CHAPTER 8

Effective Auditing for Accountability

uditing is one of those not-so-exciting areas of security that everyone knows they should do but rarely ever does. There are many reasons for this. Some don't know what to audit; some don't know how to audit; some don't know what to do with the audit records once they have audited; and some believe the audit performance overhead is a penalty that doesn't justify the process.

This chapter explores each of these issues. You'll see why auditing is not only possible, but also invaluable. Manual ways to audit will be examined as well as looking at database standard auditing and the improved fine-grained auditing technology. You'll see how the audit records will show you who, what, where, when, and how. You have to determine why and may be able to based on the captured SQL. You'll also explore how auditing can be used as a tool to show how popular an application is, what features are used, and who is using them, and you can establish usage patterns over time. In fine-grained auditing, you'll learn how to control the audit fidelity as well as how to invoke an event handler.

Now a bit of philosophy on how auditing fits and why it's important. Later extensive code examples show various methods and aspects to auditing. You'll see that auditing is a complementary process of the security cycle, and when done effectively it can act as an invaluable tool in your security toolbox.

The Security Cycle

Before discussing *how* and *what* to audit, you need to understand *why* to audit. There is a natural cycle associated with security that begins with prevention, moves to detection, and finishes with response. This cycle is described very well in *Secrets and Lies: Digital Security in a Networked World* by Bruce Schneier (John Wiley & Sons, 2000). What is important about understanding this cycle is that it helps you understand that security isn't exclusively about access control.

Prevention, which includes access control, is the first process in the cycle. It describes all of the measures put in place to control who can do what and how they can do it. In the nondigital world, you see examples of this everywhere, such as locks on doors, electric fences, and security guards restricting access to buildings and building corridors. Many people refer to a system's security to mean only the preventative security measures. While this may seem intuitive, it is incorrect.

Let's look at a bank as a reference. The bank has security designed in from the beginning. The walls are reinforced brick and concrete. This prevents people from mounting a successful Exacto knife attack in which a burglar walks up to a building, usually a residential home, and slices through the siding and insulation, thus creating an instant portal into the structure. Inside the bank you see many other prevention measures such as a security guard, bulletproof glass, and a physical separation from customers, tellers, and money. The vault itself is constructed to prevent robbers from breaking the lock, the door, or slicing through the sides with a torch. The interesting part is that the vaults are not guaranteed to prevent access. They only slow access in attempts to prolong the robbery until the authorities arrive.

This brings us to the next step: detection. Detection can easily occur when the attack happens at a time when people are watching. A bank robbery during the midmorning will be detected by the people being robbed. If the robbery happens at night, on Sunday, or any of the numerous bank holidays, the detection may be more subtle. Enter motion detectors. These are excellent detection devices. It's clear that the motion detectors cannot prevent the robbery. They can only detect it.

Once a security breach is detected, a response is usually desired. In the case of a bank robbery, the police will respond by rushing quickly to the bank. In the case of a residential home burglary, the home's alarm system will notify the alarm company, who will then notify the police. They may first notify the homeowner; in the event that the detection is a false alarm, the homeowner will cancel the call to the police. False alarms happen more than actual break-ins. This is also true for computer systems.

Security begins with a clear and concise security policy. The policy is enforced through proper design and implementation complemented with varying access control capabilities. The lessons from security in the real world translate directly to the computer world. Invariably, something bad will happen. There is no way to build a computer application that is 100 percent secure. You have to rely on auditing for the detection mechanisms to support the overall database security. The response relies on the detection—the audit trail. The section, "Fine-Grained Auditing," shows that the responses can be expedited by an alerting capability.

Auditing for Accountability

Many people use auditing to provide the detection capabilities just described to support their overall database security efforts. Others audit to satisfy compliance with mandatory regulations, corporate or organizational policies, or contractual agreements. Generally, the common thread among all of these is user accountability. You want to ensure users are doing only what they are supposed to. You want to capture privilege abuse and misuse. Auditing allows you to hold your users responsible for their actions by tracking their behavior.

When a person commits a crime, a picture that has captured them in the act can serve as very compelling evidence. An important aspect to auditing is that it can also serve as a deterrent for would-be bad guys. If you know that someone is watching, you are less likely to do something bad. Database auditing can be thought of as the security cameras that capture the actions and diabolical deeds as they unfold. Note that the cameras and auditing may capture both good or expected actions as well as the bad and unexpected actions.

Auditing Provides the Feedback Loop

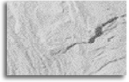
One thing worse than being robbed is not even knowing that you've been robbed. Auditing can help. The audit records are the means of capturing the robbery. If you're viewing the records and you see something has happened, then you can properly respond. Response may result in readjusting your access control mechanisms and expelling the user.

Two important things have to happen for effective auditing. First, you have to be auditing and auditing on the correct thing. This is analogous to saying you have to have security cameras turned on and facing the right direction. Second, you have to read and interpret the audit records.

The audit records act as a feedback mechanism into the prevention and access control mechanisms you've already established. They also play an important role in any investigative activities that occur either as a result of a breach or in anticipation of one. Without auditing, you may have no way of knowing whether your security is sound or whether your data has been read or modified by an unauthorized user.

Auditing Is Not Overhead

Some people feel that auditing introduces overhead which, when compared to the little value they derive from the audit records, is not worth it. This philosophy is flawed for several reasons.



NOTE
Auditing is not overhead.

Auditing all actions by all users on all data is not useful and *will* make a system perform miserably. Auditing must be selective and, when done correctly, it targets the correct data, processes, and users. This means the audit records are, by definition, very useful. The audit records also have to be reviewed and acted on when necessary. This means there is a regular process of reviewing, archiving, and deleting the unneeded records as appropriate. Chances are good that this isn't happening if the performance overhead issue is raised.

Auditing is an art that carefully balances capturing the needed audit records in a way that doesn't introduce detrimental effects on performance. People who are unfamiliar with how and when to audit may in fact end up with a slower performing system with so many audit records that an administrator can't distinguish the people doing legitimate work from the people with malevolent intentions.

The truth about auditing is that it is not overhead. If auditing is done for compliance reasons, or if it's just being done to complete the security cycle, it's necessary and a nonoptional aspect that must be incorporated into the system.

The same is true with other aspects of security. One could argue that access controls add overhead. Checks have to be performed to determine what users can see and do. All of this "overhead" is necessary to ensure the privacy and security of the data. Access controls are not blatantly discarded because there is some overhead associated with them. You accept the overhead associated with access controls, and the users accept it too. Auditing is simply the last phase in your security cycle, and it should not be discarded. You should accept auditing as the complementary security function that it is.

Audit Methods

Auditing takes many forms in today's applications; this section explores popular ways and the benefits and drawbacks of each. It'll discuss application server logs, application auditing, and trigger auditing, and then will conclude with Oracle's standard auditing and fine-grained auditing.

Note that these auditing techniques aren't mutually exclusive. They can and should, as necessary, be combined to build complementary layers of defense within the auditing realm. You will see that each possesses certain advantages and disadvantages. A composite of several auditing techniques will almost surely be the configuration you'll need.

Application Server Logs

Application server access logs and all associated log files for the application server and web server are often considered a basic form of auditing. These files vary in the amount of information they contain. In the general sense, they will list the resources that have been accessed, when and how the access occurred, and the result by way of a status code—for example, success, failure, and unknown.

The logs are very useful. The records contained in the log files are often direct indicators of the actions the user performed. For example, an update posted from a web page would have a distinct URL signature. As such, the user (or rather the user's IP address) can be audited as having invoked some program.

Application server logs are very useful in determining suspicious behavior. Denial of service (DoS) attacks may be evident. Many administrators actually use the logs to track a user's behavior as they navigate a website. This is similar to studying the shopping patterns of customers in department stores.

The challenge with using the application log files is that the information is indirect. It's only useful when combined with other data that links IP addresses to users and the URLs with actual programs. For this reason, application auditing is usually performed in addition to gathering server log files because it can directly audit who is acting on what.

Application Auditing

One of the most frequently used auditing techniques is application auditing, which is the built-in auditing services that are sometimes an actual part of a larger application. Regardless of the implementation language, application auditing is a natural to use, because it can meet most auditing requirements. It can achieve this lofty goal because the auditing is *manually programmed* into the application. As such, it's considered as extensible as the application and the developer's ability will allow.

Many people are quite familiar with application auditing. This technique is often seen when the developers don't understand or can't take advantage of the database auditing. Application auditing may also be the choice when the application wishes to remain database agnostic.

As users perform actions, the application code selectively audits. Various aspects of auditing are generally seen. User logins, data manipulations, and administration tasks can be easily audited. In mature applications, the auditing has been implemented as a service. The business objects within the application call different auditing services and different times to record different actions.

Application Audit Example

Consider an example procedure implemented in PL/SQL. The program could be invoked from an application running either within or outside the database. It may even be called from an application written in another language running outside the database.

This program will be explicitly called by the application at the appropriate time, which will vary from application to application. The example will invoke this auditing when the user performs an update to the SAL column of our table. First, create a copy of the SCOTT.EMP table:

```
scott@KNOX10g> CREATE TABLE emp_copy AS SELECT * FROM emp;

Table created.
```

Next, create the audit table, which is intended to serve as the audit table for all data manipulation on the EMP_COPY table:

```
scott@KNOX10g> CREATE TABLE aud_emp (
  2    username    VARCHAR2(30),
  3    action      VARCHAR2(6),
  4    empno       NUMBER(4),
  5    column_name VARCHAR2(255),
  6    call_stack  VARCHAR2(4000),
  7    client_id   VARCHAR2(255),
```

```
  8    old_value   VARCHAR2(10),
  9    new_value   VARCHAR2(10),
 10    action_date DATE DEFAULT SYSDATE
 11  )
 12  /

Table created.
```

The table will capture identifying information about the user performing the action, the action being performed (insert, update, or delete), new and old values, and what, if any, column is referenced. In the case of a delete, just say "ALL" for the column name.

