can use to define attributes in that context. The command to create the context links the context to the package. After a context is created, Oracle ensures that only the specified package can be used to define attributes (save values) in that context.

Examining a Seapark Example

The rest of this chapter implements a row-level security model for the Seapark database. The basic rule of that model is this:

> The records of animals and tanks may be modified only by the caretaker responsible for the animal or tank in question.

The examples throughout this chapter illustrate how to build the context and the policies to support this model. This won't be a comprehensive implementation. The example is limited to just the `AQUATIC_ANIMAL` and `TANK` tables, but that should be enough to provide you with a good understanding of how all this works.

Understanding Application Security Contexts

An *application security context* functions as a glorified scratchpad. It is an area in memory where you can store information that you may need occasionally during a session. You define the information that you need to store in terms of attributes. Each attribute in a context has a name and a value. Both the name and the value are text strings that you supply when you create the attribute.

Creating and dropping a context

You manage contexts using the `CREATE CONTEXT` and `DROP CONTEXT` commands. The `CREATE` command creates; the `DROP` command deletes. There is no `ALTER CONTEXT` command.

Creating a Context

Before creating a context, you must think up a name. The name can be anything you like, as long as it conforms to Oracle's naming rules for objects. Try to give the context a name that reflects the application that it is associated with. The syntax for the `CREATE CONTEXT` command looks like this:

```
CREATE [OR REPLACE] CONTEXT context_name
    USING [schema.]package;
```

The package name that you supply when you create a context identifies the PL/SQL package that is allowed to create and set attributes within the context. The package doesn't need to exist when you create the context. You can create it afterwards.

Creating a context doesn't really create anything in the sense that creating a table does. When you create a context, all that really happens is that you get a new entry in the data dictionary. Contexts never use disk space, and they use memory only when attributes are set.

You must have the `CREATE ANY CONTEXT` system privilege to create a context. Because contexts aren't created often, and because they are global to the entire database, you may want to do all the context creating yourself, rather than grant this role to a user.

Dropping a Context

You use the `DROP CONTEXT` command to remove a context. The syntax is simple, and it looks like this:

```
DROP CONTEXT context_name;
```

When you drop a context, its entry is removed from the data dictionary, and you can no longer define attributes for that context. However, users who currently have attributes set for the context will retain those attributes until they disconnect.

Defining and retrieving context attributes

Once you have created a context, you can define attributes in that context using the DBMS_SESSION package. Later, you can retrieve the values of those attributes using the new SYS_CONTEXT function.

Defining Attributes

You define an attribute in a context by making a call to DBMS_SESSION.SET_CONTEXT. The syntax of that call looks like this:

```
DBMS_SESSION.SET_CONTEXT (
'context_name','attribute_name','attribute_value');
```

For example, if you had a Seapark application user responsible for tank #4, and you wanted to store that information in the SEAPARK context, you could call SET_CONTEXT like this:

```
DBMS_SESSION.SET_CONTEXT('SEAPARK','TANK','4');
```

You can't execute a SET_CONTEXT call interactively using SQL*Plus or any similar tool. Context attributes can be changed only by the package named when the context was created. You must code calls to SET_CONTEXT within that package.

Retrieving Attributes

You can retrieve attribute values from a context using Oracle's new built-in SYS_CONTEXT function. The syntax for calling that function is as follows:

```
attribute_value:=SYS_CONTEXT('context_name','attribute_name');
```

Unlike the case when setting an attribute value, SYS_CONTEXT doesn't need to be invoked from any particular PL/SQL package. You can use SYS_CONTEXT from within any PL/SQL code. You can also use it as part of a SELECT statement. This example shows SYS_CONTEXT being used from within a SELECT statement to retrieve the value of the USERNAME attribute from the SEAPARK context:

```
SQL> select sys_context('seapark','username') from dual;

SYS_CONTEXT('SEAPARK','USERNAME')
------------------------------------------------------------
HAROLD

1 row selected.
```

As you can see from this example, context names and attribute names are not case-sensitive.

