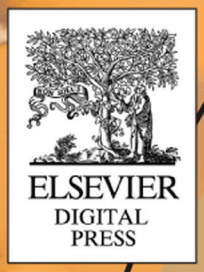
SECURITY



Ron Ben Natan

Implementing Database Security and Auditing

Includes Examples For:
ORACLE
SQL SERVER
DB2 UDB
SYBASE

6

Using Granular Access Control

Once upon a time, when we had client-server systems, we would assign a separate database login for every end user accessing the application. The application client would log in to the database, and the user model in the application relied on the database user model and privileges definitions. Some permissions were managed by the application layer, but others could be enforced directly within the database.

Along came three-tier architectures, n-tier architecture, and application servers, and we suddenly found ourselves with multiple user models. The application user model and the database user model drifted apart. Application logins are no longer commonly associated one-for-one with database logins. Instead, the application server manages a connection pool of database connections. Every time an application thread needs to access the database it requests a connection from the pool, uses it to execute queries and/or procedures, and then surrenders the connection back to the pool. Each connection in the pool is logged into the database using the same database login. Therefore, all of the database authorization mechanisms become trivial and cannot be used effectively (or even used at all!).

This is not a healthy situation, and remedying this issue is the main focus of this chapter. However, database connection pools are not the enemy, and you should not try to move away from them, because they simplify the architecture and allow for much better performance. Therefore, in aligning the user models, I certainly do not mean to suggest that you should do away with the notion of reusing database connections, getting rid of the application user model and going back to a one-to-one relationship between application logins and database logins. Among other reasons, this is completely impractical in the many Web applications where there could be hundreds of thousands and even millions of users. Instead, aligning the user models simply means that when the application gets a connection from the connection pool, the first thing it should do is to communicate with the

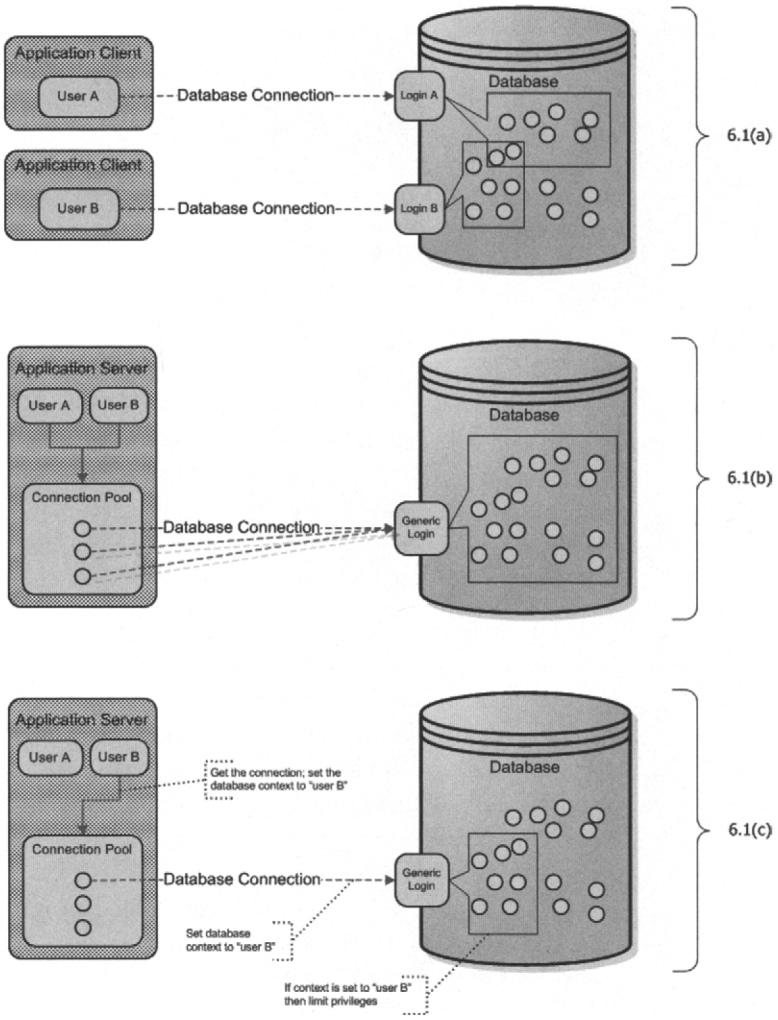


Figure 6.1
Realigning the database user model with the application user model.

database to let it know the identity of the application user, on behalf of whom all queries that will follow on this connection are made. This process is shown in Figure 6.1: 6.1(a) shows the client/server aligned model, 6.1(b) shows the user model mismatch, and 6.1(c) shows the crux of reestablishing alignment by sending the application user information to the database.

Communicating the application user on behalf of whom the current queries are being sent to the database provides many options for both the database as well as external security layers—options that can elevate your overall database security level. You will learn several techniques for communicating the application user to the databases and how to use this additional

information to implement granular access control. In learning about granular access control, you will also see some fairly advanced options that have emerged from security-conscious environments, such as federal agencies. Finally, you will get an overview of some advanced integration topics that you may encounter in large enterprises, including the integration with LDAP repositories and identity management tools.

6.1 Align user models by communicating application user information

The application user model will always be "broader" than the database user model. Applications can support hundreds of users, but they sometimes support thousands and millions of users; the database will not have that many users—at least not natively. However, you can easily "project" the application user into the database. At the most basic level, all you need to do is agree on a database call that will carry this information (i.e., on an agreed-upon communication pattern that both the application and the database can understand). You can do this using any procedure or any query, so long as both the application owner and the database security owner agree to it.

All the application needs to do is communicate the user information *within* the database session (*over* the database connection). More specifically, you only need to make an additional SQL call within that database session and communicate the user information as a data value within that SQL. This is usually done by calling a database procedure and passing the application user identifier as an argument to the procedure. If the database engine is responsible for fine-grained access control, then it can associate the username it received through the procedure call or the query with the database login that was used to initiate the connection (and which tags this session). Section 6.2 will show you how database engine-based fine-grained access control is accomplished based on this value that is communicated from the application layer.