Next, a procedure is created to perform the updates. This code could be embedded into an existing application's code or called from an existing application's code. You can also create a procedure that displays the formatted output of the data in the audit table as follows:

```
scott@KNOX10g> CREATE OR REPLACE PROCEDURE audit_emp (
  2    p_username     IN  VARCHAR2,
  3    p_action       IN  VARCHAR2,
  4    p_empno        IN  NUMBER,
  5    p_column_name  IN  VARCHAR2,
  6    p_old_value    IN  VARCHAR2,
  7    p_new_value    IN  VARCHAR2)
  8  AS
  9  BEGIN
 10  -- check data format and length
 11  -- not shown here
 12    INSERT INTO aud_emp
 13                (username,
 14                 action,
 15                 empno,
 16                 column_name,
 17                 call_stack,
 18                 client_id,
 19                 old_value,
 20                 new_value,
 21                 action_date)
 22         VALUES (p_username,
 23                 p_action,
 24                 p_empno,
 25                 p_column_name,
 26                 DBMS_UTILITY.format_call_stack,
 27                 SYS_CONTEXT ('userenv',
 28                              'client_identifier'),
 29                 p_old_value,
 30                 p_new_value,
 31                 SYSDATE);
 32  END;
 33  /

Procedure created.
```

```
scott@KNOX10g> -- create procedure to display audit trail records
scott@KNOX10g> CREATE OR REPLACE PROCEDURE show_aud_emp
  2  AS
  3  BEGIN
  4    FOR rec IN (SELECT   *
  5                    FROM aud_emp
  6                ORDER BY action_date DESC)
  7    LOOP
  8      DBMS_OUTPUT.put_line (   'User:       '
  9                            || rec.username);
 10      DBMS_OUTPUT.put_line (   'Client ID:  '
 11                            || rec.client_id);
 12      DBMS_OUTPUT.put_line (   'Action:     '
 13                            || rec.action);
 14      DBMS_OUTPUT.put_line (   'Empno:      '
 15                            || rec.empno);
 16      DBMS_OUTPUT.put_line (   'Column:     '
 17                            || rec.column_name);
 18      DBMS_OUTPUT.put_line (   'Old Value:  '
 19                            || rec.old_value);
 20      DBMS_OUTPUT.put_line (   'New Value:  '
 21                            || rec.new_value);
 22      DBMS_OUTPUT.put_line (   'Date:       '
 23                            || TO_CHAR
 24                                 (rec.action_date,
 25                                  'Mon-DD-YY HH24:MI'));
 26      DBMS_OUTPUT.put_line
 27                   ('------------------------------');
 28    END LOOP;
 29  END;
 30  /

Procedure created.
```

The code is simple as written. It's a good idea, however, to check the data types and data lengths prior to inserting (not done here, for brevity).

In this example, the auditing occurs by invoking the AUDIT_EMP procedure from another program when the user performs the actual update.

This popular design uses definer rights to help restrict access to actual database tables. Instead of allowing users to directly manipulate data, they have to invoke a procedure that performs the data manipulation task on the user's behalf. Inside the procedure, there may be some auditing code that captures various aspects of the operation. Perhaps the old and new values are captured, the user, and the time.

```
scott@KNOX10g> CREATE OR REPLACE PROCEDURE update_sal (
  2    p_empno   IN  NUMBER,
  3    p_salary  IN  NUMBER)
  4  AS
  5    l_old_sal  VARCHAR2 (10);
  6  BEGIN
  7    SELECT     sal
```

```
  8          INTO l_old_sal
  9          FROM emp_copy
 10         WHERE empno = p_empno
 11    FOR UPDATE;
 12    UPDATE emp_copy
 13       SET sal = p_salary
 14     WHERE empno = p_empno;
 15    audit_emp
 16      (p_username       => USER,
 17       p_action         => 'UPDATE',
 18       p_empno          => p_empno,
 19       p_column_name    => 'SAL',
 20       p_old_value      => l_old_sal,
 21       p_new_value      => p_salary);
 22  END;
 23  /

Procedure created.
```

For example, you want the user BLAKE to perform updates, but you don't need to grant him update on the table directly. Simply grant him execute on the preceding UPDATE_SAL procedure. Do this in addition to granting him the ability to query the EMP_COPY table:

```
scott@KNOX10g> GRANT EXECUTE ON update_sal TO blake;

Grant succeeded.

scott@KNOX10g> GRANT SELECT ON emp_copy TO blake;

Grant succeeded.
```

Now, as user BLAKE, you'll execute the procedure. Note that BLAKE has no idea the procedure is auditing the update. Our audit table has a column for the Client Identifier, which can be set to any meaningful value. By capturing this or any other application context in the audit table, you can obtain more information about the end user's context. Note the security caveats of using the Client Identifier described in Chapter 6.

The application could be setting the value explicitly. As an alternative shown next, a database logon trigger could have set the Client Identifier value transparently. Here's an example trigger that sets the Client Identifier to the user's connected IP Address:

```
sec_mgr@KNOX10g> CREATE OR REPLACE TRIGGER set_ip_in_id
  2    AFTER LOGON ON DATABASE
  3  BEGIN
  4    DBMS_SESSION.set_identifier
  5                          (SYS_CONTEXT ('userenv',
  6                                        'ip_address'));
  7  END;
  8  /

Trigger created.
```

To test your audit, connect as BLAKE, query to check the original data values, and then execute your update procedure:

```
blake@KNOX10g> SELECT empno, sal
  2    FROM scott.emp_copy
  3   WHERE ename = 'BLAKE';

     EMPNO        SAL
---------- ----------
      7698       2850

blake@KNOX10g> EXEC scott.update_sal(p_empno=>7698, p_salary=>3000);

PL/SQL procedure successfully completed.

blake@KNOX10g> COMMIT ;

Commit complete.

blake@KNOX10g> SELECT empno, sal
  2    FROM scott.emp_copy
  3   WHERE ename = 'BLAKE';

     EMPNO        SAL
---------- ----------
      7698       3000
```

Connecting back as SCOTT, you see the audit data:

```
scott@KNOX10g> EXEC show_aud_emp
User:        BLAKE
Client ID:   192.168.0.100
Action:      UPDATE
Empno:       7698
Column:      SAL
Old Value:   2850
New Value:   3000
Date:        Mar-24-04 13:34
-----------------------------
```

The call stack was also preserved for you. The call stack shows (reading from the bottom up) an anonymous PL/SQL block called the SCOTT.UPDATE_SAL procedure, which then called the SCOTT.AUDIT_EMP procedure:

```
scott@KNOX10g> COL call_stack format a50
scott@KNOX10g> COL username format a10
scott@KNOX10g> SELECT username, call_stack
  2    FROM aud_emp;
```

```
USERNAME   CALL_STACK
---------- ------------------------------------------------
BLAKE      ----- PL/SQL Call Stack -----
             object      line  object
             handle    number  name
           692097F4         1  anonymous block
           694CAF18        12  procedure SCOTT.AUDIT_EMP
           6945FAEC        15  procedure SCOTT.UPDATE_SAL
           693CEA5C         1  anonymous block
```

Benefits

One of the greatest benefits of application auditing is that it's inherently extensible. As security and auditing requirements evolve, application auditing can often be modified to meet these new and ever-changing requirements.

Not only can application auditing support many requirements, but also it controls *how* the auditing is done. This has benefits because applications in the application server may elect to audit to a file on the midtier or audit to a separate database, which would protect the audit records from the administrators of the production database. The auditing implementation can be based on anything. The previous example is only one possible method. Database tables are excellent for auditing because they provide the structure that facilitates the reporting that makes auditing useful. It's generally simple to create SQL reports on the audit data. Many questions such as, "What has the user SCOTT accessed in the last three days?" can be easily answered when the audit records are stored in database tables.

Another major motivator for application auditing is that all aspects of the application can be audited, not only the data access. If the application interfaces with multiple databases, a few flat files, and a web service, all of that can be audited in a consistent way.

Application auditing may also be done to help ensure database independence. To do this effectively, a service layer would be implemented that would separate the auditing interface calls from the actual audit implementation. While this may seem noble at first, the reality is that to get the most use of your investment, your applications should be exploiting as much database technology as possible. Why reinvent the wheel?

Finally, application auditing requires no knowledge of database auditing. Even if knowledge isn't the issue, the database auditing may provide little value if the application architecture doesn't support it. Consider an application that doesn't propagate the user's identity to the database. Database auditing wouldn't add much value, at least not for user-level auditing.

Drawbacks

After all those benefits, you might be tempted to rush right out and build in application auditing. Before you do, consider some of the following issues.

First, the programmatic nature of application auditing can be a drawback as well as a benefit. The audit code is just that—code. It's therefore subject to all the challenges that plague code, such as logic errors, bugs, and the tremendous cost of maintaining the code over time.

From the security angle, the real drawback occurs if the application is bypassed. If a user conducts a direct update on the table, the application auditing will not be done because the application has been circumvented. Applications, especially applications facing large communities of users, will be targeted and possibly hacked. As such, all the security, including the auditing, may be overthrown or at the very least, be in jeopardy.

This hints at another challenge in application auditing. The application has to know that it's supposed to call the auditing programs. One possible way to enforce this is to have the audit

program set a signal value in a user-defined application context. You could then create a row-level security policy using views or Virtual Private Database (VPD) that checks for this signal. If you don't understand this now, don't worry—we'll revisit row-level security implemented by views and VPD in Chapters 10 and 11. The point is, without a way to enforce the auditing, the auditing may not occur.

Trigger Auditing

Within the database, a very popular technique for auditing is to utilize database triggers. DML triggers will be explored: Oracle supports triggers for inserts, updates, and deletes. Oracle doesn't support SELECT triggers, but similar functionality can be achieved using fine-grained auditing—details on how to do this are in the "Fine-Grained Auditing" section. Trigger auditing provides transparency, allowing you to enable auditing without requiring application modifications. Applications don't have to be aware of the trigger auditing.