Using Oracle's Predefined Context—USERENV

Oracle implements a predefined context containing some attributes that are useful when implementing fine-grained security. The context name is USERENV, and the predefined attributes are the following:

SESSION_USER	The username that was used to log into the database.
CURRENT_USER	If a PL/SQL procedure, function, or package is being executed, this will be the name of the user who created it. Otherwise, this will be the same as SESSION_USER.
CURRENT_SCHEMA	The name of the current schema. Most often, this will be identical to the CURRENT_USER value.
SESSION_USERID	The numeric user ID corresponding to the session user. This is the same value as found in the USER_ID column of the DBA_USERS view.
CURRENT_USERID	The numeric user ID corresponding to the current user.
CURRENT_SCHEMAID	The numeric user ID corresponding to the current schema.
NLS_CALENDAR	The name of the NLS calendar currently being used for dates, for example, "GREGORIAN".
NLS_CURRENCY	The currency indicator, for example, "$".
NLS_DATE_FORMAT	The current default date format, for example, "DD-MON-YY".
NLS_DATE_LANGUAGE	The current date language, for example, "AMERICAN".
NLS_SORT	The current linguistic sorting method being used, for example, "BINARY".
NLS_TERRITORY	The current territory setting, for example, "AMERICA".

For security purposes, SESSION_USER is by far the most useful of all these attributes. In the past, detecting the logon username from within a stored PL/SQL procedure was difficult. Now, it's easy. Just code the following:

```
username := SYS_CONTEXT('USERENV','SESSION_USER');
```

Later in the chapter, you'll see how the USERENV context's SESSION_USER attribute forms the cornerstone of the row-level security model for the Seapark database.

Examining the Seapark context

Getting back to our Seapark scenario, remember that our security goal was this:

> The records of animals and tanks may be modified only by the caretaker responsible for the animal or tank in question.

To help implement that goal, we will create a context for the Seapark application. Within that context, we will define attributes that contain the current user's username and the tanks for which he or she is responsible.

You can find scripts to create the SEAPARK context and the package used to define attributes within that context on the companion CD in the directory named fine_grained_security_examples.

Creating the Seapark Context

Creating the context is the simple part. Log on as the SYSTEM user and execute a CREATE CONTEXT command, as shown in this example:

```
SQL> CREATE OR REPLACE CONTEXT seapark
  2      USING seapark.seapark_context;
```

This creates the context. The file named seapark_context.sql on the CD contains this command. The next task is to create the SEAPARK_CONTEXT package referenced in the command.

Creating the SEAPARK_CONTEXT Package

The SEAPARK_CONTEXT package is the only package that is allowed to define attributes in the SEAPARK context's namespace. Application users should not have access to this package because if they were able to change it, they would have an opening to bypass security. For this example, we will store the package in the SEAPARK schema. The assumption is that application users all have their own user IDs, and that only the DBA or other trusted users are allowed to log on as SEAPARK. If you want to secure the data from even the schema owner, you can do this by creating a different schema just for this one package.

To create the package, log on to the database as the SEAPARK user, and execute the two files on the CD named seapark_context_package.sql and seapark_context_body.sql. For an example, see Listing 12-1.

Listing 12-1: Creating the SEAPARK_CONTEXT package

```
SQL> @seapark_context_package
SQL> SET ECHO ON
SQL>
SQL> CREATE OR REPLACE PACKAGE seapark_context IS
  2      PROCEDURE set_attributes;
```

```
  3  END seapark_context;
  4  /
Package created.

SQL>
SQL> @seapark_context_body
SQL> SET ECHO ON
SQL>
SQL> CREATE OR REPLACE PACKAGE BODY seapark_context IS
  2      PROCEDURE set_attributes IS
  3          username caretaker.caretaker_name%type;
  4          tank_list VARCHAR2(100);
  5      BEGIN
  6          --Set a flag to temporarily disable policies on
  7          --the tank table.
  8          DBMS_SESSION.SET_CONTEXT
  9              ('seapark','logon','true');
 10
 11          --Use the new SYS_CONTEXT function to grab
 12          --the username of the currently logged on
 13          --user.
 14          username := SYS_CONTEXT('userenv','session_user');
 15
 16          --Now, query the tank table to see which tanks,
 17          --if any, this user is responsible for.
 18          tank_list := '';
 19
 20          FOR tank IN (
 21          SELECT tank_no
 22          FROM tank
 23          WHERE chief_caretaker_name = username)
 24          LOOP
 25              IF tank.tank_no IS NOT NULL THEN
 26                  IF (LENGTH(tank_list) > 0)  THEN
 27                      tank_list := tank_list || ',';
 28                  END IF;
 29
 30                  tank_list :=tank_list
 31                            || to_char(tank.tank_no);
 32              END IF;
 33          END LOOP;
 34
 35          --Store the username and the tank list in
 36          --the Seapark application context.
 37          DBMS_SESSION.SET_CONTEXT
 38              ('seapark','username',username);
 39          DBMS_SESSION.SET_CONTEXT
 40              ('seapark','tank_list',tank_list);
 41
 42          --Reset the logon flag, and enable security.
 43          DBMS_SESSION.SET_CONTEXT
```