Although you will see a database-centric approach in Section 6.2, not all databases support granular access control within the database. Additionally, sometimes it will not be practical to do this at the database level—either because the schema cannot be changed or because the environment cannot afford to go through a change. Luckily, communicating the application user credentials within the session also works well when using an external security system. Furthermore, using an external system is always possible, does not require changing the database environment, and

does not affect database performance. As an example, suppose that you choose to deploy a SQL firewall similar to that shown in Figure 5.11. This database security layer inspects each database session and each SQL call and compares it with a security policy. If a SQL call diverges from the security policy, it will alert on the call or even deny access to the database. Such a security system takes each SQL call and associates a set of values with it. For example, suppose that I sign on to a DB2 instance running on a server with an IP address of 10.10.10.5 from an application server running on a server with an IP address of 192.168.1.168. Assume also that I sign on using APPSRV to issue a SQL call such as `UPDATE EMPLOYEE SET SALARY=SALARY*1.1`. In this case the security system will know the following:

- The request is coming from 192.168.1.168.
- The request is being made on 10.10.10.5.
- The database login name is APPSRV.
- The command being issued is UPDATE.
- The database object being touched is EMPLOYEE.

I can implement a policy easily enough that says that the EMPLOYEE table cannot be updated by any connection using APPSRV, but what happens if all access is being done from the application server? What happens when I have certain users (e.g., managers) who are able to give everyone a 10% raise but other application users (and going forward I will use application user with an employee ID of 999) should only be able to select the data but cannot perform any DML commands on the EMPLOYEE table. In this case the information that the security system sees is not enough. Luckily, passing the user information in an additional SQL call is exactly what we're missing. Because the database security system is inspecting all SQL calls made to the database, it can look for the certain procedure call within the SQL and can extract the value representing the application user. This extracted value is associated with any SQL call made after this call within that session—so long as no additional call is made to set another application user ID (to imply that the session is now "owned" by another application user). In this case the security system has the following information about the call:

- The request is coming from 192.168.1.168.
- The request is being made on 10.10.10.5.
- The database login name is APPSRV.
- The command being issued is UPDATE.
- The database object being touched is EMPLOYEE.
- *The application user identifier.*

Using this information you can then go ahead and define a rule, as shown in Figure 6.2, to alert you whenever a DML command on the EMPLOYEE table comes from, for example, application user 999.

The methods shown are applicable to every application and every database, but they are based on proprietary handling of the application user ID and they may require a change at the application level. In some cases, the database may have built-in capabilities for passing such identifiers, and if you're really lucky (and yes, this is a long shot) the application may already be using such built-in capabilities. An example is the CLIENT_IDENTIFIER attribute supported by the Oracle database.

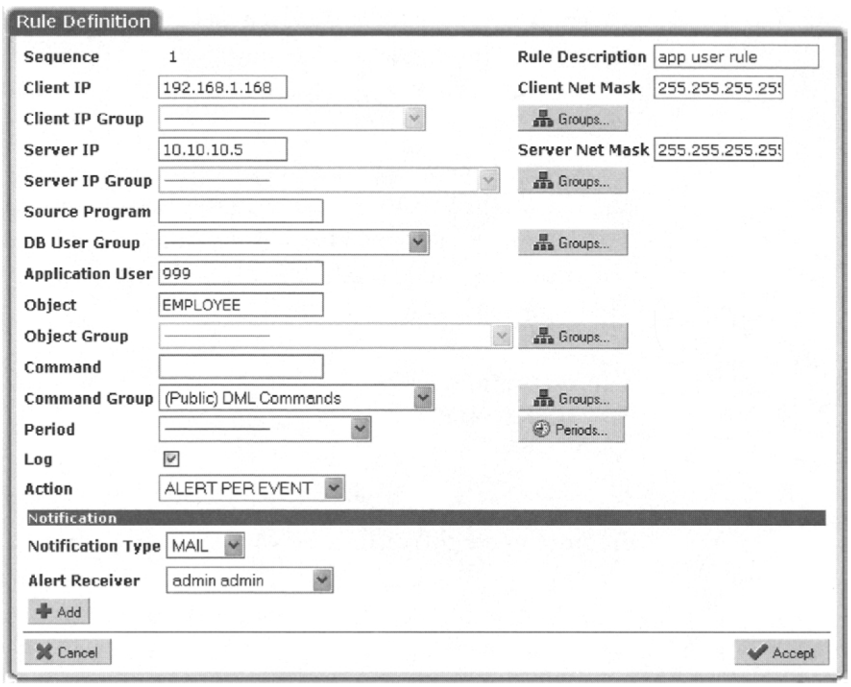


Figure 6.2 *Database access rule based on application user as implemented within an external security layer.*

CLIENT_IDENTIFIER is a predefined attribute of Oracle's built-in application context namespace USERENV that can be set using the DBMS_SESSION interface. This interface allows you to associate a client identifier with an application context, and Oracle keeps this information as a global mapping within the SGA.

The simplest way to use this identifier is through the built-in USERENV namespace, independently from the global application context. You can use this only if you are using an OCI driver (including thick JDBC). In this case the application layer can set the identity of the application user for use within the database using built-in OCI functions. When the application starts making calls on behalf of a user ID of "999," it can use the `OCIAttrSet` function as follows:

```
OCIAttrSet(session, OCI_HTYPE_SESSION,
       (dvoid *)"999", (ub4)strclen("999"),
       OCI_ATTR_CLIENT_IDENTIFIER,
       OCIError *error_handle);
```

If you are using a thick Oracle JDBC driver, you can use the encapsulating methods `setClientIdentifier` and `clearClientIdentifier`. After you call `getConnection` and receive the connection from the pool, call `setClientIdentifier` to let the database know that any statements sent to the database within the session are now made on behalf of the application user. When you're done, call `clearClientIdentifier` before surrendering the connection back to the pool.