Trigger Audit Example

Auditing via triggers usually consists of writing to an auxiliary auditing table. Generally, the new and old data, along with some other useful information, is captured. Create the trigger to call the AUDIT_EMP procedure defined previously:

```
scott@KNOX10g> CREATE OR REPLACE TRIGGER update_emp_sal_trig
  2    BEFORE UPDATE OF sal
  3    ON emp_copy
  4    FOR EACH ROW
  5  DECLARE
  6  BEGIN
  7    audit_emp (p_username       => USER,
  8               p_action         => 'UPDATE',
  9               p_empno          => :OLD.empno,
 10               p_column_name    => 'SAL',
 11               p_old_value      => TO_CHAR (:OLD.sal),
 12               p_new_value      => TO_CHAR (:NEW.sal));
 13  END;
 14  /

Trigger created.
```

To test, perform an update on the table as the user BLAKE. For the update, BLAKE requires the update privileges on the table.

```
scott@KNOX10g> GRANT UPDATE(sal) ON emp_copy TO blake;

Grant succeeded.
```

BLAKE now performs a direct update giving everyone in department 20 a 10 percent raise.

```
blake@KNOX10g> UPDATE scott.emp_copy
  2     SET sal = sal * 1.1
  3   WHERE deptno = 20;
```

```
5 rows updated.

blake@KNOX10g> COMMIT ;

Commit complete.
```

Returning to SCOTT to view the audit data, you see the updates that occurred for each record:

```
scott@KNOX10g> EXEC show_aud_emp;
User:        BLAKE
Client ID:   192.168.0.100
Action:      UPDATE
Empno:       7369
Column:      SAL
Old Value:   800
New Value:   880
Date:        Mar-24-04 14:23
------------------------------
User:        BLAKE
Client ID:   192.168.0.100
Action:      UPDATE
Empno:       7566
Column:      SAL
Old Value:   2975
New Value:   3272.5
Date:        Mar-24-04 14:23
------------------------------
User:        BLAKE
Client ID:   192.168.0.100
Action:      UPDATE
Empno:       7788
Column:      SAL
Old Value:   3000
New Value:   3300
Date:        Mar-24-04 14:23
------------------------------
User:        BLAKE
Client ID:   192.168.0.100
Action:      UPDATE
Empno:       7902
Column:      SAL
Old Value:   3000
New Value:   3300
Date:        Mar-24-04 14:23
------------------------------
User:        BLAKE
Client ID:   192.168.0.100
Action:      UPDATE
Empno:       7876
Column:      SAL
```

```
Old Value:  1100
New Value:  1210
Date:       Mar-24-04 14:23
------------------------------
User:       BLAKE
Client ID:  192.168.0.100
Action:     UPDATE
Empno:      7698
Column:     SAL
Old Value:  2850
New Value:  3000
Date:       Mar-24-04 13:34
------------------------------

PL/SQL procedure successfully completed.
```

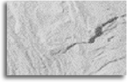
The last record is the original record that was generated in the previous section. The trigger fires five times—once for each row because the trigger fires for each row; you could easily audit at the statement level by making the trigger fire once per statement.

Finally, you can see that the preceding audit records were initiated by the trigger by viewing the call stack:

```
scott@KNOX10g> SELECT DISTINCT call_stack
  2              FROM aud_emp;

CALL_STACK
--------------------------------------------------
----- PL/SQL Call Stack -----
  object      line  object
  handle    number  name
692097F4         1  anonymous block
694CAF18        12  procedure SCOTT.AUDIT_EMP
691DECC0         3  SCOTT.UPDATE_EMP_SAL_TRIG

----- PL/SQL Call Stack -----
  object      line  object
  handle    number  name
692097F4         1  anonymous block
694CAF18        12  procedure SCOTT.AUDIT_EMP
6945FAEC        15  procedure SCOTT.UPDATE_SAL
693CEA5C         1  anonymous block
```

Benefits

One major benefit to trigger auditing is that the auditing can be transparent to the application. If you have purchased an application in which the code can't be modified, then trigger auditing may provide a robust mechanism for adding or augmenting what is already provided. The triggers also can be enabled only when specific columns are being manipulated, as seen in the previous example. The trigger can operate for each row or for each statement. This allows selectivity in auditing and reduces the number of unnecessary audit records. The trigger auditing will also be invoked for all applications regardless of language; that is, no matter how the user interacts with the data, the trigger will audit. This consistency is important.



NOTE
As with the application auditing, trigger auditing is programmed. From the benefits angle, this gives you many of the extensibility virtues that were discussed earlier.

Drawbacks

Triggers, while effective, aren't guaranteed. They don't fire for certain actions, such as TRUNCATE statements.

Triggers don't allow applications to pass additional parameters. They are constrained to the table columns. Outside of the new and old values of the data, the only other information the trigger can use are application contexts and environmental data, such as the user's name and connecting IP address.

Also, just like application auditing, triggers have to be created and defined for every object. Calling procedures from within triggers can help if the procedures can be shared across several triggers.

Autonomous Transactions and Auditing

As you may have noticed in the AUDIT_EMP procedure, the transaction wasn't committed. Note what happens then if the user doesn't commit the transaction. First, the audit trail is truncated:

```
scott@KNOX10g> TRUNCATE TABLE aud_emp;

Table truncated.
```

Now, the BLAKE user will check the SAL values, issue an update, and write down the results. Once he has the information he wants, he issues a rollback:

```
blake@KNOX10g> SELECT SUM (sal)
  2    FROM scott.emp_copy
  3   WHERE deptno = 20;

  SUM(SAL)
----------
   11962.5

blake@KNOX10g> UPDATE scott.emp_copy
  2     SET sal = sal * 1.1
  3   WHERE deptno = 20;

5 rows updated.

blake@KNOX10g> SELECT SUM (sal)
  2    FROM scott.emp_copy
  3   WHERE deptno = 20;

  SUM(SAL)
----------
  13158.75
```

```
blake@KNOX10g> ROLLBACK ;

Rollback complete.
```

When the audit trail is queried, there is no audit data! Its as if the update never happened or the audit data has been erased:

```
scott@KNOX10g> EXEC show_aud_emp;

PL/SQL procedure successfully completed.
```

That's because the updates never did happen. At least, the updates never happened from a database transactional perspective. The audit was part of the same transaction, so the rollback removed the audit entries. You know the updates did happen and not only that, but the user was able to see the results of the updates. The rollback performed as it was supposed to.

You may try to solve this issue by immediately placing a commit statement in the procedure, but you can't place a commit in the trigger or you'll receive a runtime "ORA-04092: can't COMMIT in a trigger" error.

Ensuring the audit data isn't erased on rollback is important for auditing in cases where you wish to capture actions being performed even if the actions are later "unperformed." In fact, the Oracle database auditing works precisely on this principle. Rollbacks don't erase the entries from the audit trails. To do this within the database, you can simply utilize autonomous transactions. These transactions are independent of the other transaction that the user session has created.

You can modify both your database trigger and your stored procedure to run as autonomous transactions. Once done, the updates will be audited even if the user issues a rollback. For the procedure, add a PRAGMA in the variables section and add a commit. This commit affects only our autonomous transaction:

```
scott@KNOX10g> CREATE OR REPLACE PROCEDURE audit_emp (
  2    p_username     IN  VARCHAR2,
  3    p_action       IN  VARCHAR2,
  4    p_empno        IN  NUMBER,
  5    p_column_name  IN  VARCHAR2,
  6    p_old_value    IN  VARCHAR2,
  7    p_new_value    IN  VARCHAR2)
  8  AS
  9    PRAGMA AUTONOMOUS_TRANSACTION;
 10  BEGIN
 11    INSERT INTO aud_emp
 12                (username,
 13                 action,
 14                 empno,
 15                 column_name,
 16                 call_stack,
 17                 client_id,
 18                 old_value,
 19                 new_value,
 20                 action_date)
 21         VALUES (p_username,
```

```
 22                  p_action,
 23                  p_empno,
 24                  p_column_name,
 25                  DBMS_UTILITY.format_call_stack,
 26                  SYS_CONTEXT ('userenv',
 27                               'client_identifier'),
 28                  p_old_value,
 29                  p_new_value,
 30                  SYSDATE);
 31     COMMIT;
 32   END;
 33   /

Procedure created.
```

Because both the trigger and the UPDATE_SAL procedure call this procedure, all updates will be audited, regardless of whether there is a commit or rollback issued. To test this, Blake issues the same update:

```
blake@KNOX10g> UPDATE scott.emp_copy
  2      SET sal = sal * 1.1
  3    WHERE deptno = 20;

5 rows updated.

blake@KNOX10g> ROLLBACK ;

Rollback complete.
```

User SCOTT can count the audits, and he'll see all five records even though the rollback removed the actual updates. The autonomous transaction-enabled AUDIT_EMP procedure preserved the audit records:

```
scott@KNOX10g> SELECT COUNT (*)
  2     FROM aud_emp;

  COUNT(*)
----------
         5
```

Data Versioning

New to the Oracle Database 10*g* are expanded Flashback capabilities. See Chapter 10 of the *Oracle High Availability Architectures and Best Practices* product documentation for details. The Flashback Versioning feature allows the user to view data as it has evolved over time. Depending on your auditing requirements, this may be precisely what you need.

Flashback Version Query

To illustrate this capability, refresh the data table and issue a 10 percent raise for everyone in the table:

```
scott@KNOX10g> -- Refresh data
scott@KNOX10g> TRUNCATE TABLE emp_copy;

Table truncated.

scott@KNOX10g> INSERT INTO emp_copy
  2    SELECT * FROM emp;

14 rows created.

scott@KNOX10g> COMMIT ;

Commit complete.

scott@KNOX10g> -- Give everyone a 10% raise
scott@KNOX10g> UPDATE emp_copy
  2     SET sal = sal * 1.1;

14 rows updated.

scott@KNOX10g> COMMIT ;
```

Behind the scenes, it appears the database has logged the updates. To access the data, you can use the following:

```
scott@KNOX10g> -- Show database record of values
scott@KNOX10g> COL ename format a6
scott@KNOX10g> COL sal format a8
scott@KNOX10g> COL "Start" format a12
scott@KNOX10g> COL "End"   format a12
scott@KNOX10g> COL "XID" format a17
scott@KNOX10g> COL operation format a9
scott@KNOX10g> SELECT   ename,
  2           TO_CHAR (sal) sal,
  3           DECODE (versions_operation,
  4                   'I', 'Insert',
  5                   'U', 'Update',
  6                   'D', 'Delete') operation,
  7           versions_xid "XID",
  8           TO_CHAR (versions_starttime,
  9                    'MM/DD HH24:MI') "Start",
 10           TO_CHAR (versions_endtime,
 11                    'MM/DD HH24:MI') "End"
 12      FROM emp_copy
 13     VERSIONS BETWEEN TIMESTAMP MINVALUE AND MAXVALUE
 14     WHERE deptno = 20
```

```
 15  ORDER BY 1, 2;

ENAME  SAL      OPERATION XID              Start        End
------ -------- --------- ---------------- ------------ ------------
ADAMS  1100                                              04/20 16:16
ADAMS  1210     Update    04000F00390B0000  04/20 16:16
FORD   3000                                              04/20 16:16
FORD   3300     Update    04000F00390B0000  04/20 16:16
JONES  2975                                              04/20 16:16
JONES  3272.5   Update    04000F00390B0000  04/20 16:16
SCOTT  3000                                              04/20 16:16
SCOTT  3300     Update    04000F00390B0000  04/20 16:16
SMITH  800                                               04/20 16:16
SMITH  880      Update    04000F00390B0000  04/20 16:16

10 rows selected.
```

The pseudocolumns and syntax that asks the database for the versioned data are in bold. This shows the old data, the new data, the type or operation, when the operation occurred, and that the update was part of the same transaction. Alternatively, the start and end values can be based on SCN.