Continued

Listing 12-1 *(continued)*

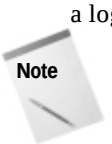
```
44              ('seapark','logon','false');
45      END;
46  END seapark_context;
47  /

Package body created.
```

The purpose of the SET_ATTRIBUTES procedure in this package is to determine the tank(s) for which a user is responsible, and to store that information in the SEAPARK context. Notice that the previous code does the following:

1. Makes a call to SET_CONTEXT to set a logon flag to TRUE
2. Makes a call to SYS_CONTEXT to get the logon username
3. Uses the logon username to query the TANK table
4. Builds a list of tanks for which the caretaker is chiefly responsible
5. Uses the SET_CONTEXT function to store the username and the tank list in the SEAPARK context for the user
6. Makes a call to SET_CONTEXT to set the logon flag to FALSE

This procedure is designed to be called from a database logon trigger. The logon flag is set to TRUE at the beginning and FALSE at the end to avoid the catch-22 of having security enforced on the tank table before the tank list has been retrieved for the user. The reason for this will become more apparent when you read the section on policies.

The last issue to deal with, now that the context and the context's package have been created, is to find some way to run the SET_ATTRIBUTES procedure for each user. One possibility is to have the application call the procedure. Another possibility is to write a logon trigger to call SET_ATTRIBUTES automatically whenever a user connects.

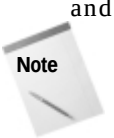


Even if users were to manually invoke SET_ATTRIBUTES, they wouldn't be able to control the attributes that it defined in the context. The keys here are to ensure that users can't log on as SEAPARK (give them their own usernames to use), and that you don't grant CREATE ANY PROCEDURE to users you do not trust.

Creating a Logon Trigger

Logon triggers are a new Oracle8i feature that provides an excellent method for ensuring that a user's application context attributes are properly set no matter how that user connects to the database.

To create a logon trigger to call the SET_ATTRIBUTES procedure, log on as SYSTEM and execute a CREATE TRIGGER statement like that shown in the following example:



Note Before attempting to create this trigger, you must grant execute privileges on the SEAPARK_CONTEXT package to the SYSTEM user.

```
SQL> CREATE OR REPLACE TRIGGER seapark_logon
  2      AFTER LOGON ON DATABASE
  3      CALL seapark.seapark_context.set_attributes
  4  /
```

Note If you change the SEAPARK_CONTEXT package after creating the logon trigger, you will need to recompile the trigger. Do that by logging on as SYSDBA and issuing an ALTER TRIGGER SYSTEM.SEAPARK_CONTEXT command.

Once you have created the trigger, you can log on as any user, and the attributes for the Seapark application will automatically be set. You can see these attributes through the SESSION_CONTEXT view, as the following example shows:

```
SQL> CONNECT harold/harold
Connected.
SQL> SELECT * FROM session_context;

NAMESPACE  ATTRIBUTE  VALUE
---------- ---------- ----------
SEAPARK    USERNAME   HAROLD
SEAPARK    TANK_LIST  1,4
```

Note If you run the schema creation scripts from the CD included with this book, HAROLD will be the only database user created who also manages a tank.

Once the logon trigger sets these attributes, the user can't do anything to change them. They stay set until the user logs off. The logon trigger ensures that no matter how the user connects to the database, these attributes get defined.

Caution Be careful when creating logon triggers. If you create a trigger with errors, those errors will prevent all database users from logging on to the database. If a bad logon trigger prevents you from connecting, you can bypass it by connecting as either SYSDBA or as INTERNAL. Then you can drop the offending trigger. For more information, see Chapter 24, "Using Procedures, Packages, Functions, and Triggers."

Getting the context set up and the attributes properly defined is the first big step towards implementing row-level security for an application. The next step is to write some security policies to enforce security based on those attributes.

