Fine Grained Access Control

Arup Nanda, Proligence, Inc.

Introduction

Just as a picture is worth thousand words, it's my belief that even the most complex of concepts or features can be easily explained through an example, where the reader has the ability to test out the concepts while reading them online. Following my own convictions, I will explain the concepts and uses of this powerful feature of the database using an example of a hospital database.

The application that the hospital uses must adhere to the HIPAA regulations and one of the mandates is to allow viewing of patient data only to the people who need them and have authorization for. This means the doctors can access the data on just those patients they are treating, not all. In the old days, this could have been done using a simple filtering predicate in the application queries. This is not practical for a variety of reasons – the primary of which is the modification of a large number of SQL statements inside the application. In case of third party canned applications, this is even more difficult, with the source code controlled in a different place outside the control of the hospital.

The other problem is security – what if the user just bypasses the applications and queries the data directly from the database? This sidesteps the filtering predicate inside the code, allowing full access to the data. Obviously the solution calls for the filtering predicate to be applied automatically, regardless of how the access is made.

One of the approaches is to use a view on the table with a filtering predicate built in, and allowing users to access the view instead of the base table. This accomplishes the security requirement, but with the proliferation of the views, it might be impractical to maintain this setup. In addition, if the need ever comes to restrict the access by the owner itself, this solution will not work, since the owner can select from the base table. Using views also forces us to use a predicate that is static, not generated at the runtime.

The perfect solution comes from Oracle's implementation of Fine Grained Access Control (FGAC). It is also known as Row Level Security (RLS) or Virtual Private Database (VPD). In this article we will explore the use of this powerful feature in the form of an example setup, to bolster the understanding. We will also learn how to use another advanced feature – application contexts in a database setup.

Setup

You can download the scripts to create all the objects used in this article and populate the data from my website, www.proligence.com/pubsupp.

In our example, we have used the case of an overly simplified hospital database. Typically hospitals have several doctors and each patient is assigned a doctor. The tables are owned by the schema HOSPITAL. The table DOCTORS holds the information on doctors as follows.

```
Name                 Null?    Type
-------------------- -------- ------------
DOCTOR_ID            NOT NULL NUMBER
DOCTOR_NAME                   VARCHAR2(20)
GROUP_ID                      NUMBER
```

The primary key of the table is DOCTOR_ID. It's assumed that the column DOCTOR_NAME contains the doctor's login id to the database. A group may consist of several doctors. The column GROUP_ID specifies the group the doctor belongs to.

The other important table holds information on patients visiting the hospital, named PATIENTS, shown below.

```
 Name              Null?    Type
 ----------------- -------- ------------
 PATIENT_ID        NOT NULL NUMBER
 DOCTOR_ID                  NUMBER
 PATIENT_NAME               VARCHAR2(20)
 DISEASE                    VARCHAR2(20)
```

The column PATIENT_ID is the primary key of the table. The column DOCTOR_ID is a foreign key to the table DOCTORS shown above. For the purpose of simplicity, let's assume that the relationship between DOCTORS to PATIENTS is one-to-many, whereas in the real life, it is probably many-to-many.

Here is how the data looks like in the table DOCTORS.

```
DOCTOR_ID DOCTOR_NAME            GROUP_ID
---------- -------------------- ----------
         1 DRADAM                        1
         2 DRBARB                        2
         3 DRCHARLIE                     2
```

And here is the data in table PATIENTS.

```
PATIENT_ID  DOCTOR_ID PATIENT_NAME         DISEASE
---------- ---------- -------------------- -----------
         1          1 LARRY                EGO
         2          1 BILL                 CONTROL
         3          2 SCOTT                FICKLENESS
         4          3 CRAIG                LOWVISION
         5          3 LOU                  GREED
```

Corresponding to all the names of the doctors, we need to have the userids created in Oracle in the same name. Therefore, we have the users named DRADAMS, DRBARB and DRCHARLIE, all with SELECT, INSERT, UPDATE and DELETE privileges on the tables DOCTORS and PATIENTS.