Flashback Transaction Query

You can now use another feature called Flashback Transaction Query. The transaction ID returned in the preceding query can be used to get additional information stored in the FLASHBACK_TRANSACTION_QUERY view:

```
system@KNOX10g> COL table_owner format a11
system@KNOX10g> COL table_name format a10
system@KNOX10g> COL operation format a10
system@KNOX10g> COL logon_user format a10
system@KNOX10g> SELECT DISTINCT table_owner,
  2                   table_name,
  3                   operation,
  4                   logon_user
  5              FROM flashback_transaction_query
  6             WHERE xid =
  7                        HEXTORAW ('04000C00340B0000')
  8               AND table_name IS NOT NULL;

TABLE_OWNER TABLE_NAME OPERATION  LOGON_USER
----------- ---------- ---------- ----------
SCOTT       EMP_COPY   UPDATE     SCOTT
```

The Flashback capabilities, which allow you to restore data very efficiently, are discussed in the *Oracle High Availability* document. The view stores the SQL statements needed to recover the data to its original state, as you can see:

```
system@KNOX10g> SELECT undo_sql
  2    FROM flashback_transaction_query
```

```
  3   WHERE xid = HEXTORAW ('0400150012010000')
  4     AND table_name IS NOT NULL
  5     AND ROWNUM <= 1;

UNDO_SQL
------------------------------------------------------------
update "SCOTT"."EMP_COPY" set "SAL" = '1300' where ROWID =
AAAMa1AAEAAACQUAAN';
```

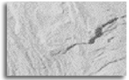
If you wanted to recover the original salary values, you could run the following:

```
system@KNOX10g> -- current data
system@KNOX10g> SELECT ename, sal FROM scott.emp_copy
  2   WHERE deptno = 20;

ENAME             SAL
---------- ----------
SMITH             880
JONES          3272.5
SCOTT            3300
ADAMS            1210
FORD             3300

system@KNOX10g> -- recover data
system@KNOX10g> DECLARE
  2    l_undo_sql  VARCHAR2 (32767);
  3  BEGIN
  4    FOR rec IN
  5      (SELECT undo_sql
  6         FROM flashback_transaction_query
  7        WHERE xid = HEXTORAW ('0400150012010000')
  8          AND table_name IS NOT NULL)
  9    LOOP
 10      l_undo_sql := REPLACE (rec.undo_sql, ';', '');
 11      EXECUTE IMMEDIATE l_undo_sql;
 12    END LOOP;
 13
 14    COMMIT;
 15  END;
 16  /

PL/SQL procedure successfully completed.

system@KNOX10g> SELECT ename, sal FROM scott.emp_copy
  2   WHERE deptno = 20;

ENAME             SAL
---------- ----------
SMITH             800
JONES            2975
```

```
SCOTT             3000
ADAMS             1100
FORD              3000
```

The ability to recover the data isn't limited to the transactions. The database supports additional flashback modes: Flashback Table, which allows you to recover an entire table to a point in time; Flashback Drop, which allows you to restore tables that were dropped; and Flashback Database, which recovers the entire database to a point in time. All of these may prove beneficial if a hacker attacks your database and starts corrupting or destroying data.

All of the flashback technology, except Flashback Database, is built on Oracle's multiversion read consistency implementation. This means there is no auditing performance penalty. Oracle is always "logging" the changes. The flashback operations are a new way of exploiting an implementation design that has been available for several years.



NOTE
The flashback features use the undo management system for data access. The data is therefore not permanent. Its lifetime is dependent on the UNDO_RETENTION value and the UNDO_MANAGEMENT initialization variables. To persist the data, you will have to copy it from the Flashback areas.

Standard Database Auditing

From a native database perspective, auditing comes in four flavors: mandatory, standard, Fine-Grained, and administrator (SYS) auditing.

Mandatory Auditing

The database always records three important things: database startup, database shutdown, and users authenticated with the SYSDBA or SYSOPER roles. For the database startup, the audit record also indicates whether standard auditing has been enabled. This allows one to determine if an administrator has disabled auditing (setting the AUDIT_TRAIL value to none or FALSE) and is now restarting the database.

These audit records have to be stored on the operating system because the database isn't available (it's being started or stopped). The actual location varies depending on OS platform—for example, on Windows, the records are written to the Event Logs; on Linux, the audits are generally found in the $ORACLE_HOME/rdbms/audit directory.

Auditing SYS

As of Oracle9*i* Database Release 2, auditing actions performed by users authenticated as SYSDBA or SYSOPER are also supported. The audit records are again written to OS files. This is important for two reasons. First, these users have the most significant privileges in the database, such as the ability to see and modify all data, change passwords, log in to any schema, and drop schemas. As such, it's generally advisable to monitor their actions.

Second, they control the database auditing. If they want to disable it, they have the privileges to do so. They also have the privileges to delete the audit records. Therefore, auditing to the database is useless since the user would be able to modify or delete the audit records. When auditing on SYS, it's important to remember that you shouldn't allow the database user access to

the operating system directories where the audits will be written, or else you will suffer from the same challenge as auditing to the database. To enable audits for the SYS user, you have to set the AUDIT_SYS_OPERATIONS initialization parameter to TRUE:

```
system@KNOX10g> ALTER SYSTEM SET audit_sys_operations=TRUE
  2    SCOPE=SPFILE;

System altered.
```

This change is written to the initialization file (init.ora), and the database has to be rebooted for the change to take effect. Trying to change this parameter at runtime results in the error: "ORA-02095: specified initialization parameter cannot be modified". This is a security feature. If the database could be modified at runtime without booting, the SYS user could turn off auditing, do something bad, and then re-enable auditing. By forcing a reboot, you know that you'll have captured both the disabling of the auditing as well as the reboot.

After restarting the database, all successful actions performed by the SYS user will be audited. To illustrate this point, set the parameter to TRUE. Then authenticate as SYSDBA and change the SYS user's password. In Figure 8-1, the Microsoft Event Viewer shows how the event has been audited.

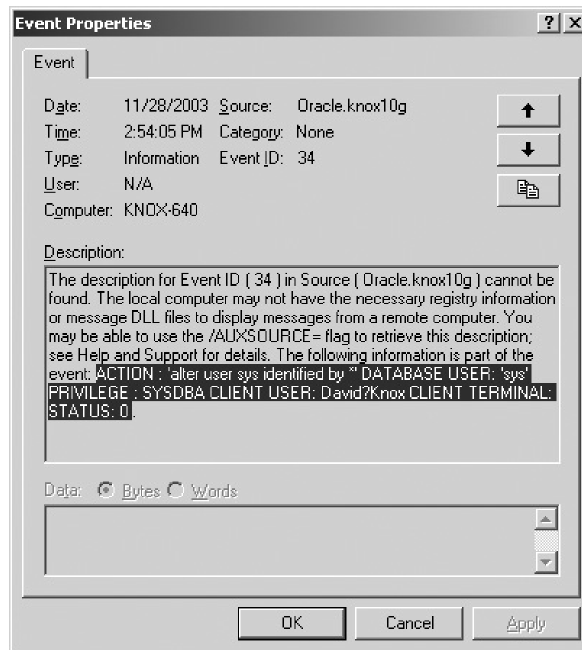


FIGURE 8-1. *Auditing SYS actions ensures accountability for the most privileged database users.*

Enabling Standard Auditing

When most people think of database auditing, they think of standard auditing. Before you can successfully utilize the database standard auditing, you have to enable it by setting the AUDIT_TRAIL initialization parmeter. Note this is a distinctly separate parameter from AUDIT_SYS_OPERATIONS, which just audits the actions of SYS. The AUDIT_TRAIL parameter allows standard database auditing to occur. By default, this parameter is also set to FALSE.

To enable auditing, you can set the AUDIT_TRAIL parameter to several values. Setting the value to "OS" will enable the database audit records to be written to the operating system (OS). As with auditing on SYS, this is a good idea if you're concerned with someone modifying or deleting the audit records that might otherwise be contained within the database. However, auditing to the OS can make it difficult to run reports because the data is in a text file.

To enable auditing for records stored in the database (records are stored in the SYS.AUD$ table), you have to set the AUDIT_TRAIL parameter to either "DB" or to "TRUE". New in Oracle Database 10*g* is the ability to set auditing to "DB_EXTENDED". DB_EXTENDED also enables database auditing but captures additional information in the audit record. The SQL text and any potential bind variables that actually caused the audit to occur, as well as some other useful information, will be captured. You see this in the following example:

```
system@KNOX10g> ALTER SYSTEM SET audit_trail=db_extended SCOPE=SPFILE;

System altered.
```

For the same security precautions just discussed, the database has to be rebooted before the changes will take effect. As a reboot of the database is required, it's best to set the AUDIT_TRAIL parameter and reboot so once you decide what you really want to audit, you'll be able to do it without needing to restart the database. A security best practice is to set the AUDIT_TRAIL value to DB_EXTENDED on database creation. By default the value is not set.

Auditing By User, Privilege, and Object

The database allows for a very robust auditing environment:

- You can audit on objects such as tables and views. For example, every time someone accesses the APP.USER_DATA table an audit will be recorded.
- You can audit procedure executions.
- You can audit when someone exercises a system privilege, such as disabling a trigger or using the SELECT ANY TABLE privilege.
- You can restrict your audits to specific users.
- You can audit for successful actions, unsuccessful actions, or both.
- With all of the preceding, you can audit every time someone performs the action, or audit only once per session regardless of the number of times they perform the action within the session.

The point is that this extensive capability allows you to focus your auditing to precisely the things that are of interest. This fidelity is what makes the auditing a real asset.

Targeted Auditing

As mentioned previously, auditing can only be done successfully once you have a clear idea of why you are auditing and what you are auditing. If you audit too much, you could suffer performance hits, and more importantly, generate excessive and potentially useless records. Culling through thousands of audit records is generally ineffective, especially when most of the audits were done for users that were performing the tasks they were assigned. The greater the fidelity in the auditing capability, the better your chances are of focusing the auditing to just the right level on just the right things.

Auditing can help you identify gaps in your security policies. From the audit records, you may notice that authorized users don't have the necessary privileges, or on the other extreme that they are over privileged. You may even identify contradictions within your security policy. For example, it's typical to have one requirement to support database backups that allows a user to gain access to the entire database. Another requirement based on "need to know" may say that users are only allowed access to certain records based on their affiliation. These two requirements may conflict.

This highlights an important point to auditing. In cases where administrators require super privileges, *auditing may be the only thing you can do to ensure privileges are not abused and misused*. This is exactly why the database supports audits for SYS.

Auditing Best Practices

In addition to targeted auditing, there are a couple of occasions worth constant consideration.