Writing Security Policies

A *policy* is a set of rules that governs the data that a user is allowed to see and modify within a table. In Oracle8i, you implement policies by writing functions that return query predicates. These predicates are then appended to queries issued by a user. Thus, while a user might issue a query like this:

```
DELETE
FROM aquatic_animal
WHERE id_no = 151
```

the security policy on the AQUATIC_ANIMAL table would transparently add a predicate to the query so that the end result would look like this:

```
DELETE
FROM aquatic_animal
WHERE id_no = 151
AND tank_no IN (1,4)
```

In this example, the list of tank numbers represents the tanks for which the currently logged-on user is responsible. The additional predicate prevents the user from deleting animals from other caretaker's tanks.

Implementing a policy

To implement a policy, you generally follow these steps:

1. Write a stored function or package that conforms to the policy specifications.
2. Use the DBMS_RLS.ADD_POLICY procedure to associate the policy function with the table(s) that you are protecting.

One policy function can protect more than one table, and a table can have different policies for selecting, inserting, updating, and deleting. Examples of both scenarios are provided throughout this section.

Creating a Policy Function

Policies are written as PL/SQL functions. Oracle calls these functions automatically and expects them to conform to this specification:

```
FUNCTION function_name (
        object_schema IN VARCHAR2,
        object_name IN VARCHAR2) RETURN VARCHAR2;
```

Oracle uses the two input arguments to pass in the owner name and object name of the object being queried. In return, Oracle expects a character string that can be appended onto the end of a SQL statement's WHERE clause. This return value is referred to as a *predicate*.

Your function may do whatever it needs to do to determine the proper predicate to return. It can query other database tables, check the username of the currently logged on user, and reference values stored in an application context. For example, the function shown in Listing 12-2 retrieves the user's tank list from the SEAPARK context and uses it to build a predicate that limits the results of a query to records with tank numbers that are found in the user's list.

Listing 12-2: **Returning a predicate**

```
FUNCTION update_limits (
        object_schema IN VARCHAR2,
        object_name IN VARCHAR2) RETURN VARCHAR2
    IS
        tank_list VARCHAR2(100);
        predicate VARCHAR2(113);
    BEGIN
        --Retrieve the tank list for this user
        tank_list := SYS_CONTEXT('seapark','tank_list');

     --Limit the user to his or her own tanks
        predicate := 'tank_no IN (' || tank_list || ')';

        --Return the predicate so it can be appended
        --to the query's WHERE clause.
        RETURN predicate;
    END;
```

The predicate returned by this function looks like this:

```
tank_no IN (1,4)
```

Note that there is no trailing semicolon, and the word WHERE isn't included as part of the predicate. Policy functions may return any valid predicate. Predicates may contain subqueries or any other valid SQL construct.

Implementing the Seapark Policies

If you have several related policies, consider implementing those policies as part of a package. The SEAPARK_POLICY package shown in this section implements the row-level security requirements discussed earlier for the Seapark database, and it contains these two functions:

- UPDATE_LIMITS
- SELECT_LIMITS

The UPDATE_LIMITS function is intended to be called whenever the user issues a query that changes the data in either the TANK table or the AQUATIC_ANIMALS table. It returns the predicate necessary to limit changes to the tanks for which the user is responsible. It also disallows changes to dead animals. In addition, users who aren't caretakers — in other words, who don't have a tank list — aren't allowed to make any changes.

The SELECT_LIMITS function is intended to be called whenever a SELECT statement is issued against one of the two tables. The rules are different for SELECTS. Caretakers are allowed to see each other's records, and they may see information about dead animals.

The code to create the SEAPARK_POLICY package is contained on the CD in the two files named seapark_policy_package.sql and seapark_policy_body.sql. You can create the package by logging on as the user SEAPARK and executing those scripts from SQL*Plus, as shown in Listing 12-3.