A more general approach uses global application contexts supported by the DBMS_SESSION interface. In this case you can not only align the user models but also assign additional attributes, which can be used within your database code. The DBMS_SESSION interfaces available for setting (and clearing) contexts and identifiers are:

```
SET_CONTEXT
SET_IDENTIFIER
CLEAR_IDENTIFIER
CLEAR_CONTEXT
```

In order to use this technique, you first need to create a global context:

```
CREATE CONTEXT sec USING sec.init ACCESSED GLOBALLY
```

Now you can start assigning additional attributes that will be available and that can be used once you set the user identity within the database. For example, if you want to assign a "TOP SECRET" security clearance to be associated with an application user, you can execute:

```
DBMS_SESSION.SET_CONTEXT('sec', 'clearance', 'TOP SECRET',
'APPSRV', '999')
```

In this case APPSRV is the login used by the application server to sign onto the database. This is the username shared by all connections within the connection pool, and 999 is the unique identifier of the application user. You can make the context available for any database login by using:

```
DBMS_SESSION.SET_CONTEXT('sec', 'clearance', 'TOP SECRET',
NULL, '999')
```

At this point the application server can retrieve the connection from the pool and set the application user identifier using a single additional SQL call:

```
begin
  DBMS_SESSION.SET_IDENTIFIER('999');
end;
```

As an example, if a servlet running within a J2EE server needs to make database queries, it can follow these steps:

1. Retrieve the user identifier using the `getUserPrincipal`
2. Get the connection from the pool
3. Set the identifier within the context
4. Perform the database operations
5. Clear the identifier
6. Close the connection

Sample code for this sequence is shown as follows:

```
1->    String identifier = request.getUserPrincipal().getName();
       InitialContext ctx = new InitialContext();
       DataSource ds = (DataSource)ctx.lookup("java:/comp/env/oracle");
```

```
2->    Connection conn = ds.getConnection();
3->    PreparedStatement stmt =
              conn.prepareCall("begin dbms_session.set_identifier(?); end;");
       stmt.setString(1, identifier);
       stmt.execute();
       stmt.close();
4->    // Run application queries here
5->    PreparedStatement stmt =
              conn.prepareCall("begin dbms_session.clear_identifier(); end;");
       stmt.execute();
       stmt.close();
6->    conn.close();
```

Any code running within the database can now extract the security clearance using:

```
SYS_CONTEXT('sec', 'clearance')
```

Note that this call will first look at the current identifier and then use it to extract the correct value associated with this identifier. You can assign any number of attributes to be linked with the application user identifier—attributes that can help you better secure and limit what the application can access and how. For example, you can set both an attribute for security clearance as well as an attribute defining whether access is allowed outside of normal business hours:

```
DBMS_SESSION.SET_CONTEXT('sec', 'clearance', 'TOP SECRET',
'SCOTT', '999')
DBMS_SESSION.SET_CONTEXT('sec', 'off_hours_allowed', '1',
'SCOTT', '999')
```

This facility is more flexible than using the OCI's client identifier mechanism for several reasons: (1) you have more options and better control; (2) because this simply uses an additional SQL call, it is not limited to OCI or thick JDBC—it will run using any driver; and (3) this method can be used with an external security system. Moreover, using an external security layer with this facility is actually simpler to implement than using internal `SYS_CONTEXT('sec', 'clearance')` calls because you do not make changes to your database code and because you can support any query, whereas `SYS_CONTEXT('sec', 'clearance')` is mostly useful within stored procedures. If you do not want to change your database code but still prefer doing granular access control within the database (as opposed to an external

system), your database needs to support row-level security, as described in the next section.

6.2 Use row-level security (fine-grained privileges/ access control)

Let's continue with the topic of using the application user to implement better database access control—this time within the database engine. One of the advanced security features available in many databases is that of row-level security. The vendors have various names for this feature: Oracle calls it Virtual Private Database (VPD)/Fine-Grained Access Control (FGAC). DB2 currently only supports this feature on z/OS (i.e., mainframe) and calls it Multi-Level Security (MLS). SQL Server only supports this feature in SQL 2005 and calls it Fine-Grained Privileges. Sybase ASE also calls it Fine-Grained Access Control—feature introduced in ASE 12.5. These options are not fully equivalent in terms of functionality, but in all cases they allow you to implement row-level security. Using row-level security is generally a good idea when you need to have fine-grained access control, so this is a good technique to know. Furthermore, some of the databases allow you to use this feature to implement application user-based access control, so it fits right in with the topic of this chapter.

Let's start by looking at DB2's MLS and then move on to Oracle's VPD. After reviewing VPD, you'll complete the example started in the previous section with Oracle's context mechanism and see how to use VPD/FGAC to implement application user-based access control within the database. Even if your environment is not DB2 or Oracle, you should understand these concepts; they will probably be relevant to your environment either today or in the near future.

DB2 UDB 8 Multi-Level Security (MLS) is available for z/OS V1R54 systems and is based on the Resource Access Control Facility (RACF) (and specifically on the SECLABEL feature of RACF). For non-IBMers, z/OS means mainframe. For us non-mainframe people, let's do a two-minute review of RACF.

RACF was originally developed by IBM in 1976 and is still being used to manage security within mainframes. RACF has evolved and has been greatly enhanced over the years and has even been moved off the mainframe to other environments. RACF manages user authentication, data access authorization, journaling, DES encryption, and many other security features. IBM mainframes are arguably the most secure computing environments out there—and a lot of that is due to RACF.

One of the features supported by RACF is security labels (SECLABEL). RACF allows you to associate a security label with every user profile. These can then be used by RACF to compare the security label of the user with the security label assigned to a resource. Labels are ordered through relationships—a label can be equivalent to another, can dominate it, or can be less than another (reverse dominate). The ordering relationship is completely flexible, allowing you to represent pretty much any type of security hierarchy. Label security is discussed further in Section 6.3.