Audit Connections

Auditing when users log on and log off of the database is a good thing to do. It's a matter of common sense. You should also know who has been in your database(s). If something bad happens to the database and you know about when the bad thing happened, it's invaluable to be able to find out who was working in the database when the incident happened. Acknowledging the fact that most database outages are human induced and not a software or hardware failure, auditing connections will help narrow the field of possible suspects.

However, there are two things that have to be done to ensure this type of auditing is effective. Most importantly, you have to be able to distinguish between users. Auditing on applications that conceal the user's identity may be pointless—after all, you really can't distinguish the "who," only that it was a person running the application. For applications with connection pools, this type of auditing may be ineffective.

Second, and this applies to all auditing, you have to do something with the records. With connections, this point is particularly acute. Because the users will be connecting all the time, unless you have a tremendous amount of disk space, you'll have to archive and delete the old records.

To enable auditing for logons and logoffs, simply audit the user's connection. Again, you can do this at a user level or for all users. First, check to ensure that auditing is enabled, and then audit all connections for all users:

```
system@KNOX10g> show parameter audit_trail

NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------
audit_trail                          string      DB_EXTENDED
```

```
system@KNOX10g> AUDIT SESSION;

Audit succeeded.
```

The check for the value of the AUDIT_TRAIL parameter is a good habit. This is because the "audit session;" statement will succeed even if the database auditing is disabled (AUDIT_TRAIL = FALSE). You might think you are auditing, but in reality you aren't.

To complete our example, log on and log off as the user SCOTT (not shown). Upon logon, the database writes an entry to the audit trail:

```
system@KNOX10g> BEGIN
  2    FOR rec IN
  3      (SELECT username,
  4              action_name,
  5              TO_CHAR (TIMESTAMP, 'Mon-DD HH24:MI')
  6                                              LOGON,
  7              TO_CHAR (logoff_time,
  8                       'Mon-DD HH24:MI') LOGOFF,
  9              priv_used,
 10              comment_text
 11         FROM dba_audit_trail)
 12    LOOP
 13      DBMS_OUTPUT.put_line (   'User:      '
 14                            || rec.username);
 15      DBMS_OUTPUT.put_line (   'Action:    '
 16                            || rec.action_name);
 17      DBMS_OUTPUT.put_line (   'Logon:     '
 18                            || rec.LOGON);
 19      DBMS_OUTPUT.put_line (   'Logoff:    '
 20                            || rec.LOGOFF);
 21      DBMS_OUTPUT.put_line (   'Priv Used: '
 22                            || rec.priv_used);
 23      DBMS_OUTPUT.put_line (   'Comments:  '
 24                            || rec.comment_text);
 25      DBMS_OUTPUT.put_line
 26        ('--------------- End of Record ---------------');
 27    END LOOP;
 28  END;
 29  /
User:      SCOTT
Action:    LOGON
Logon:     Mar-24 15:50
Logoff:
Priv Used: CREATE SESSION
Comments:  Authenticated by: DATABASE; Client address:
(ADDRESS=(PROTOCOL=tcp)(HOST=192.168.0.100)(PORT=3445))
--------------- End of Record ---------------

PL/SQL procedure successfully completed.
```

Note the useful information the database automatically places in the COMMENT_TEXT field. The logoff time is NULL because the user in this example is still logged in. Once the user has disconnected, the database updates the audit entry. The ACTION is changed to LOGOFF, and the actual logoff time is indicated as well:

```
system@KNOX10g> /
User:       SCOTT
Action:     LOGOFF
Logon:      Mar-24 15:50
Logoff:     Mar-24 15:53
Priv Used: CREATE SESSION
Comments:  Authenticated by: DATABASE; Client address:
(ADDRESS=(PROTOCOL=tcp)(HOST=192.168.0.100)(PORT=3445))
--------------- End of Record ---------------

PL/SQL procedure successfully completed.
```

This simple ability to track who was connected and when is valuable and may be the only records you have prior to something happening.

Audit Whenever Unsuccessful

Auditing unsuccessful actions represents the database's ability to detect the burglar rattling the windows and checking the doors. You should know when someone is banging on a locked door. This is especially important with truly sensitive data or with data that can be used to derive sensitive information, such as privacy-related data, encryption keys, passwords, and user preferences.

Let's show this in action. You'll enable auditing on a table, but your auditing policy will only audit when users try to access the object but don't have the proper privileges. This may be an indication that the user is trying to gain unauthorized access to our sensitive data.

This will audit by access and not by session. This means for every SQL statement that touches the table (or rather, tries to touch the table), an audit record will be generated. If you audited by session, there would be only one record for each session regardless of the number of actual SQL statements that accessed the table.

```
sec_mgr@KNOX10g> CREATE TABLE t AS SELECT * FROM DUAL;

Table created.

sec_mgr@KNOX10g> -- audit selects on T for failures
sec_mgr@KNOX10g> AUDIT SELECT ON t
  2    BY ACCESS WHENEVER NOT SUCCESSFUL;

Audit succeeded.
```

When an unauthorized user tries to access this table the action will be audited.

```
scott@KNOX10g> SELECT *
  2    FROM sec_mgr.t;
  FROM sec_mgr.t
```

```
               *
ERROR at line 2:
ORA-00942: table or view does not exist
```

Checking the audit trail, you see this failed attempt to access the table:

```
sec_mgr@KNOX10g> BEGIN
  2    FOR rec IN (SELECT audit_type,
  3                       db_user,
  4                       object_schema,
  5                       object_name,
  6                       extended_timestamp,
  7                       sql_text
  8                  FROM dba_common_audit_trail)
  9    LOOP
 10      DBMS_OUTPUT.put_line (   'Audit Type: '
 11                            || rec.audit_type);
 12      DBMS_OUTPUT.put_line (   'Who:        '
 13                            || rec.db_user);
 14      DBMS_OUTPUT.put_line (   'What:       '
 15                            || rec.object_schema
 16                            || '.'
 17                            || rec.object_name);
 18      DBMS_OUTPUT.put_line (   'When:       '
 19                            || rec.extended_timestamp);
 20      DBMS_OUTPUT.put_line (   'How:        '
 21                            || rec.sql_text);
 22      DBMS_OUTPUT.put_line
 23        ('--------------- End of Record ---------------');
 24    END LOOP;
 25  END;
 26  /
Audit Type: Standard Audit
Who:        SCOTT
What:       SEC_MGR.T
When:       24-MAR-04 04.11.10.350000 PM -05:00
How:        SELECT * FROM sec_mgr.t
--------------- End of Record ---------------
```

The audit type of “Standard Audit” is used because this new view integrates both the standard audit records and fine-grained audit records.

Tracking Database Use

You can also use auditing to check or verify how the database is being used. For example, you may wonder whether a table, schema, or procedures are still being used. Enabling auditing will indicate if the object(s) is still in use.

Also recall that removing unused schemas is a security best practice. Before you drop the schema, it's best to audit access to objects or connections to the schema to determine if the schema really is unused.

Determining the Audit Status

Evaluating the current audit status is important especially when performance becomes questionable. It can also be useful for proving compliance with a security policy. When you want to inspect the audit status on your objects, you can query the DBA_OBJ_AUDIT_OPTS or the USER_OBJ_AUDIT_OPTS view. The views show the audit status of every object even if auditing isn't enabled. You should therefore form your query to target the specific object or a specific statement of interest:

```
sec_mgr@KNOX10g> COL "select option" format a13
sec_mgr@KNOX10g> SELECT sel "select option"
  2    FROM user_obj_audit_opts
  3   WHERE object_name = 'T';

select option
-------------
-/A
```

The format is a bit cryptic, but it's easy to use once you know the code. The USER_OBJ_AUDIT_OPTS view lists all the options that can be performed on an object. In this query, you're just looking at the select option. There are two values represented for each option. The first field, represented by a hyphen in the above output, indicates if auditing should occur when the user is successful in performing the action. The second field, an "A" in the output above, indicates whether auditing will occur for unsuccessful actions. A hyphen or a blank means no auditing will occur. An "A" means auditing will occur for every access, and an "S" means auditing will occur once for each session.

If you enable auditing by session on the table, you can verify the previous rules:

```
 sec_mgr@KNOX10g> AUDIT SELECT ON t BY SESSION WHENEVER SUCCESSFUL;

Audit succeeded.

sec_mgr@KNOX10g> SELECT sel "select option"
  2    FROM user_obj_audit_opts
  3   WHERE object_name = 'T';

select option
-------------
S/A
```

The DBA_STMT_AUDIT_OPTS view shows system wide auditing. The auditing that was enabled to track user logons can be viewed by this query:

```
sec_mgr@KNOX10g> COL user_name format a10
sec_mgr@KNOX10g> COL proxy_name format a10
sec_mgr@KNOX10g> COL audit_option format a20
sec_mgr@KNOX10g> SELECT * FROM dba_stmt_audit_opts;
```

```
USER_NAME  PROXY_NAME AUDIT_OPTION         SUCCESS    FAILURE
---------- ---------- -------------------- ---------- ----------
                      CREATE SESSION       BY ACCESS  BY ACCESS
```

The last essential view is DBA_PRIV_AUDIT_OPTS, which allows you to check the auditing status of system privileges. For example, if you enable auditing for users exercising the SELECT ANY TABLE privilege, you can check the status as follows:

```
sec_mgr@KNOX10g> -- Audit the SELECT ANY TABLE privilege
sec_mgr@KNOX10g> AUDIT SELECT ANY TABLE;

Audit succeeded.

sec_mgr@KNOX10g> COL privilege format a20
sec_mgr@KNOX10g> -- Show privilege auditing
sec_mgr@KNOX10g> SELECT * FROM dba_priv_audit_opts;

USER_NAME  PROXY_NAME PRIVILEGE            SUCCESS    FAILURE
---------- ---------- -------------------- ---------- ----------
                      SELECT ANY TABLE     BY SESSION BY SESSION
```

Extending the Audit Data with Client Identifiers

A similar statement made earlier regarding preventive security measures also holds true with auditing: The more information the database has, the better the security. With auditing, you can augment the audit trails using Client Identifiers. If the user's identity is already known to the database, the Client Identifier should be seeded with other useful information such as the user's IP address, the application they are running, how they authenticated, and anything and everything that can be used to describe the user's operating context. Please see the section "Securing the Client Identifier" in Chapter 6.

As illustrated previously, logon triggers can be used to effectively set meaningful information about the user in the Client Identifier.