Listing 12-3: **Creating the SEAPARK_POLICY package**

```
SQL> @seapark_policy_package
SQL> SET ECHO ON
SQL>
SQL> CREATE OR REPLACE PACKAGE seapark_policy IS
  2      FUNCTION update_limits (
  3          object_schema IN VARCHAR2,
  4          object_name IN VARCHAR2) RETURN VARCHAR2;
  5
  6      FUNCTION select_limits (
  7          object_schema IN VARCHAR2,
  8          object_name IN VARCHAR2) RETURN VARCHAR2;
  9  END seapark_policy;
 10  /

Package created.
SQL>
SQL> @seapark_policy_body
SQL> SET ECHO ON
SQL>
SQL> CREATE OR REPLACE PACKAGE BODY seapark_policy IS
  2      FUNCTION update_limits (
  3          object_schema IN VARCHAR2,
  4          object_name IN VARCHAR2) RETURN VARCHAR2
  5      IS
  6          tank_list VARCHAR2(100);
  7          predicate VARCHAR2(113);
  8      BEGIN
  9          --Retrieve the tank list for this user
```

```
10          tank_list := SYS_CONTEXT('seapark','tank_list');
11
12          --If the tank list is null, the user has
13          --NO access. Otherwise, the user can change
14          --data related to his or her tank.
15          IF tank_list IS NULL THEN
16              --No data will match this condition.
17              predicate := '1=2';
18          ELSE
19              --Limit the user to his or her own tanks
20              predicate := 'tank_no IN ('
21                           || tank_list || ')';
22          END IF;
23
24          --Prevent changes to dead animals
25          IF object_schema = 'SEAPARK'
26          AND object_name = 'AQUATIC_ANIMAL' THEN
27              predicate := predicate
28                           || ' AND death_date IS NULL';
29          END IF;
30
31          --Return the predicate so it can be appended
32          --to the query's WHERE clause.
33          RETURN predicate;
34      END;
35
36      FUNCTION select_limits (
37          object_schema IN VARCHAR2,
38          object_name IN VARCHAR2) RETURN VARCHAR2
39      IS
40          tank_list VARCHAR2(100);
41          predicate VARCHAR2(113);
42      BEGIN
43          --If logging in, then return null so
44          --the logon trigger can see values in
45          --the tank table.
46          IF SYS_CONTEXT('seapark','logon') = 'true' THEN
47              RETURN '';
48          END IF;
49
50          --Retrieve the tank list for this user
51          tank_list := SYS_CONTEXT('seapark','tank_list');
52
53          --If the tank list is null, the user has
54          --NO access. Otherwise, the user can see
55          --anything.
56          IF tank_list IS NULL THEN
57              --No data will match this condition.
58              predicate := '1=2';
59          ELSE
60              --The user is a caretaker. Let him or her
```

Continued

Listing 12-3 *(continued)*

```
61              --see data for the other caretakers.
62              predicate := '';
63          END IF;
64
65          --Return the predicate so it can be appended
66          --to the query's WHERE clause.
67          RETURN predicate;
68      END;
69  END seapark_policy;
70  /

Package body created.
```

You can see that the two functions are pretty much the same, except that the SELECT_LIMITS function allows caretakers to see any data. One other difference is that SELECT_LIMITS issues a call to SYS_CONTEXT to see if the logon attribute is defined as 'true'; if true, it returns a null value as the predicate. The function does this because the logon trigger needs to query the tank table to retrieve the list of tanks for which a user is responsible. Because that list can't be known until after the query has been made, the logon trigger needs a way to bypass the security policy for that one query, hence the use of the logon flag.

Users will not be able to bypass security by setting the logon flag themselves. Only the SEAPARK_CONTEXT package can change the value of an attribute in the SEAPARK context.

Three other points are worth noting about these functions, and about policy functions in general:

- When you want to allow a user unrestricted access to a table, have your policy function return a null string. The SELECT_LIMITS function does that when the logon flag is true.
- When you want to prevent a user from seeing any data in a table, return a predicate that will always be false. These functions use 1=2 for that purpose.
- The OBJECT_SCHEMA and OBJECT_NAME arguments may be used to code rules that are specific to a particular table. The UPDATE_LIMITS function uses these to apply the dead animal rule only to the AQUATIC_ANIMAL table.

With the policy functions created, the next step is to associate them with the tables that you want to secure.

Adding a Policy to an Object

Once you have created a policy, you associate that policy with a table using the DBMS_RLS package.

You must have execute access on DBMS_RLS to use it. Because the package is owned by SYS, you will need to log on as SYS to grant that access.