MLS in DB2 UDB for z/OS uses RACF to implement row-level security. If you want to implement row-level security for a table, you first need to add a column that will serve as the security label column. Whenever data is added to a table (e.g., using INSERT), the security label for the added row is set to the SECLABEL taken from the user profile for the user making the INSERT. In the same way, when you try to access a record, your SECLABEL is compared with whatever is stored within the security label column, and access is allowed only if your SECLABEL dominates the security label of the row you are trying to access.

A second security feature in DB2 was specifically built for WebSphere application servers, and while it does not support precisely the type of application user-based access control described in the previous section, it is somewhat related. DB2 UDB 8 on z/OS has four special registers (shown in Table 6.1) that are set by the client when initiating the connection. These are set by the DB2 JDBC driver used from a WebSphere application server. You can use the client user ID and/or the application name to enhance your security policy and/or view management within the database. Unfortunately, the user identifier is only set at connection time and does not change when the connection is used within another application session, and therefore cannot be used for fine-grained access control. However, future versions of DB2 for z/OS will include this functionality.

Table 6.1 *User identification registers in DB2 UDB 8 for z/OS*

Register Name	Description
`CLIENT_ACCTNG`	Accounting/journaling
`CLIENT_APPNAME`	Application name initiating connection
`CLIENT_USERID`	Used identification for the connection
`CLIENT_WRKSTNNAME`	Name of the workstation initiating the connections

Next let's look at Oracle's Virtual Private Database (VPD) and how it merges row-level security with application user information to fully support application user-based access control. VPD brings together server-enforced fine-grained access control (FGAC) by using the application context mechanism. VPD supports the automatic addition of additional predicates to every SQL statement issued. By allowing this predicate to be based on application contexts, which can be used to set the application user identifier, these additional predicates can achieve precisely the behavior we want.

VPD enforces fine-grained security on tables, views, and synonyms. Security policies are attached directly to these database objects and are automatically applied whenever a user accesses these objects. There is no way to bypass this added security once the policy has been activated; any access to an object protected with a VPD policy is dynamically modified by the server by adding potentially more limiting predicates to SELECTs, INSERTs, UPDATEs, INDEXs, and DELETEs. It's a very flexible mechanism: you can define functions that return the predicates that will be added and implement *any* kind of access control mechanism you desire.

VPD has two parts: the policy defining the function that returns the predicate and the runtime that adds the predicate to every SQL. Let's start with what the runtime does. Assume that we are accessing the EMP table:

```
Name                                      Null?    Type
----------------------------------------- -------- ----------
EMPNO                                     NOT NULL NUMBER(4)
ENAME                                              VARCHAR2(10)
JOB                                                VARCHAR2(9)
MGR                                                NUMBER(4)
HIREDATE                                           DATE
SAL                                                NUMBER(7,2)
COMM                                               NUMBER(7,2)
DEPTNO                                             NUMBER(2)
```

Assume that you want to build a security policy that defines that users should only be able to view data about people within their own department. If I am a user who belongs to the research department (DEPTNO=20) and I try to get the data using:

```
SELECT * FROM EMP;
```

then the VPD runtime will retrieve the predicate from the security policy and make sure that the query that is really executed is:

```
SELECT * FROM EMP WHERE DEPTNO=20;
```

This is all done transparently and without my knowledge, so effectively I truly have my own (virtually) private database.

In order for VPD to work, it needs to get the predicate from the security policy; this is where FGAC comes in. FGAC allows you to attach a security policy to tables, views, and synonyms. First, you need to create a PL/SQL function that returns the predicate (as a string) that will be used to restrict the queries:

```
create or replace function get_dept_id
(
       p_schema_name in varchar2,
       p_table_name in varchar2
)
return varchar2
is
       l_deptno number;
begin
       select deptno
       into l_deptno
       from scott.emp
       where empno = sys_context('app_ctx', 'app_userid');
       return 'deptno = ' || l_deptno;
end;
```

What this function does is the following:

1. It gets an application user ID from an application context (this context must already be defined as described in the previous section). In this case the application user ID is precisely the employee ID maintained in table EMP.
2. It selects the department number of this employee/application user. Assume in my case that this is department 20.
3. It returns the string `deptno = 20`.
4. This predicate is then added to the select statement by the VPD runtime as discussed.

The last thing left to do is to define the security policy that associates this function (called a policy function) with the EMP table. This is done using `add_policy` within the row-level security package:

```
begin
     dbms_rls.add_policy
     (
         object_schema       => 'APPSRV,
         object_name         => 'EMP',
         policy_name         => 'EMP_POLICY',
         policy_function     => 'GET_DEPT_ID',
         function_schema     => 'APPSRV,
         statement_types     => 'SELECT,UPDATE,INSERT,DELETE',
         update_check        => true,
         enable              => true
     };
end;
```

So now whenever anyone issues `SELECT * from EMP` the VPD runtime will see that there is a policy associated with EMP, call the policy function, which will return (in my case) the string `deptno = 20` so that the statement that will really be executed will be `SELECT * FROM EMP WHERE deptno = 20.`

Both VPD and FGAC have many features that you can exploit to implement almost any type of access control. These features are beyond the scope of this chapter; for more information, see Chapter 13 in the *Oracle 10g Database Security Guide* or in an article by Arup Nanda titled "Fine Grained Access Control" available at www.proligence.com/nyoug_fgac.pdf.

6.3 Use label security

The "bible" of all information security is a U.S. Department of Defense (DoD) standard titled "Trusted Computer System Evaluation Criteria" carrying the designation DoD 5200.28-STD. The document dating August 1983 (with a revision from December 1985) is also nicknamed "the Orange Book," and although it is quite old, it is still considered the origin of many security requirements even today. This is perhaps because the DoD and agencies such as the National Security Agency (NSA), Central Intelligence Agency (CIA), and so on have some of the most stringent security requirements.