Building the FGAC Setup

In the existing application, the following statement is a call made to the database to see the patient data.

```
SELECT * FROM PATIENTS;
```

With the new requirements in mind, the call must be changed to

```
SELECT * FROM PATIENTS WHERE DOCTOR_ID = <id of the doctor logged in>
```

We somehow have to make a system where the application need not be changed, and the first call will automatically select only the records related to the doctor currently logged in, not all. In other words, we have to generate a filtering predicate,

i.e. a WHERE clause to be appended to he query automatically. Building this predicate is the first step. The following function returns a string that can be applied to the query as a WHERE clause.

```
create or replace function get_doctor_id
(
   p_schema_name   in   varchar2,
   p_table_name   in   varchar2
)
return varchar2
is
   l_doctor_id   number;
begin
   select doctor_id
   into l_doctor_id
   from doctors
   where doctor_name = USER;
   return 'doctor_id = '||l_doctor_id;
end;
```

Note how the function returns the string **`DOCTOR_ID =`** <*id*> where <*id*> is the numerical ID of the doctor who is logged in now, as returned by the function call **`USER`** above.

This special function is called a policy function and is the building block of a FGAC setup. Note that this has exactly two input arguments, for the schema and the table on which it will be applied and has exactly one return value, the string that will be used as a WHERE clause. The structure of the policy function must be exactly this and is not flexible. The logic can be changed inside, though.

Next step is to build a policy to be placed on a table. This policy is the one that restricts the rows accessible to the users. It does so by applying the output of the policy function. The following code segment sets up a policy on the table PATIENTS.

```
begin
   dbms_rls.add_policy
   (
      object_schema    => 'HOSPITAL',
      object_name      => 'PATIENTS',
      policy_name       => 'PATIENT_VIEW_POLICY',
      policy_function => 'GET_DOCTOR_ID',
      function_schema => 'HOSPITAL',
      statement_types => 'SELECT, INSERT, UPDATE, DELETE',
      update_check    => true,
      enable      => true
   );
end;
```

Here we have defined a policy named PATIENT_VIEW_POLICY on the table PATIENTS in schema HOSPITAL. The policy calls the function GET_DOCTOR_ID as shown in the parameter **`policy_function`**. This policy is applied to all types of DML statements on the table – SELECT, INSERT, UPDATE and DELETE. The other options will be explained later.

Once the policy is in place, logon to the database as the user DRADAM and select from the table

```
SQL> select * from hospital.patients
```

```
PATIENT_ID  DOCTOR_ID PATIENT_NAME         DISEASE
---------- ---------- -------------------- --------
         1          1 LARRY                EGO
         2          1 BILL                 CONTROL
```

Well what happened? There are only two rows selected from the table; but we know for a fact that the table has five rows. However, only patients 1 and 2 are supposed to be seen by the user DRADAM, and only those were displayed. The most important point to note here is that the user did not specify any where clause. Yet, the filtering predicate was applied automatically, by the policy using the output from the policy function.

The user's original query

```
SELECT * FROM PATIENTS
```

was rewritten to

```
SELECT * FROM
(SELECT * FROM PATIENTS)
WHERE DOCTOR_ID = 1
```

What if the user deletes the table, as seen below?

```
SQL> delete hospital.patients;

2 rows deleted.
```

Note, only 2 rows are deleted, not all the five. The same principle hold true – the filtering predicate is applied automatically to the query. If the user updates the table,

```
SQL> update hospital.patients set disease = null;

2 rows updated.
```

Again, only 2 rows are updated, not all five.

Another user, DRBARB is allowed to see only patient 2. If she logs in and uses the same query:

```
SQL> select * from hospital.patients;

PATIENT_ID  DOCTOR_ID PATIENT_NAME         DISEASE
---------- ---------- -------------------- ------------
         3          2 SCOTT                FICKLENESS
```

Notice how only one row was returned, even though the table has 5. Dr Barb was authorized to see patient 3 only and that is what she saw.

The policy is applied regardless of how the table is accessed – through a trigger, a procedure, an application, anything. It is as if the table contains only two rows for the user DRADAMS, not the five which are actually there. This facility creates a window where the user always sees the rows he is supposed to see, not *all*, something akin to a private view of the table. By applying the same policy to all the tables in a database, the users can see the data which they are authorized to see only, as if they have a private database inside a real database. Hence the Fine Grained Access Control is also known as **Virtual Private Database** feature.

Note how the content of the table changes depending upon the user logged in, eliminating the need for application changes. This powerful feature makes applications developed only once; the policy dictates the filtering predicate.

In addition to example given above, FGAC has other uses in hosting companies, where multiple users share the same database and in some cases the same table. FGAC allows the creation of several *virtual* databases, not physically different ones – making the setup simpler and less expensive to maintain.