You use the DBMS_RLS.ADD_POLICY procedure to link a policy function with a table. The syntax looks like this:

```
DBMS_RLS.ADD_POLICY (
'object_schema','object_name','policy_name',
'function_schema','function_name'
'statement_types',update_check)
```

The following list describes the syntax in detail:

- *object_schema* — The name of the schema containing the object that you are protecting. In this chapter, we are dealing with the SEAPARK schema.
- *object_name* — The name of the object, table, or view that you are protecting.
- *policy_name* — A name for the policy. This can be any name that you like. It's used with other DBMS_RLS calls to identify the policy that you are working with.
- *function_schema* — The owner of the policy function. For the examples in this book, the object owner (SEAPARK) also owns the function. It is possible, though, to have someone else own the function.
- *function_name* — The name of the policy function. If the function is part of a package, you must qualify the function name with the policy name.
- *statement_types* — A comma-separated list of data manipulation language (DML) statements for which you want this function to be invoked. The valid DML statements are SELECT, INSERT, UPDATE, and DELETE.
- *update_check* — A Boolean argument (not a string), which can be either TRUE or FALSE. A value of TRUE prevents a user from updating a row in such a way that the result violates the policy. A value of FALSE prevents that check from being made.

To associate the Seapark policies with the TANK and AQUATIC_ANIMALS tables, you can log on as SEAPARK using SQL*Plus and execute the calls to DBMS_RLS.ADD_POLICY as found in the file named seapark_add_policy.sql. See Listing 12-4.

Listing 12-4: **Associating policies with tables**

```
SQL> @seapark_add_policy
SQL> SET ECHO ON
SQL>
SQL> EXECUTE DBMS_RLS.ADD_POLICY ( -
>     'SEAPARK','TANK','TANK_UPDATE', -
>     'SEAPARK','SEAPARK_POLICY.UPDATE_LIMITS', -
>     'INSERT, UPDATE, DELETE', TRUE);

PL/SQL procedure successfully completed.

SQL>
SQL> EXECUTE DBMS_RLS.ADD_POLICY ( -
>     'SEAPARK','TANK','TANK_SELECT', -
>     'SEAPARK','SEAPARK_POLICY.SELECT_LIMITS', -
>     'SELECT', TRUE);

PL/SQL procedure successfully completed.

SQL>
SQL> EXECUTE DBMS_RLS.ADD_POLICY ( -
>     'SEAPARK','AQUATIC_ANIMAL','AQUATIC_ANIMAL_UPDATE', -
>     'SEAPARK','SEAPARK_POLICY.UPDATE_LIMITS', -
>     'INSERT, UPDATE, DELETE', TRUE);

PL/SQL procedure successfully completed.

SQL>
SQL> EXECUTE DBMS_RLS.ADD_POLICY ( -
>     'SEAPARK','AQUATIC_ANIMAL','AQUATIC_ANIMAL_SELECT', -
>     'SEAPARK','SEAPARK_POLICY.SELECT_LIMITS', -
>     'SELECT', TRUE);

PL/SQL procedure successfully completed.
```

Two policies have been linked with each table: one for selecting and one for everything else. You now have the situation shown in Table 12-1.

Each table has two policies. Each policy function is used for two objects. Having the tank number field named the same in both tables makes that easy to do. The exception — where `AQUATIC_ANIMAL` is the only table with a death date — is handled by having the `UPDATE_LIMITS` function check the object name.

Table 12-1
Seapark Security Policies

Table	*Policy*	*DML*	*Function*
TANK	TANK_UPDATE	INSERT UPDATE DELETE	SEAPARK_POLICY.UPDATE_LIMITS
	TANK_SELECT	SELECT	SEAPARK_POLICY.SELECT_LIMITS
AQUATIC _ANIMAL	AQUATIC_ ANIMAL_ UPDATE	INSERT UPDATE DELETE	SEAPARK_POLICY.UPDATE_LIMITS
	AQUATIC_ ANIMAL_ SELECT	SELECT	SEAPARK_POLICY.SELECT_LIMITS

Verifying Policy Enforcement

If you've been following along with all the examples in this chapter, you can see the effects of policy enforcement by logging on as the user named HAROLD. If you log on as HAROLD and select data from the AQUATIC_ANIMAL table, you'll see all the records, as shown in Listing 12-5.