Among the many concepts introduced and mandated by the Orange Book is the topic of security labels. If you have ever been in any military

organization or have worked with such an organization, you know that any document is marked with a classification such as Confidential, Classified, Top Secret, and so on. These security labels are a core piece of security in that any piece of information is labeled with its clearance level so that at any point in time anyone can review whether an individual can have access to the information (based on clearance level levels assigned to individuals). The Orange Book mandates this labeling for any type of information and mandates that this labeling be a part of the security policy defined within information systems, including data stored in databases. More specifically, the following extracts from the Orange Book give you an idea of what may be required of you in such an environment (TCB stands for Trusted Computer Base and is the component of the system responsible for security):

> **Requirement 1**—SECURITY POLICY—There must be an explicit and well-defined security policy enforced by the system. Given identified subjects and objects, there must be a set of rules that are used by the system to determine whether a given subject can be permitted to gain access to a specific object. Computer systems of interest must enforce a mandatory security policy that can effectively implement access rules for handling sensitive (e.g., classified) information. These rules include requirements such as: No person lacking proper personnel security clearance shall obtain access to classified information. In addition, discretionary security controls are required to ensure that only selected users or groups of users may obtain access to data (e.g., based on a need-to-know).
>
> **Requirement 2**—MARKING—Access control labels must be associated with objects. In order to control access to information stored in a computer, according to the rules of a mandatory security policy, it must be possible to mark every object with a label that reliably identifies the object's sensitivity level (e.g., classification), and/or the modes of access accorded those subjects who may potentially access the object.
>
> **Labels**—Sensitivity labels associated with each subject and storage object under its control (e.g., process, file, segment, device) shall be maintained by the TCB. These labels shall be used as the basis for mandatory access control decisions. In order to import nonlabeled data, the TCB shall request and receive from an authorized user the security level of the data, and all such actions shall be auditable by the TCB.

> **Label Integrity**—Sensitivity labels shall accurately represent security levels of the specific subjects or objects with which they are associated. When exported by the TCB, sensitivity labels shall accurately and unambiguously represent the internal labels and shall be associated with the information being exported.

Label security is an advanced security option and one that you will probably need to be familiar with in a military or agency-type environment. Still, it is always useful to understand such advanced security methodologies because they may come up elsewhere; for example, I was recently introduced to a project within an investment bank with a focus on data classification. More important, label security is usually viewed as an advanced implementation using row-level security and granular access control. In fact, you can think of label security as the addition of another column to every table in your schema—a column that will house a classification label for every record. You can then use row-level security to ensure that a user with a Secret classification will be able to access rows with Classified or Secret labels but not those that have a Top Secret label.

Most of the database vendors can offer functions supporting label security through the use of row-level security/fine-grained access control. Both DB2 UDB 8 for z/OS and Oracle support label security—DB2 through the SECLABEL feature in RACF, and Oracle through an advanced offering called Label Security that is available as part of the Enterprise Edition. Oracle has a packaged label security solution that is implemented using Oracle's VPD and uses sensitivity of data to implement fine-grained access control. As shown in Figure 6.3, it works by comparing sensitivity labels assigned to rows with label authorization assigned to users.

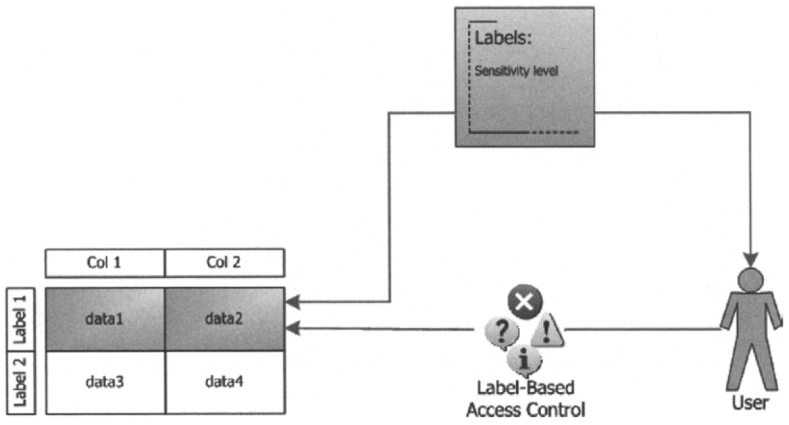


Figure 6.3 *Label-based access control in Oracle Label Security.*

At a high level, a label represents a sensitivity level. At closer look, it has a few elements and comprises several components. Note that labels do not have to incorporate every one of these components. Only the sensitivity level is mandated, but these additional components allow you to finely tune data-level security. Labels can include:

- A sensitivity level that is usually one of a hierarchy of values (i.e., data that is top secret is by nature also classified)
- A category or compartment used to segregate data; compartments are used when data security is based on a "need-to-know basis"
- A group component that can be used to record data ownership
- An inverse group component that can be used to control dissemination of information

The inverse group component differs from the group component in that it defines a set of groups to which users must be assigned before they can access the data. As an example, a row may be labeled with the groups NAVY, AIR FORCE, meaning that any user belonging to *either* the NAVY or the AIR FORCE groups (and having the appropriate sensitivity level) can access the information. However, if you label a row with the inverse groups NAVY, AIR FORCE then only users assigned to *both* of these groups can access this data; a user belonging to only the NAVY group (even with the right sensitivity level) will not be able to see this data.

Label security is available through custom installation of Enterprise Edition. In Oracle 8i this was only available for Solaris, but as of Oracle 9i this is available on all platforms. Once installed you need to use the database configuration tool to create the necessary data dictionary objects for label security. The initial database administrator account for label security is called LBACSYS, and you will need to unlock it after the installation. You can administer label security by issuing commands in SQL*Plus (or other tools) logged in as LBACSYS or by using the Policy Manager (available in the Integrated Management Tools submenu on Windows or as the `oemapp` utility in UNIX). Whenever you create a policy, you will have to specify a column name; this column will be appended to the application table but can be hidden from describe statements for better security. You should also always create a bitmap index on the label security column; the percentage of the unique labels compared

to the number of data rows will almost always be extremely low, making it an ideal candidate for a bitmap index.