Bypassing the Restrictions

Sometimes it might be required to bypass the restrictions. For instance, we may decide to remove restrictions from the owner of the tables. To do this, we will pass a NULL as the return value to be used as a predicate. If the value is NULL, the policy allows all rows to be visible without restrictions. The policy function can then be rewritten as follows.

```
create or replace function get_doctor_id
(
   p_schema_name   in   varchar2,
   p_table_name   in   varchar2
)
return varchar2
is
   l_doctor_id   number;
begin
   if (p_schema_name = USER) then
      return null;
   end if;
   select doctor_id
   into l_doctor_id
   from doctors
   where doctor_name = USER;
   return 'doctor_id = '||l_doctor_id;
end;
```

Note the newly added segment shown in bold. If the current user is the schema owner of the table, then the policy function returns NULL, i.e. no filtering predicate is applied to the query.

There is a special type of system privilege called `EXEMPT ACCESS POLICY` used to exempt a specific user from being subject to any kind of restriction on any of the tables. The DBA role is granted this system privilege by default; hence a user with DBA role accessing data will not be restricted by any policy – a very important point to keep in mind while designing applications involving FGAC.

Other Dependent Tables

So far we have talked about only one table. Suppose there is another table in the database to hold information on treatments given to patients. Since a patient may be given treatment more than once, the table TREATMENTS, shown below, has column to record the dates, too.

```
 Name              Null?    Type
 ----------------- -------- ------------
 PATIENT_ID                 NUMBER
 TREATMENT_DT               DATE
 TREATMENT                  VARCHAR2(30)
```

Here are the sample data in this table.

```
PATIENT_ID TREATMENT TREATMENT
---------- --------- ------------------------------
         1 19-OCT-03 ASK THE ORACLE
         1 29-OCT-03 SELL THE SAILBOAT
         2 19-OCT-03 LOOK AT THE WINDOWS
         2 29-OCT-03 GIVE S/W AWAY
         3 19-OCT-03 LOOK AT THE SETTING SUN
         4 29-OCT-03 GENERATE INTEL-LIGENCE
         4 29-OCT-03 PROCESS MORE
         5 29-OCT-03 STOP MACHINES IN BUSINESS
         5 29-OCT-03 STOP INTERNATIONAL MACHINES
```

Applying the same principle of the FGAC, a user should be able to see only the rows allowed to be seen. For instance, Dr Adam is allowed to only records for his patients, 1 and 2, not for all the other patients. Earlier we used DOCTOR_ID as a filtering condition; but this table does not have the column. Therefore we have to devise a separate filtering condition.

```
WHERE PATIENT_ID IN (SELECT PATIENT_ID FROM PATIENTS)
```

Since the table PATIENTS is restricted, the list of PATIENT_IDs will be restricted too and hence this filtering condition will work. We need to define a new policy function for this policy.

```
create or replace function get_patient_id
(
   p_schema_name  in   varchar2,
   p_table_name   in   varchar2
)
return varchar2
is
   l_patient_id   number;
begin
   if (p_schema_name = USER) then
      return null;
   end if;
   return 'patient_id in (select patient_id from patients)';
end;
```

This returns the predicate in the format we want. Then we need to create a policy for this table using this function.

```
begin
   dbms_rls.add_policy
   (
      object_schema   => 'HOSPITAL',
      object_name     => 'TREATMENTS',
      policy_name     => 'TREAT_VIEW_POLICY',
      policy_function => 'GET_PATIENT_ID',
      function_schema => 'HOSPITAL',
      statement_types => 'SELECT, INSERT, UPDATE, DELETE',
      update_check    => true,
      enable          => true
   );
end;
```

This policy will restrict the table TRATMENTS in such a way that only the authorized rows are visible.

Update and Insert Checks

So far we have talked about restricting the policy on the value of a column name. The user DRBARB is allowed to see only patient id 3. However, what if she issues the following update?

```
SQL> update hospital.patients set doctor_id = 3;
update hospital.patients set doctor_id = 3
                *
ERROR at line 1:
ORA-28115: policy with check option violation
```

Note the new type of error - `ORA-28115: policy with check option violation`. It's raised since the user attempted to change the very column that the table is restricted on. The user is allowed to see only records with DOCTOR_ID = 2, changing to 3 would have removed the record from her private view of the table. The parameter `update_check` in the packaged procedure `dbms_rls.add_policy` prevents this from happening. Setting this parameter to FALSE will allow this update and place the row in a set inaccessible by the user.