Listing 12-5: **HAROLD can see all animal records.**

```
SQL> SELECT id_no, tank_no, animal_name
  2  FROM seapark.aquatic_animal;

    ID_NO   TANK_NO ANIMAL_NAME
--------- --------- ------------------------------
      100         1 Flipper
      105         1 Skipper
      112         1 Bopper
      151         2 Batty
      166         2 Shorty
      145         2 Squacky
      175         2 Paintuin
      199         3 Nosey
      202         3 Rascal
      240         3 Snoops

10 rows selected.
```

This ability to see all the records is consistent with the policy of allowing caretakers to view data about each other's animals. However, if you try to delete or update those same records, the results are different:

```
SQL> DELETE FROM seapark.aquatic_animal;

3 rows deleted.
```

Even though the table contains ten rows, Harold was able to delete only three. Harold is the caretaker for tank 1, so he is allowed to delete records only for animals in that tank.

You may want to roll back this deletion.

Here is another example showing the update check option at work:

```
SQL> UPDATE seapark.aquatic_animal
  2  SET death_date = SYSDATE;
UPDATE seapark.aquatic_animal
               *
ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation
```

Harold has attempted to define a death date for all his animals. The UPDATE_LIMITS policy doesn't allow records for dead animals to be altered. By setting the update check option to true, we have prevented Harold from changing a record from a state in which it can be altered to a state in which it cannot.

You might want to disable the update check on the AQUATIC_ANIMAL_UPDATE policy because you probably do want Harold to be able to record the death of one of his animals.

To see the effect of the policies on a user who is not a caretaker, log on as SEAPARK and try selecting from either the TANK or AQUATIC_ANIMAL tables. You won't get any rows back because SEAPARK isn't listed in the TANK table as a caretaker.

Listing policies on an object

You can get a list of policies on an object by querying one of these views:

DBA_POLICIES	Returns information about all policies defined in the database
ALL_POLICIES	Returns information about policies defined on all objects to which you have access
USER_POLICIES	Returns information about policies on objects that you own

These views are fairly self-explanatory. The column names make obvious the type of data they contain. The following example shows how you can generate a list of all policies on all objects that you own:

```
SQL> SELECT object_name, policy_name, sel, ins, upd, del
  2  FROM user_policies
  3  ORDER BY object_name, policy_name;

OBJECT_NAME     POLICY_NAME               SEL INS UPD DEL
--------------- ------------------------- --- --- --- ---
AQUATIC_ANIMAL  AQUATIC_ANIMAL_SELECT     YES NO  NO  NO
AQUATIC_ANIMAL  AQUATIC_ANIMAL_UPDATE     NO  YES YES YES
TANK            TANK_SELECT               YES NO  NO  NO
TANK            TANK_UPDATE               NO  YES YES YES
```

The yes/no flags indicate the type of DML statements to which the policy applies. The USER_OBJECTS view also contains columns to return the name of the package and function used to enforce the policy, if you need that information. The ALL_POLICIES and DBA_POLICIES views differ from USER_POLICIES by one additional field that identifies the owner of an object.

Enabling and disabling a policy

The DBMS_RLS package allows you to temporarily disable a policy without having to drop it. This comes in handy if you have some maintenance work to do on a table and you don't want to be thwarted by your own security. The DBMS_RLS.ENABLE procedure is used both to disable and to enable a policy. The syntax looks like this:

```
DBMS_RLS.ENABLE_POLICY (
object_schema,
object_name,
policy_name,
enable_flag);
```

The *enable_flag* argument is a Boolean, and you set it to either TRUE or FALSE, depending on whether you want the policy enforced.

Dropping a policy

You can drop (remove) a policy by making a call to the DBMS_RLS package's DROP_POLICY procedure. The syntax for this looks like the following:

```
DBMS_RLS.DROP_POLICY (
   object_schema,
   object_name,
    policy_name);
```

If you've been following along through the examples in this chapter, you may want to drop any policies that you have created. The SQL*Plus script on the CD named drop_policy_examples.sql will do this for you.

Using Pre-8i Alternatives

People have used a few approaches in the past to implement row-level security within the database. These approaches generally include some combination of the following:

- ✦ Views to restrict the data that a user can see
- ✦ Triggers to prevent unauthorized updates to tables
- ✦ Stored procedures as a vehicle for managing updates and deletes

Enforcing security using these features can work, but it takes careful application design because use of these features isn't always transparent to the application.