Finally, before leaving the topic of label security, be aware that using these advanced security features absolutely does not mean that you can avoid the basics already discussed in previous chapters. For example, in October 2001, Oracle issued Security Alert #21, which was a mandatory security patch for Oracle Label Security. This patch (2022108) for Oracle 8.1.7 on Solaris fixes three vulnerabilities (1816589, 1815273, and 2029809), allowing users to gain a higher level of access than authorized by their labels.

6.4 Integrate with enteprise user repositories for multitiered authentication

The Lightweight Directory Access Protocol (LDAP) is an open industry standard that defines methods for accessing and updating information in a directory. A directory is a database that stores typed and ordered information about user objects (e.g., IBM's SecureWay LDAP server has an embedded DB2 UDB database, and Oracle Internet Directory [OID] is built on top of an Oracle database). An LDAP directory is optimized for read performance, which means it assumes that the user data will be read far more than it will be changed. LDAP servers base their naming models on either the X.500 methodology or the DNS naming model. The X.500 methodology sets the root of the directory to an organization and has a suffix like `o=myCompany, c=us`. The DNS model uses the domain name as the suffix like `dc=myCompany.com`. As an example, IBM's SecureWay uses an X.500-like methodology and Microsoft's Active Directory uses the DNS naming model.

Data in a directory is stored hierarchically in a Directory Information Tree (DIT) over one or more LDAP server(s). The top level of the LDAP directory tree is called the base Distinguished Name (DN) or a suffix. Each directory record has a unique DN and is read backward through the tree from the individual entry to the top level. The DN is used as a key to the directory record. For example, in Figure 6.4, Ron's entry would be accessed using `cn=Ron,ou=Development,dc=myCompany,dc=com`.

LDAP servers have become ubiquitous in the enterprise. In fact, they've become ubiquitous everywhere! On Windows environments this is all-encompassing, because Microsoft Active Directory server is part of the Windows 2000 system, and Windows 2000 and 2003 use Active Directory as the authentication mechanism for Windows. More important, all of the

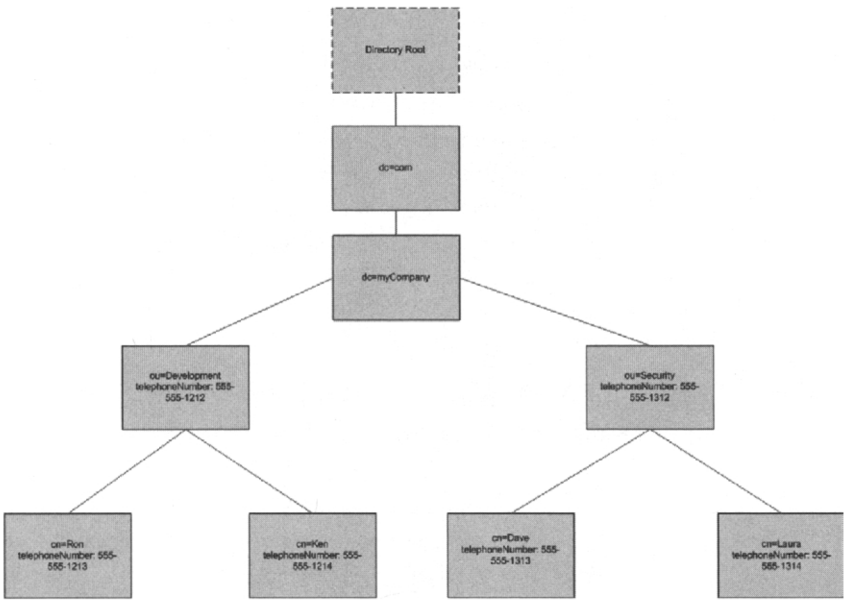


Figure 6.4
A sample directory naming structure.

main database vendors have interfaces to all LDAP servers—sounds like an impossible dream-come-true, doesn't it? This is because LDAP is an industry standard that defines the protocol allowing the database to talk with the LDAP server. Some examples of common integrations that are often seen include the following:

- DB2 UDB on Windows integrates with Active Directory. DB2 UDB 8 can be configured to integrate instance and database objects within Active Directory. Note that in this case all authenticated users in the domain have read permission to the DB2 instance and any database object registered with Active Directory (i.e., both authentication models and authorization models are integrated).
- Not surprisingly, Microsoft SQL Server is integrated with the Active Directory. In fact, if SQL Server uses Windows Authentication (the preferred configuration—see Chapter 4), then SQL Server really uses the Windows operating system for authentication, which, as mentioned, uses Active Directory. In addition to authentication, the integration between SQL Server and Active Directory includes the following:

  - SQL Server registers itself within Active Directory in order to support discovery services. You can register instances of SQL Server, databases, replication publications, and Analysis servers in the Active Directory.
  - SQL Server tools provide a dialog box that supports browsing for replication publications registered in the Active Directory.
  - When registering replication publications in the Active Directory, the Create Pull Subscription Wizard allows users to search for registered publications in the Active Directory.
  - The `sp_ActiveDirectory_Obj` stored procedure supports programmatically registering databases from T-SQL scripts or from applications.
  - Multiple SQL Server instances that are integrated with Active Directory create an environment that supports security account delegation. This means you can connect to multiple servers, and with each server change, you retain the authentication credentials of the original login. For example, if I sign on to the finance server as user FINANCE\ronb, which then connects to the sales server, then the second server knows that the connection security identity is FINANCE\ronb.
- Oracle is often integrated with Oracle's LDAP server Oracle Internet Directory (OID) but is just as often integrated with the iPlanet LDAP server and Novell's NDS. You can create a user within Oracle that is identified with an LDAP name by using:

```
CREATE USER ronb
IDENTIFIED GLOBALLY AS 'cn=ronb,ou=mycompany,c=us'
```

Note that while integration with an LDAP server is effective for enterprise authentication and authorization, it can also be used for storing information that would otherwise be stored in database configuration files. As an example, network connectivity information that Oracle usually stores in the `tnsnames.ora` file can also be stored in Active Directory.