The same thing happens when the user tries to insert a row that will not satisfy the policy for her viewing. Dr Barb tries to insert the row with DOCTOR_ID = 3, but she is allowed to see only DOCTOR_ID = 2.

```
SQL> insert into hospital.patients values (6,3,'CARLY','OBSESSION');
insert into hospital.patients values (6,3,'CARLY','OBSESSION')
                    *
ERROR at line 1:
ORA-28115: policy with check option violation
```

The same error is produced here, too.

Multiple Policies

In real life, you should probably create individual procedures for each type of access. In that case the parameter `statement_types` should contain only the statement type it should be applied on, e.g. 'INSERT', not the list of all the statement types. If a table has multiple policies, the predicates are AND*ed*, i.e. the predicates are all appended to the original query with AND. For instance the query

```
SELECT * FROM PATIENTS
```

is rewritten to

```
SELECT * FROM
(SELECT * FROM PATIENTS)
WHERE <predicate clause from first policy>
AND <predicate clause from second policy>
AND <predicate clause from third policy>
.....
... and so on....
```

Defining multiple policies makes administration tasks easier – you can enable or disable specific policies, based on need and each policy can have a different policy function. The latter is useful when the access patterns are different for each statement types. For instance, you could have a user perform DML on his or her patient records, but other doctors in the

same group can also have SELECT access to the records. In this case you have to define a policy for SELECT statement type and another for the other types.

Policy Management

The previous examples showed how to add a policy. To drop the policy, the procedure `dbms_rls.drop_policy` can be used. It takes the schema, table and policy names as parameters. The policies can be temporarily disabled by using the API `dbms_rls.enable_policy`. It takes, in addition to these three, another parameter enable. To disable a policy, call this with the parameter set to FALSE. To enable it, set the parameter to TRUE and execute the function.

Policy can also be managed through an Oracle Enterprise Manager component known as Oracle Policy Manager, which provides all these functions via a GUI interface. To invoke it, just issue the following in the command line

```
oemapp opm
```

This will bring up the OPM main screen. The operations are easy to follow and intuitive.

Several data dictionary views are present for information on FGAC policies. The most common one is **DBA_POLICIES**. Here is a brief description of the most important columns of the view.

Column	Description
OBJECT_OWNER	The schema owner who owns the object on which the policy is defined.
OBJECT_NAME	The name of the object
POLICY_NAME	Name of the policy
PF_OWNER	The owner of the function used to enforce security of the policy.
PACKAGE	Sometimes a package used to enforce the policy. This shows the name of the package if that is the case.
FUNCTION	The name of the function that is used to enforce security.
SEL	Indicates whether the policy applies to select statements. A value of YES indicates that, otherwise NO.
INS	Same explanation as above, but for inserts.
UPD	Same explanation as above, but for updates.
DEL	Same explanation as above, but for deletes.
CHK_OPTION	If the policy has been defined with an update check that will make sure the new value also conforms to the policy after the change. A value of YES indicates that, otherwise NO.
ENABLE	Indicates whether the policy is enabled. A value of YES indicates that, otherwise NO.
STATIC_POLICY	Indicates whether the policy is static, i.e. the function returns the same value regardless of the

user who calls it.

Troubleshooting and Potential Concerns

Before proceeding further on the advanced topics, let's examine some of the potential problems and their resolution. Most of these errors produce a trace file in the `user_dump_dest` directory, which contains useful information for further diagnosis. The most common problem is a badly constructed predicate generated by the policy function throwing error `ORA-28113: policy predicate has error`. This occurs when the policy function generates a predicate that is syntactically incorrect and the actual statement will be visible in the trace file. For instance, the following trace file shows the problem statement.

```
PATIENT_ID IN ()
ORA-00936: missing expression
```

Note there is nothing inside the parentheses, which is wrong and hence the policy failed to apply. Further diagnosis must be done on the policy function on how to correct this error.

Another error is "`ORA-28112: failed to execute policy function`", which indicates that the policy function failed at the runtime, typically due to unhandled exceptions. Luckily, the trace files generated show the exact problem. A less common error is "`ORA-28110: Policy function or package has error`", which simply indicates that the policy function is not valid. Mere recompilation of the function will resolve the problem.