The code presented earlier can be modified to set the user's IP address and the program they are using and storing the result in the Client Identifier. The module name can be spoofed by the user but it does often indicate correctly the name of the application being used such as SQL*Plus, MS Excel, or T.O.A.D. Note that changing the program's executable name changes the program and not the module:

```
sec_mgr@KNOX10g> CREATE OR REPLACE TRIGGER set_default_client_info
  2    AFTER LOGON ON DATABASE
  3  DECLARE
  4    l_module  v$session.module%TYPE;
  5  BEGIN
  6    SELECT UPPER (module)
  7      INTO l_module
  8      FROM v$process a, v$session b
  9     WHERE a.addr = b.paddr
 10       AND b.audsid = USERENV ('sessionid');
 11    DBMS_SESSION.set_identifier
 12                        (   SYS_CONTEXT
 13                                          ('userenv',
 14                                           'ip_address')
```

```
 15                        || ':'
 16                        || l_module);
 17  END;
 18  /

Trigger created.
```

This code could be extended to capture the authentication mode, the network protocol, or any other valuable piece of information you want to extract from the user session environment.

Now enable auditing on the SCOTT.EMP object. By not specifying when you want to audit, you'll be auditing for both successful and unsuccessful actions as well as auditing by access so you can capture a record for every SQL statement:

```
sec_mgr@KNOX10g> AUDIT SELECT ON scott.emp BY ACCESS;

Audit succeeded.
```

For testing, delete the existing records from the audit table:

```
sys@KNOX10g> DELETE FROM aud$;
```

Next log on as SYSTEM and issue a query on the table:

```
system@KNOX10g> SELECT ename, sal
  2    FROM scott.emp
  3   WHERE sal < (SELECT sal
  4                  FROM scott.emp
  5                 WHERE ename = 'SCOTT')
  6     AND deptno = (SELECT deptno
  7                     FROM scott.emp
  8                    WHERE ename = 'SCOTT');

ENAME             SAL
---------- ----------
SMITH             800
JONES            2975
ADAMS            1100
```

Now, as SCOTT, issue two more queries:

```
scott@KNOX10g> SELECT SUM (sal) FROM scott.emp;

  SUM(SAL)
----------
     29025

scott@KNOX10g> SELECT ename FROM scott.emp
  2   WHERE 1 = 2;

no rows selected
```

Checking the audit records, you see there are five records in the audit trail: the two from SCOTT and three that were generated from the single statement issued by SYSTEM. You'll create a procedure to group the records together so you actually see only three results. This ability to correlate records is also a new capability enabled by the DB_EXTENDED value of the AUDIT_TRAIL parameter.

```
sec_mgr@KNOX10g> SELECT COUNT (*)
  2    FROM dba_common_audit_trail;

  COUNT(*)
----------
         5

sec_mgr@KNOX10g> CREATE OR REPLACE PROCEDURE show_aud
  2  AS
  3  BEGIN
  4    FOR rec IN (SELECT   db_user,
  5                         client_id,
  6                         object_schema,
  7                         object_name,
  8                         extended_timestamp,
  9                         sql_text,
 10                         statementid
 11                    FROM dba_common_audit_trail
 12                GROUP BY db_user,
 13                         statementid,
 14                         sql_text,
 15                         object_schema,
 16                         object_name,
 17                         client_id,
 18                         extended_timestamp
 19                ORDER BY extended_timestamp ASC)
 20    LOOP
 21      DBMS_OUTPUT.put_line ('Who:  ' || rec.db_user);
 22      DBMS_OUTPUT.put_line (    'What: '
 23                             || rec.object_schema
 24                             || '.'
 25                             || rec.object_name);
 26      DBMS_OUTPUT.put_line (    'Where: '
 27                             || rec.client_id);
 28      DBMS_OUTPUT.put_line
 29                   (    'When:  '
 30                    || TO_CHAR
 31                             (rec.extended_timestamp,
 32                              'Mon-DD HH24:MI'));
 33      DBMS_OUTPUT.put_line ('How:    '
 34                             || rec.sql_text);
 35      DBMS_OUTPUT.put_line
 36        ('-------------- End of Record --------------');
 37    END LOOP;
```

```
 38  END;
 39  /

Procedure created.

sec_mgr@KNOX10g> EXEC show_aud
Who:  SCOTT
What: SCOTT.EMP
Where: 192.168.0.100:SQLPLUS.EXE
When:  Mar-24 16:28
How:   SELECT ename FROM scott.emp
 WHERE 1 = 2
-------------- End of Record --------------
Who:  SCOTT
What: SCOTT.EMP
Where: 192.168.0.100
When:  Mar-24 16:28
How:   SELECT SUM (sal) FROM scott.emp
-------------- End of Record --------------
Who:  SYSTEM
What: SCOTT.EMP
Where: 192.168.0.100:SQLPLUS.EXE
When:  Mar-24 16:28
How:   SELECT ename, sal
  FROM scott.emp
 WHERE sal < (SELECT sal
                FROM scott.emp
WHERE ename = 'SCOTT')
   AND deptno = (SELECT deptno
                   FROM scott.emp
WHERE ename = 'SCOTT')
-------------- End of Record --------------

PL/SQL procedure successfully completed.
```

The correlation of records was done by the grouping by statementIDs. For other DML types, you also can correlate by transactionID. This will link together all the inserts, updates, and deletes that were part of the same transaction, which allows us to support transaction level auditing in addition to the statement (task) level auditing.

TIP
To disable auditing (useful if you are following along), you have to issue the NOAUDIT command: NOAUDIT SELECT ON scott.emp;.

Peformance Test

No matter what you read, auditing has to be (at least theoretically) slower than not auditing because you are doing more work. A quick test should always be performed in coordination with

your auditing policy. In the preceding example, you can run a loop of queries on the table with and without auditing enabled to determine how much the auditing operations will add to your execution time.

Here are the results—your mileage may vary. First, without auditing, query a table 100,000 times in just over six seconds:

```
scott@KNOX10g> BEGIN
  2    FOR rec IN 1 .. 100000
  3    LOOP
  4      FOR irec IN (SELECT ename
  5                     FROM scott.emp)
  6      LOOP
  7        NULL;
  8      END LOOP;
  9    END LOOP;
 10  END;
 11  /

PL/SQL procedure successfully completed.

Elapsed: 00:00:06.33
```

Auditing can be disabled at the system level via the AUDIT_TRAIL parameter and at the object level via audit policies. The time to execute the preceding was approximately the same, regardless of how the auditing was disabled.

Next, the same 100,000 queries are executed. When auditing by access, this will insert 100,000 records into the audit table. This naturally is where the slow-down occurs:

```
scott@KNOX10g> /

PL/SQL procedure successfully completed.

Elapsed: 00:23:06.22
```

You might expect something like this. There are a lot of records being inserted into a table. The point is that there is no magic with auditing. That insert process is going to take time.

There are a couple of important conclusions that should be obvious from this. First, this test is contrived and doesn't represent any real-world application. It was done intentionally to get the drastic results seen. *You have to test based on your true expected load and access methods.* Querying in the loop shows how fast the database can process serial requests from the same user. It doesn't indicate how it will work concurrently with mulitple users.

Second, this shows how careful consideration has to be given to auditing. One hundred thousand audit records are a lot of records. This might not be the right level of auditing for our specific data usage. You might decide to change the audit from "access" to "sesssion," or you might want to audit only on failed access attempts. Both would limit the number of records and increase performance. The answer is dependent on why you are auditing and what you hope to get from the audit records. One thing the preceding example illustrates is that SCOTT is querying this table many times in a short period of time using the same SQL. Perhaps he is executing a program that is (accidentally) continuously looping.

Caveats

Standard auditing has a few drawbacks. The auditing fidelity, while very good to a point, may not be adequate for some requirements. For example, if you want to capture audits on specific columns or when specific conditions arise, then standard auditing would be ineffective.

Another issue is one that is arguably not part of the auditing domain but nonetheless still frequently brought up. The audits don't indicate what the user actually saw as a result of the query. None of the implementations of auditing in this chapter have that ability by themselves. However, Oracle is aware of this, and the audit records do capture the time of the audit and the SCN (system change number). Either of these values allows an administrator to execute the Flashback ability of the database. Once the Flashback has occurred, you can rerun the query the user ran logged in as the user and see what the user saw when they issued the query. Note if you issue an ALTER SESSION SET CURRENT_SCHEMA=<AuditedUser> you will only be resolving objects as the user; their privileges and context values aren't included, so the results from the Flashback query are very likely to be erroneous.

There are limitations to the longevity of the Flashback data. An alternative for those that are concerned with this issue is to use an Oracle solution called Selective Audit. *Selective Audit* is an Oracle Consulting supported solution that puts Oracle's Standard Auditing on steroids. Search on the term "Selective Audit" on www.oracle.com for more information.

Fine-Grained Auditing

With Oracle9*i* Database Release 9.0.1, Oracle introduced another level of auditing in fine-grained auditing (FGA). Originally, FGA was only possible for SELECT statements. Other DML wasn't supported (the alternative was to use database triggers on insert, update, and delete statements). FGA has been significantly enhanced in Oracle Database 10*g* by allowing audits to extend to all DML statements.

FGA's extension to the standard auditing was actually borne out of the Oracle Consulting work done in the government division. FGA is designed to solve some of the issues illuminated in the previous "Caveats" section. Audit policies sometime require another level of fidelity not currently supported in the standard auditing. The auditing can be more selectively enabled to occur only when certain conditions arise and specific columns are queried. One of the practical aspects of FGA is that it makes the auditing functions behave more like intrusion detection systems. That is, you can set up your preventive security controls as normal. Then you can use the FGA as a safety net to catch things that fall through. You'll see this come to life in the upcoming examples.

In security practice, there is no such thing as perfect security. It's not *whether* you will be compromised, but rather *when* you will be. You can only hope to have the auditing enabled.

Audit Conditions

When FGA was originally introduced, it offered four major advantages over standard auditing: a Boolean condition check, the SQL capture ability, a column-sensitivity feature, and an event handler. The first advantage allowed the auditing to occur only when a specific condition was met. That is, at execution time, the audit policy would test a Boolean condition. If the condition was met, then and only then would the audit occur.

This has enormous advantages. It's very flexible because the Boolean conditions can be anything specified in SQL, including a comparison of the results of function calls. Another advantage is that the conditional auditing helps to eliminate unnecessary audits.

In standard auditing, you could audit anytime someone queried a table. However, you couldn't specify any exemptions to this based on specific conditions. Perhaps you are concerned with privacy issues, and you set up your access control to prevent users from seeing other users' records. In standard auditing, there is no inherent way to validate this policy is being accurately enforced. You could set up auditing for selects on your table. However, the problem is that the audit records would only indicate a user selected from the table and would not indicate whether they were able to access another user's records. You might be able to derive the result set of the query by looking at the SQL captured, but this would prove cumbersome and unreliable. Assuming users are supposed to access the table to see their records, auditing SELECT statements to meet this requirement is ineffective.