Let's look at an example of how an integrated environment can help in preserving the user identity end-to-end. The example is based on integrating with Oracle Internet Directory (OID). In this case the database accepts the connection from the application server but also uses the additional user information from the application server as a key to user information stored within OID. The database will access OID to retrieve information such as roles and shared schemas that should be associated with the user credentials.

Historically, Oracle has merged the concepts of users and schemas, but in essence a schema is a logical container of database objects, whereas a user is someone who signs on to the database to do work. These two concepts must be separated once you move user management out of the database; after all, different databases may have the same users, and this should not mean the same access controls and the same schemas. Oracle 9i allows you to do this. The first step is to create a shared schema, which can be shared by a large number of OID-managed users:

```
CREATE USER SHARED_SCHEMA IDENTIFIED GLOBALLY AS '';
```

Now that we have a schema definition, let's define a role that will be used to associate application users defined within OID to permissions. Roles are important because there are usually many application users—sometimes too many. The best association is therefore through roles. First, we'll define a role in the database and then attach it to user profiles in OID. To create the role within the database:

```
CREATE ROLE APP_USER_ROLE1 IDENTIFIED GLOBALLY;
GRANT CREATE SESSION TO APP_USER_ROLE1;
```

Next, open the Enterprise Security Manager and go to Enterprise Domains->Enterprise Roles and click Add. This allows you to add an enterprise role; specify the role as APP_USER_ROLE1 and give the database's name. This should reflect a business-level function to represent sets of permissions, and note that you can have a single enterprise role that is mapped to many role/database pairs. This role can then be associated with users defined within the DIT by using the third tab in the dialog used for creating or editing a user profile, the Enterprise Roles tab.

At this point you will want to attach the schema created earlier to a set of users managed within OID. You can do this by using the Enterprise Security Manager tool, and you can assign it based on any level within the DIT (i.e., per a set of users or by assigning it to a complete subtree). In either case, this is done using the Database Schema Mapping by navigating to Enterprise Domains->Oracle Default Domain, clicking the Database Schema Mapping, and adding a mapping between the schema and a directory entry within the DIT. The schema is now associated with an application user or a set of application users.

At this point you are all set. When you access the application server connected to the same OID, the authentication stage associates you with a

node in the DIT. This node is then associated with the role and the schema, so that when the application server accesses the database it uses SHARED_SCHEMA and the permissions are defined based on APP_USER_ROLE1.

Oracle proxy authentication is closely related to this usage of OID and roles. This feature allows the application user to be communicated in addition to the database login name over a connection initiated using OCI or thick JDBC. When using proxy authentication, the end-to-end identification process is as follows:

1. The user authenticates with the application server. This can be done using a username and password or through the use of an X.509 certificate by means of SSL.
2. The application server uses OID to authenticate the user credentials and gets the DN for the user profile.
3. The application server connects to the database using proxy authentication. In this process it passes not only the username and password used to sign onto the database, but also the DN to the database.
4. The database verifies that the application server has the privileges to create sessions on behalf of the user.
5. The database gets user information from OID using the DN.

Proxy authentication is a useful feature, and you would think that a lot of what you learned in this chapter is unnecessary given proxy authentication. This is not true, mainly because the association between application users is not dynamic. The first limitation is point number 4, listed previously. In order for proxy authentication to work, you need to allow the application server to connect on behalf of the user using GRANT CONNECT:

```
ALTER USER RONB
       GRANT CONNECT THROUGH APPSRV;
```

This requirement is difficult to maintain for a large number of users, and many of the techniques you learned earlier in this chapter are often more scalable in the long run. The second issue is best seen by looking at what a connection within the application code would look like (in this case you are using a thick JDBC driver):

```
String userName;

InitialContext initial = new InitialContext();
OracleOCIConnectionPool ds =
   (OracleOCIConnectionPool)initial.lookup("jdbc/OracleOciDS");

oracle.jdbc.OracleConnection conn = null;

Properties p = new Properties();
p.setProperty(OracleDataSource.PROXY_USER_NAME, userName);
conn = ds.getProxyConnection(OracleDataSource.PROXYTYPE_USER_NAME, p);
```

Note that proxy authentication occurs during the connection initiation. This means that while you can pass an application user, you can only do this once and you cannot dynamically modify the application user on behalf of which SQL is issued. Therefore, proxy authentication may be another trick you may want to know about, but it cannot really be used to align with the application user model.

Finally, Sybase ASE has a slightly different feature that should not be confused with proxy authentication in Oracle. In Sybase this is called proxy authorization, and it allows you to impersonate a user with another. It would seem to be an effective way to implement the dynamic change of application user, but unfortunately it requires that all users be defined at the database level, which is not always realistic. The syntax to change the authorization credentials to the user ronb for the session is:

```
set proxy ronb
set session authorization ronb
```

You have to first enable the original login name for impersonation:

```
grant set proxy to rona,ronb
grant set session authorization to rona,ronb
```

As long as all application users are defined as users in the master database, you can use this mechanism to implement dynamic user-to-session association.