Since FGAC works by applying a predicate to the query at runtime, they cannot applied in operations that bypass the SQL processing layer, such as Direct Path export/import, Direct Path SQL*Loader, Direct Path Inserts, etc. In the last three cases, the operation fails with an error ORA-28116: insufficient privileges to do direct path access. The only options are using the conventional mode of these tools, temporarily disabling the policy or using a user with EXEMPT ACCESS POLICY system privilege. Direct path export does not fail but export is carried out in conventional mode, even if DIRECT=Y is given. This can be seen from the informational message on the screen while export is going on.

Debugging

It might be helpful, especially during development, to see exactly the kind of predicate returned by FGAC. You can do that in a few different ways. The simplest is to use a query like this.

```
select sql_text, predicate, policy, object_name
from v$sqlarea , v$vpd_policy
where hash_value = sql_hash
/

SQL_TEXT
--------------------------------------------
PREDICATE
--------------------------------------------
POLICY                         OBJECT_NAME
------------------------------ --------------
select * from hospital.patients
patient_id in (1,2,5)
PATIENT_VIEW_POLICY            PATIENTS
```

This shows that the predicate `where patient_id in (1,2,5)` was appended to the original query `select * from hospital.patients`. This is useful if the original query is still in the SQL area in SGA and not flushed out yet. The view v$vpd_policy also shows a lot of other details relevant for analysis, such as the group, in case of a grouped policy, etc.

The other option is to use an event for throwing out the details of the rewritten query. As the user DRADAM, use the following command before selecting from the table.

```
alter session set events '10730 trace name context forever, level 12';
```

Then select from the table. A trace file will be generated in the user_dump_dest directory, an excerpt of which is shown below.

```
*** 2003-11-19 17:32:11.508
*** SESSION ID:(23.1512) 2003-11-19 17:32:11.508
------------------------------------------------------------
Logon user     : DRADAM
Table/View     : HOSPITAL.PATIENTS
Policy name    : PATIENT_VIEW_POLICY
Policy function: HOSPITAL.GET_PATIENT_ID
RLS view :
SELECT  "PATIENT_ID","DOCTOR_ID","PATIENT_NAME","DISEASE" FROM "HOSPITAL"."PATIENTS"
"PATIENTS" WHERE (patient_id in (1,2,5))
```

Note the changed query is neatly shown in the trace file. This shows exactly which predicate was generated and applied. This is particularly useful for multiple policies applied on a table – all the predicates are visible. If the trace file is not generated, try refreshing the policy using `dbms_rls.refresh_policy` packaged procedure.

Application Contexts

The above example relies on the assumption that the user accesses are limited by userid in Oracle. In some cases, especially in web applications, the application connects to the database using a single generic userid, e.g. APPUSER. The users such as DrAdam, DrBarb and DrCharlie are users in application, not the database. As far as the database is concerned, the userid is APPUSER, in all cases. Hence the function USER in the policy functions defined above always returns APPUSER and the FGAC cannot be established.

To alleviate the problem, another approach can be used using application contexts. App Contexts, simply put, are similar to global variables, whose value once set is available in the same session. Another session can have a different value. An App Context is actually a collection of such variables, known as Attributes. Attributes are to a context what columns are to a table. However, unlike columns, the attributes are not defined during creation of the context; rather anyone can set any context attribute and set a value. Also, contexts are not stored objects and they don't consume space.

To create a context, the user should have CREATE ANY CONTEXT system privilege. After granting this privilege to HOSPITAL, connect to the database as HOSPITAL and issue

```
create context app_ctx using set_app_ctx;
```

Note the clause, **`using set_app_ctx`**, which indicates that the attributes in this context can be assigned only by executing a procedure called set_app_ctx. The next step is to create the procedure.

```
create or replace procedure set_app_ctx
(
```

```
      p_app_user in varchar2
)
is
begin
      dbms_Session.set_context('app_ctx','app_userid',p_app_user);
end;
```

Note, the above segment assigns to the attribute app_userid of the context app_ctx the value of the input parameter p_app_user. While creating the context, we have not created any attribute; it was done during the call to dbms_Session package, which is used to set the application contexts. Let's set the value of the context as follows.

```
SQL> exec hospital.set_app_ctx ('Context1')

PL/SQL procedure successfully completed.
```

Once an application context is set, it can be retrieved in several ways. The following query shows all the context attributes set in the session.

```
SQL> select * from session_context;

NAMESPACE      ATTRIBUTE      VALUE
-------------- -------------- ----------------
APP_CTX        APP_USERID     Context1
```