With FGA, you can complement the security policy by auditing only when a user is accessing someone else's records. The database allows you to specify a condition that when met will audit.

NOTE
You don't care why or how they circumvented our security policy; auditing is not prevention—it is detection.

Enabling FGA

Enabling FGA is completely different from enabling standard auditing. FGA doesn't rely on any initialization parameters. Instead, you control FGA via the DBMS_FGA package. Consider the following in which you want to audit SCOTT.EMP. Assume you want to audit only when a user is accessing another user's records. This is done by calling the ADD-POLICY procedure and specifying the condition as seen next. If you leave the condition NULL, then the audit will always occur. This is a new feature; in Oracle9*i* Database, to ensure auditing would always occur, you had to define a condition that was always true such as '1=1'.

```
sec_mgr@KNOX10g> BEGIN
  2    DBMS_FGA.add_policy
  3                 (object_schema      => 'SCOTT',
  4                  object_name        => 'EMP',
  5                  policy_name        => 'Example',
  6                  audit_condition    => 'ENAME != USER');
  7  END;
  8  /

PL/SQL procedure successfully completed.
```

To illustrate FGA at work, you can simply connect as SCOTT and issue queries. Before you do, it's a good practice to ensure the cost-based analyzer is being used (especially for Oracle9*i* Databases). Strange results from FGA can occur if you don't first issue an ANALYZE on the table. Prior to running this, the current audit records were deleted and the standard auditing on the EMP table was disabled:

```
scott@KNOX10g> ANALYZE TABLE emp COMPUTE STATISTICS;

Table analyzed.
```

```
scott@KNOX10g> SELECT sal, comm FROM emp
  2   WHERE ename = 'SCOTT';

       SAL       COMM
---------- ----------
      3000

scott@KNOX10g> SELECT sal, comm FROM emp
  2   WHERE deptno = 20;

       SAL       COMM
---------- ----------
       800
      2975
      3000
      1100
      3000
```

Going to the audit trail, you'll notice that only the second query was recorded. This is good because the first query was, by our security policy, an allowable query:

```
sec_mgr@KNOX10g> EXEC show_aud

PL/SQL procedure successfully completed.

sec_mgr@KNOX10g> EXEC show_aud
Who:  SCOTT
What: SCOTT.EMP
Where: 192.168.0.100:SQLPLUS.EXE
When:  Mar-24 18:34
How:   SELECT sal, comm FROM emp
 WHERE deptno = 20
-------------- End of Record --------------
```

The condition capability of FGA allows you a higher degree of fidelity in the auditing. The trick to making this most effective is to implement the Boolean checks within a separate PL/SQL function. Allowing the complexity to reside inside an external program allows for simplicity in adding and validating the audit policy.

For example, you might want your audit policy to audit only on weekends and off hours. You can create a program that returns true (represented by the number 1) if it is currently outside of normal operating hours (Monday through Friday, 8 A.M.–6 P.M.).

```
sec_mgr@KNOX10g> CREATE OR REPLACE FUNCTION is_off_hours
  2     RETURN BINARY_INTEGER
  3   AS
  4     l_return_val  BINARY_INTEGER;
  5     l_day_number  VARCHAR2 (1)
  6                           := TO_CHAR (SYSDATE, 'D');
  7     l_hour        VARCHAR2 (2)
```

```
  8                        := TO_CHAR (SYSDATE, 'HH24');
  9  BEGIN
 10    IF (   l_day_number IN ('1', '7')
 11        OR l_hour < 8
 12        OR l_hour > 18)
 13    THEN
 14      l_return_val := 1;
 15    ELSE
 16      l_return_val := 0;
 17    END IF;
 18
 19    RETURN l_return_val;
 20  END;
21  /

Function created.
```

You can simply update the current policy and add the function as the audit condition. To do this, first drop the policy, and then add it again:

```
sec_mgr@KNOX10g> BEGIN
  2    DBMS_FGA.drop_policy (object_schema    => 'SCOTT',
  3                          object_name      => 'EMP',
  4                          policy_name      => 'Example');
  5    DBMS_FGA.add_policy
  6      (object_schema      => 'SCOTT',
  7       object_name        => 'EMP',
  8       policy_name        => 'Example',
  9       audit_condition    => 'SEC_MGR.IS_OFF_HOURS = 1');
 10  END;
 11  /

PL/SQL procedure successfully completed.
```

The result is that your audit condition will then audit all selects on the EMP table after normal operating hours. You have time-sensitive auditing. Unfortunately, when you execute a query as SCOTT, it fails as follows:

```
scott@KNOX10g> select * from emp;
select * from emp
                *
ERROR at line 1:
ORA-28112: failed to execute policy function
```

SCOTT doesn't have execute privileges on the IS_OFF_HOURS function. To resolve this, grant execute on the function to SCOTT. Note that you specified the schema.function name when you registered the policy. This is also critical because the policy would fail to execute if the function couldn't be resolved by the invoking user. (This could provide a useful and additional tool to protecting your database tables.)

```
sec_mgr@KNOX10g> GRANT EXECUTE ON is_off_hours TO scott;

Grant succeeded.
```

The privilege to execute the audit event handler function is of significance mostly because this behavior isn't consistent with the policy functions used in Oracle's DBMS_RLS feature, which provides row-level security (discussed in detail in Chapter 10).

Column Sensitivity

A second advantage FGA introduced was the notion of column sensitivity. Assume now that users are allowed to access other users' records, just not the salary field within those records. In this case, your sensitive column is the SAL column. If the user's query doesn't touch this column, then you don't audit.

The actual auditing will occur when both the sensitive column(s) is queried and the Boolean condition is met. In the case that no condition is specified, the auditing will occur only when the sensitive column is queried or manipulated.

As an example, the previous audit policy can be modified to audit when users query other users' salaries. Therefore, you'll have to specify both a condition *and* a sensitivity column:

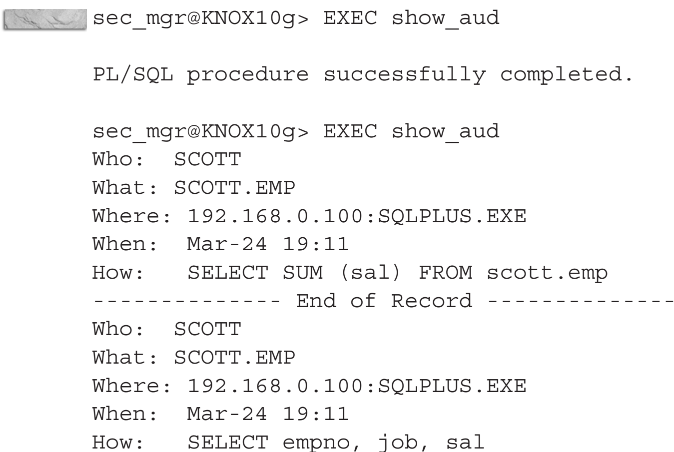
```
sec_mgr@KNOX10g> BEGIN
  2    DBMS_FGA.drop_policy (object_schema    => 'SCOTT',
  3                          object_name      => 'EMP',
  4                          policy_name      => 'Example');
  5    DBMS_FGA.add_policy
  6               (object_schema      => 'SCOTT',
  7                object_name        => 'EMP',
  8                policy_name        => 'Example',
  9                audit_condition    => 'ENAME != USER',
 10                audit_column       => 'SAL');
 11  END;
 12  /

PL/SQL procedure successfully completed.
```

Now, test the auditing by executing several queries. Again the current audit records were deleted prior to running the queries shown here. The comments indicate what should happen with respect to auditing:

```
scott@KNOX10g> -- Aggregate value on sensitive column causes audit
scott@KNOX10g> SELECT SUM (sal) FROM scott.emp;

  SUM(SAL)
----------
     29025

scott@KNOX10g> -- This will not audit since ename = user

scott@KNOX10g> SELECT sal FROM emp
  2   WHERE ename = 'SCOTT';
```

```
       SAL
----------
      3000

scott@KNOX10g> -- Direct query on column will audit
scott@KNOX10g> SELECT empno, job, sal
  2    FROM emp
  3   WHERE deptno = 10;

     EMPNO JOB              SAL
---------- --------- ----------
      7782 MANAGER         2450
      7839 PRESIDENT       5000
      7934 CLERK           1300

scott@KNOX10g> -- No Audit since SAL column is not queried
scott@KNOX10g> SELECT empno, job
  2    FROM emp
  3   WHERE deptno = 10;

     EMPNO JOB
---------- ---------
      7782 MANAGER
      7839 PRESIDENT
      7934 CLERK

scott@KNOX10g> -- No audit since no records are returned
scott@KNOX10g> SELECT ename FROM emp
  2   WHERE ename = 'KNOX';

no rows selected
```

Viewing the audit records, you see that the first two queries were audited as expected:



```
sec_mgr@KNOX10g> EXEC show_aud

PL/SQL procedure successfully completed.

sec_mgr@KNOX10g> EXEC show_aud
Who:  SCOTT
What: SCOTT.EMP
Where: 192.168.0.100:SQLPLUS.EXE
When:  Mar-24 19:11
How:   SELECT SUM (sal) FROM scott.emp
-------------- End of Record --------------
Who:  SCOTT
What: SCOTT.EMP
Where: 192.168.0.100:SQLPLUS.EXE
When:  Mar-24 19:11
How:   SELECT empno, job, sal
```

```
  FROM emp
 WHERE deptno = 10
-------------- End of Record --------------

PL/SQL procedure successfully completed.
```

The value provided by the SQL capture shows that the first query really wasn't harmful. There was no direct way for the user to determine the employee's salary.

The lack of audit on the last query verifies that your auditing is selective to your condition and column. There is an important and the often overlooked reason for this: *fine-grained audits occur only when at least one record is returned*. There is no "WHENEVER NOT SUCCESSFUL" in FGA. At first this might seem like a bad idea. After all, there is no way to capture the equivalent of someone banging on a locked door. Unlike standard auditing, a user's failed access doesn't create an FGA audit record. However, this fact is very desirable when you consider that the audits only occur when something of interest has happened. In the preceding example, if there is a record in the audit trail, then the user did access another user's salary.

Capturing SQL

The third jewel brought by FGA was the introduction of a concept called SQL capture. This concept has been extended to standard auditing in Oracle Database 10*g* when you have the AUDIT_TRAIL set to DB_EXTENEDED as was seen in the previous examples. It includes capturing the SQL text as well as the values for any bind variables.

The SQL capture capability is enormous in auditing. It clearly shows what statement caused the audit record to be written. That has an important translation to the real world. First, the queries often indicate the user's intentions. If the query ends with "where ename = 'KING'" then you know the user was targeting KING's records.