6.5 Integrate with existing identity management and provisioning solutions

Because of the complexity involved with security features such as authentication and authorization in environments including many applications and

information sources, a new category of products has emerged in the past few years. These products manage repositories of users and their profiles and implement security policies for authenticating and authorizing access based on identifying users and mapping them to static or dynamic roles. These tools allow you to manage a complex entitlement model that spans multiple applications and sources. Perhaps the most well-known issue that is handled by these tools is that of single sign-on (SSO). A good SSO environment means that once users have authenticated with the system once, they will not be asked to authenticate again even when they traverse application boundaries. A bad SSO implementation (or no SSO implementation) will constantly ask users for a username and a password, every time they access a separate application. This, together with the fact that complex enterprise environments may include tens or hundreds of applications that users may need to access, is the reason why security and identity management tools have been highly successful in the past few years and why a new category of products has emerged. The main functions supported by security and identity management tools are the following:

- Support for heterogeneous environments and servers within a single and consistent security model
- Ability to manage virtually any resource, including applications and databases
- Central management of security information
- Central management of user profiles
- Configurable session management (e.g., session timeouts)
- Full support for user provisioning
- Definition of security and access control rules based on users, roles, dynamic roles, and even through rules that match data in a user context with conditions that determine whether the user should have access to a particular resource
- Support for personalized Web and portal content using a consistent rule set regardless of the underlying provider
- Policies and personalization based on IP addresses
- Enhanced security attributes
- Multigrained security (i.e., the ability to define fine-grained access control on some resources and coarse-grained access at the same time)

- Support for single-sign on
- End-to-end handling of security credentials and security policies

A simple example may convince you much more than a long laundry list of functions and features. I've had a couple of experiences with companies that have pretty secure database environments and yet because it takes almost a week to set up new accounts for new employees or consultants, they often start working using "borrowed" database logins, so all good security intentions practically go out the window. Similar examples involve people who are no longer with the company. How many of you have accounts defined within a production system that are no longer used or that you are not sure whether they are used? This topic is broader than database security and is the topic of user provisioning, which is an important piece of security and identity management.

However, if you are managing a complex and dynamic user environment and especially if you have managed to align closer to the end user model, then you may select to integrate your database environment with a security and identity management solution. If you do, don't underestimate the added complexity that this adds and don't underestimate the time you will have to invest.

6.6 Summary

In this chapter you saw that granular access control can only be achieved through aligning the application user model with the database security system (which can be internal within the database engine or implemented as an external security system). You saw why this is important, what methods exist to communicate the application user information to the database security system, and how to use this information to implement granular access control. You also saw some broader issues pertaining to user directories and identity management.

I want to make one brief comment about the techniques you saw in this chapter. Many of the methods shown here are proprietary and exist on some databases and not in others. Even when two vendors support the same basic concept, this is done differently. Another example for this non-standard implementation is the fact that some of the examples I showed you in Oracle or DB2 will only work with a thick JDBC driver or OCI, because the APIs depend on proprietary techniques. This will change over time. This topic is of primary importance for good database security, and

more techniques are being built as you read these words. In fact, I know of Java work being done at IBM (which will then be submitted for acceptance to Sun) to support granular access control in a J2EE environment. Because this is such an important topic, I hope this will happen sooner rather than later.

The next chapter goes back to the core database engine to discuss some of the extensions and rich functions that modern databases can do other than simple persistence and data lookup, and what pitfalls you should be aware of when you use these advanced functions.

Computers / Data management / Security

Implementing Database Security and Auditing

A Guide for DBAs, information security administrators and auditors

Ron Ben Natan

Today, databases house our "information crown jewels", but database security is one of the weakest areas of most information security programs. With this excellent book, Ben-Natan empowers you to close this database security gap and raise your database security bar!

—**Bruce W. Moulton,** CISO/VP, Fidelity Investments (1995 - 2001)

It's been said that everyone has their 15 minutes of fame. You certainly don't want to gain yours by allowing a security breach in your database environment or being the unfortunate victim of one. Information and Data are the currency of On Demand computing, and protecting their integrity and security has never been more important. Ron's book should be compulsory reading for managing and maintaining a secure database environment.

—**Bob Picciano,** VP Database Servers, IBM

Let's start with a simple truth about today's world: If you have a database and you make it available to customers, employees, or whomever over a network, that database will be attacked by hackers–probably sooner rather than later. If you are responsible for that database's security, then you need to read this book. No other single source covers all of the many disciplines and layers involved in protecting exposed databases, and it especially shines in synthesizing all of its concepts and strategies into very practical and specific checklists of things you need to do. I've been an Oracle DBA for 15 years, but I'm not embarrassed to admit that five minutes into Chapter One I was making notes on simple measures I had overlooked.

—**Charles McClain,** Senior Oracle DBA
North River Consulting, Inc.

This book is about database security and auditing. You will learn many methods and techniques that will be helpful in securing, monitoring and auditing database environments. The book covers diverse topics that include all aspects of database security and auditing - including network security for databases, authentication and authorization issues, links and replication, database Trojans, etc. You will also learn of vulnerabilities and attacks that exist within various database environments or that have been used to attack databases (and that have since been fixed). These will often be explained to an "internals" level. There are many sections which outline the "anatomy of an attack" before delving into the details of how to combat such an attack. Equally important, you will learn about the database auditing landscape–both from a business and regulatory requirements perspective as well as from a technical implementation perspective.

- Useful to the database administrator and/or security administrator–regardless of the precise database vendor (or vendors) that you are using within your organization
- Has a large number of examples–examples that pertain to Oracle, SQL Server, DB2, Sybase and even MySQL..
- Many of the techniques you will see in this book will never be described in a manual or a book that is devoted to a certain database product
- Addressing complex issues must take into account more than just the database and focusing on capabilities that are provided only by the database vendor is not always enough. This book offers a broader view of the database environment– which is not dependent on the database platform–a view that is important to ensure good database security

Ron Ben Natan is CTO at Guardium Inc., a leader in database security and auditing. Prior to Guardium Ron worked for companies such as Intel, AT&T Bell Laboratories, Merrill Lynch, J.P. Morgan and ViryaNet. He holds a Ph.D. in the field of distributed computing from the University of Jerusalem. Ron is an expert on the subject of distributed application environments, application security and database security and has authored nine technical books and numerous articles on these topics.

Audience: DBA's, System and Network Administrators and Auditors

books.elsevier.com/digitalpress

ISBN:1-55558-334-2

90000

9 781555 583347