Enforcing security with views

You can use views as a mechanism to enforce row-level security. They work best when you're only selecting data, not changing or deleting it. The following view on the `TANK` table limits users to viewing information only about their tanks:

```
CREATE OR REPLACE VIEW user_tank AS
    SELECT *
    FROM tank
    WHERE CHIEF_CARETAKER_NAME = USER;
```

By granting users `SELECT` access on this view but not the underlying table, you limit them to seeing only tanks that they manage. Since this particular view doesn't contain any joins or subqueries, it could be used to control `INSERTS`, `UPDATES`, and `DELETES` as well as `SELECTS`.

Oracle uses the view method extensively in its data dictionary. The `ALL` views, for example, are defined in such a way as to allow you to see information about any objects to which you have access. That's all done through the `WHERE` clause. You will run into some issues, however, when using views for security:

- ✦ The `WHERE` clauses can quickly become complex. Subqueries and joins may become necessary, thus rendering the views nonupdateable.
- ✦ When a view is nonupdateable, you need a different mechanism for securing `INSERTS`, `UPDATES`, and `DELETES`.
- ✦ You may need different views for different classes of users. Your application programs need to be aware of which view to use for any given user.
- ✦ If the number of views is quite high, and especially if they change frequently, the maintenance burden can be significant.

In spite of these issues, views do have a place in a security plan. They remain the only method of limiting a user's access to specific columns within a table.

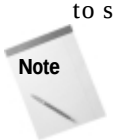
Enforcing security with triggers

Triggers provide another mechanism for enforcing security on a table. The trigger on the TANK table shown in Listing 12-6 restricts users to their own tanks.

Listing 12-6: **Using a trigger to enforce security on a table**

```
CREATE OR REPLACE TRIGGER user_tank
    BEFORE UPDATE OR INSERT OR DELETE ON tank
    FOR EACH ROW
BEGIN
    IF UPDATING OR DELETING THEN
        IF :old.chief_caretaker_name = USER THEN
            NULL;
        ELSE
            RAISE_APPLICATION_ERROR (
                -20000,
                'You can''t update another caretaker''s
tank.');
        END IF;
    END IF;
    IF INSERTING THEN
        IF :old.chief_caretaker_name = USER THEN
            NULL;
        ELSE
            RAISE_APPLICATION_ERROR (
                -20000,
                'You can''t insert a tank for another
caretaker.');
        END IF;
    END IF;
END;
/
```

One advantage of using triggers is that they are transparent to the application. A major disadvantage, however, is that they work only for inserts, updates, and deletes. You still need to depend on some other method, such as using views, to secure selects.



It is possible to grant SELECT on a view, and only INSERT, UPDATE, and DELETE on the underlying table. Applications would then select from the view, but update (or delete or insert into) the table.

Enforcing security with procedures

You can sometimes use stored procedures to enforce business rules and security at the database level. The following procedure, for example, allows you to change an animal's name, but only if that animal is still alive:

```
CREATE OR REPLACE PROCEDURE change_animal_name (
    id_no IN NUMBER,
    animal_name IN VARCHAR2) AS
BEGIN
    UPDATE aquatic_animal
    SET animal_name = change_animal_name.animal_name
    WHERE id_no = change_animal_name.id_no
    AND death_date IS NULL;
END;
/
```

To make this secure, you would grant users execute access on the stored procedure, but you wouldn't grant `UPDATE` access to the underlying table. That would force all updates to be done through procedure calls. One problem with this approach is that it isn't transparent to the application. Instead, the application has to be coded to perform updates through procedure calls. Depending on the development tool that you are using, this may not be as easy as you would like it to be. Another problem with using stored procedures is that prior to the release of Oracle8i, with its `SYS_CONTEXT` function, getting the name of the currently logged-on ser from a stored procedure wasn't possible.

Summary

In this chapter, you learned:

- In Orace8i, you can use policies and contexts to enforce fine-grained security, giving you control over which rows a user is allowed to access.
- Application contexts serve as scratchpad areas in memory. Users can freely read from a context, but updates can be made only by the specific PL/SQL package named when the package was created.
- You can use logon triggers to ensure that context attributes for a user are set when a user first logs on. This is done by having the logon trigger call the specific package allowed to update the context.
- Security policies function by dynamically adding a predicate to a user's queries. This transformation is transparent to the user, and the user never sees the final version of the query.
- Fine-grained security offers advantages over previous methods for implementing row-level security in that it allows for a unified and consistent approach to be taken, and it doesn't require any specific coding at the application level.

✦ ✦ ✦