The audited SQL can help the security administrator better conclude the actual method the user used when the audit was triggered. Ad hoc queries tend to look much different than packaged application queries, which in turn look different than queries issued by database tools. Administrators can sometimes identify the source by simply looking at the SQL construction.

This is also an invaluable tool because FGA and standard auditing can be used to validate that applications are issuing proper queries. Many times a user's access, while originally believed to be malicious, may in fact have occurred because of a poorly written application. The SQL can show that the user's query was written by the application. Therefore, the user either compromised the application, or the application has a bug. Either way, the audit has provided some very useful and practical information.

A caveat to the SQL capture is that it doesn't capture the modified SQL that results from an applied Virtual Private Database policy.

Acting on the Audit

FGA adds the ability to invoke an event handler. In standard auditing, the records were written as they occurred, but there is no guarantee that they will be viewed and acted upon in a timely fashion. It's possible that the DBA leaves at 6:00 P.M. Friday night. A hacker begins to poke around the system ten minutes later. Finally, on Sunday afternoon, the hacker has gathered enough information to do something destructive. The audit records were being written. Unfortunately, no one was around to hear the screams coming from the database. This is like having a burglar alarm that no one hears. The DBA returns Monday to a compromised database.

It would have been nice if the DBA could have been alerted that something was happening while that something was happening.

With FGA, you do have the ability to invoke an event handler. In this manner, you can see FGA as analogous to a SELECT trigger. The event handler can do anything you like because you write the code.

One thing the event handler shouldn't do is to write (redundant) audit data to a private audit table. I've seen this countless times as people fail to realize that the audit data is already being written. A variation of the theme is a good idea: you may find it useful to write the audit data to another database or to the file system in hopes of protecting the data from a malicious and privileged database administrator.

Writing the Event Handler

The event handler that is used in FGA accepts three parameters: the schema of the object being audited, the name of the object being audited, and the name of the auditing policy. Within the procedure, you can easily access the SQL that caused the auditing to occur. This is obtained by referencing the USERENV context called CURRENT_SQL.

To illustrate an implementation, the following example shows an event handler that sends an e-mail alerting the receiver of the audit. In the e-mail, the details of the audit record are given. The trick is that you don't have to query the audit logs to get the SQL that caused the audit to occur.

```
sec_mgr@KNOX10g> CREATE OR REPLACE PROCEDURE fga_notify (
  2    object_schema  VARCHAR2,
  3    object_name    VARCHAR2,
  4    policy_name    VARCHAR2)
  5  AS
  6    l_message    VARCHAR2 (32767);
  7    l_mailhost   VARCHAR2 (30)
  8                    := '<your smtp server>';
  9    l_mail_conn  UTL_SMTP.connection;
 10    l_from       VARCHAR2 (30)
 11                          := 'FGA_ADMIN@<your email domain>';
 12    l_to         VARCHAR2 (30)
 13                         := '<administrator email address>';
 14  BEGIN
 15    l_message :=
 16         'User '
 17      || USER
 18      || ' successfully accessed '
 19      || object_schema
 20      || '.'
 21      || object_name
 22      || ' at '
 23      || TO_CHAR (SYSDATE, 'Month DD HH24:MI:SS')
 24      || ' with this statement:  "'
 25      || SYS_CONTEXT ('userenv', 'current_sql')
 26      || '"';
 27    l_mail_conn :=
 28           UTL_SMTP.open_connection (l_mailhost, 25);
```

```
 29    UTL_SMTP.helo (l_mail_conn, l_mailhost);
 30    UTL_SMTP.mail (l_mail_conn, l_from);
 31    UTL_SMTP.rcpt (l_mail_conn, l_to);
 32    UTL_SMTP.DATA (l_mail_conn,
 33                      UTL_TCP.crlf
 34                   || 'Subject: FGA Alert'
 35                   || UTL_TCP.crlf
 36                   || 'To: '
 37                   || l_to
 38                   || UTL_TCP.crlf
 39                   || l_message);
 40    UTL_SMTP.quit (l_mail_conn);
 41  EXCEPTION
 42    WHEN OTHERS
 43    THEN
 44      UTL_SMTP.quit (l_mail_conn);
 45      raise_application_error
 46          (-20000,
 47              'Failed due to the following error: '
 48           || SQLERRM);
 49  END;
 50  /

Procedure created.
```

Now register this as the event handler for our FGA policy:

```
sec_mgr@KNOX10g> BEGIN
  2    DBMS_FGA.drop_policy (object_schema    => 'SCOTT',
  3                          object_name      => 'EMP',
  4                          policy_name      => 'Example');
  5    DBMS_FGA.add_policy
  6               (object_schema      => 'SCOTT',
  7                object_name        => 'EMP',
  8                policy_name        => 'Example',
  9                audit_condition    => 'ENAME != USER',
 10                audit_column       => 'SAL',
 11                handler_schema     => 'SEC_MGR',
 12                handler_module     => 'FGA_NOTIFY');
 13  END;
 14  /

PL/SQL procedure successfully completed.
```

As seen in Figure 8-2, when a user queries another user's salary, an e-mail is sent to notify the DBA that this has happened. The DBA should then be able to react to the situation and handle it accordingly. The alerting capability is a critical element to helping administrators respond to incidents as they occur.

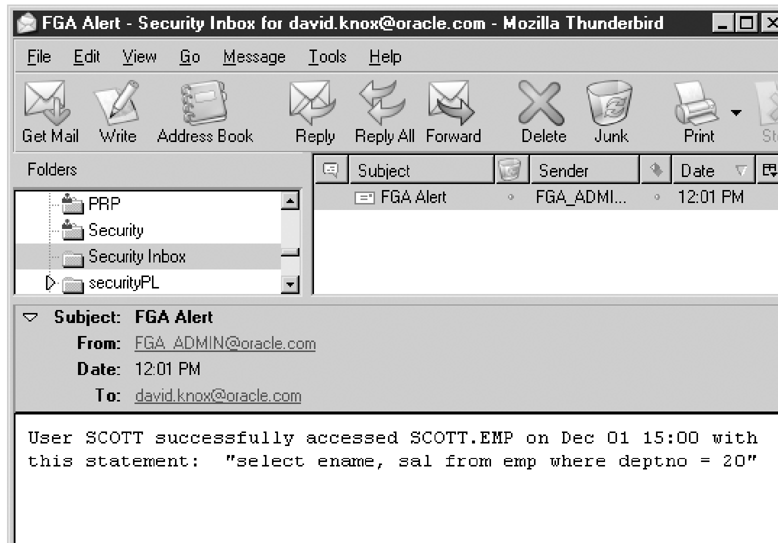


FIGURE 8-2. *Alerts can be sent to administrators to notify them of event occurrences as they happen.*

Invalid Event Handler in Oracle9*i* Database

In Oracle9*i* Database, there is an interesting occurrence that can be witnessed if the event handler throws an exception or is invalid. The database returns a result set that is the opposite of the condition specified. For example, if the condition is "ENAME != USER" meaning you want to audit when a user is accessing another user's records, the database will only return records where "ENAME = USER".

This is known as *failsafe*. The thinking is that the event handler must be doing something critical. Because the event handler can't perform its function, the database will eliminate any potentially harmful records. It does this by reversing the condition.

In Oracle9*i* Database, this was a way to create a column-sensitive, row-level security capability. The following output shows what happens when you apply the same audit policy in an Oracle9*i* Database but have an invalid event handler:

```
system@ORA92> BEGIN
  2    DBMS_FGA.add_policy
  3             (object_schema      => 'SCOTT',
  4              object_name        => 'EMP',
  5              policy_name        => 'Example',
  6              audit_condition    => 'ENAME != USER',
  7              audit_column       => 'SAL',
  8              handler_schema     => 'NO_SCHEMA',
  9              handler_module     => 'UNDEFINED');
```

```
 10  END;
 11  /

PL/SQL procedure successfully completed.

system@ORA92> CONN scott/tiger
Connected.
scott@ORA92> SELECT COUNT (ename) FROM emp;

COUNT(ENAME)
------------
          14

scott@ORA92> SELECT ename, sal FROM emp;

ENAME             SAL
---------- ----------
SCOTT            3000
```

You see that the user can access all records when the SAL column (your column-sensitive attribute) is not queried. Once the audit condition and column are specified, the database tries to invoke the event handler. There is no schema in the database or a procedure as defined by the policy. A trace file is generated indicating the error and the database reverses the condition from "ENAME != USER" to "ENAME = USER". Consequently, the user gets only their record.

In Oracle Database 10*g*, there is native support for column-sensitive, row-level security. As such, the behavior in Oracle Database 10*g* is different. The queries aren't modified with the inverse condition. That is, the queries execute as if no event handler were registered, and all rows are returned even when the SAL column is selected.

Caveats

While FGA is powerful, there are some challenges. First, problems with the event handler may not be obvious when the policy is established. In Oracle9*i* Database, the database reversed the condition in the FGA policy. The results might act as a clue that something has gone awry with your event handler. In Oracle Database 10*g*, there is no clue. The event handler is simply not called.

Recall the feature described earlier that prevents users from executing DML if the function in the FGA policy condition can't be resolved or executed. This can be both helpful and hurtful. Assuming that you don't want this to occur, it may be difficult to detect this ahead of time and difficult to rectify it afterwards.

While the audit provides a wealth of information, it doesn't tell you what the user received as a result from the query. You only know that they got at least one record. This leads to another caveat—audits don't occur when no records are returned. This can be good and bad. It's good for indicating that something bad happened, but it's bad in that there is no indication that someone is banging on a locked door. You can, however, combine standard auditing with FGA to accomplish this task.

Summary

Auditing is a complementary part of the security process. Prevention and access control will always be important in security. However, auditing should not be overlooked. Providing the detection and response capabilities can be equally, if not more, important. Auditing effectively is critical to ensuring that performance and the value received from the auditing are congruent.

Oracle standard and fine-grained auditing (FGA) provide a powerful and secure way to ensure user accountability. Standard auditing allows for auditing to occur in many different ways to meet many different requirements. FGA augments this and allows for a higher fidelity in auditing, which reduces the number of extraneous audit records. The event handler can be used to do many powerful things. The auditing acts as both the detection and response system for the database. Similar to motion detectors that are wired to notify the police, the event handlers in FGA can be used to alert administrators of serious incidents the instant that they occur. Instant notification is essential to guarding the data.

Database auditing provides high auditing assurance because the auditing process can't be circumvented or bypassed. The auditing is consistent regardless of application, query, user, and protocols being used to access the data. When done correctly, the auditing can provide valuable information about the users and their interactions with the database. Of most value may be the fact that you do not have to program the auditing. Oracle has already done this for you.