Note, the attribute app_userid has been assigned the value Context1 as shown above. The context is referred to as NAMESPACE, too, as shown above. Another method to retrieve the attribute value is the following query

```
SQL> select sys_context('app_ctx','app_userid') from dual;

SYS_CONTEXT('APP_CTX','APP_USERID')
------------------------------------------------------------
Context1
```

This method of accessing context attributes is also used in other areas. Remember the function USER

```
Select USER from dual;
```

It returns the current logged-in user. The above can also be written as

```
Select SYS_CONTEXT('USERENV','CURRENT_USER') from dual;
```

Note the use of the SYS_CONTEXT function. USERENV is a predefined context namespace and an attribute CURRENT_USER is populated automatically.

Remember, the function set_app_ctx sets the value of the attribute, which, inside, calls the package dbms_session.set_context. If the user calls this package directly,

```
SQL> begin
  2  dbms_Session.set_context('app_ctx','app_userid','Context1');
  3  end;
  4  /
begin
*
ERROR at line 1:
```

```
ORA-01031: insufficient privileges
ORA-06512: at "SYS.DBMS_SESSION", line 78
ORA-06512: at line 2
```

Note the error produced - `ORA-01031: insufficient privileges`. Even after granting the execute privileges on the package to the user HOSPITAL, this error persists. Why so? The reason is the way the context is declared: `create context app_ctx using set_app_ctx`, which means the attributes of this context can only be set using its trusted procedure, set_app_ctx, no other way.

This important property of the attribute is useful in securing an application. Since the only way the attribute values can be set is through the procedure, we can place all kinds of security checks inside the procedure to make sure that the context setting is valid.

Application Users and FGAC

Going back to the problem of the FGAC in a situation where the database user is generic, but the application users must be passed to the FGAC setup for building a correct set of rows, we can use the application contexts. In the above example of contexts, the value of the attribute APP_USERID can be set by the application to the real user and the FGAC policy function can retrieve the value to build the correct FGAC version of the table.

```
create or replace function get_doctor_id
(
   p_schema_name   in   varchar2,
   p_table_name   in   varchar2
)
return varchar2
is
   l_doctor_id   number;
begin
   if (p_schema_name = USER) then
      return null;
   end if;
   select doctor_id
   into l_doctor_id
   from doctors
   where doctor_name in (USER, sys_context('app_ctx','app_userid'));
   return 'doctor_id = '||l_doctor_id;
end;
```

Note the line where it says

```
where doctor_name in (USER, sys_context('app_ctx','app_userid'));
```

Here the **`doctor_name`** column is not only matched against the database userid, but against the value of the attribute APP_USERID in context APP_CTX, too.

The generic userid used is APPUSER, which is then given execute privilege on the procedure `set_app_ctx`. Now, the application connects as APPUSER and sets the context attribute to the value of the real user, say DRBARB, and selects from the table.

```
SQL> exec hospital.set_app_ctx('DRBARB')

PL/SQL procedure successfully completed.
```

```
SQL>select * from hospital.patients

PATIENT_ID  DOCTOR_ID PATIENT_NAME         DISEASE
---------- ---------- -------------------- ---------------
         3          2 SCOTT                FICKLENESS
```

Note how only the record which DRBARB is supposed to see is shown, not all. Now change the app user to DRCHARLIE and do the same.

```
SQL> exec hospital.set_app_ctx('DRCHARLIE')

PL/SQL procedure successfully completed.

SQL> select * from hospital.patients

PATIENT_ID  DOCTOR_ID PATIENT_NAME         DISEASE
---------- ---------- -------------------- -------------------
         4          3 CRAIG                LOWVISION
         5          3 LOU                  GREED
```

We got it again! Dr. Charlie is supposed to see these patients, too.

In this way, we can set a userid to an attribute of an application context and use it inside a policy function to simulate database users. However, the power of application context does not end here. An attribute can be setup in such a way that it old the entire predicate, not just a few discrete values. It can also hold other values which may or may not be related to application users. One such example is accessibility of patients unrelated to the DOCTOR_ID column. In our example, we now have a requirement that the doctors can access data based on some authorization, not necessarily their own patients. We now have a new table called DOCTOR_AUTHORITY as follows

```
 DOCTOR_ID PATIENT_ID
---------- ----------
         1          1
         1          2
         1          5
```

It means doctor 1 can see the patients 1, 2 and 5. Even though the doctor_id value for the patient 5 is 3, doctor 1 can still see the patient. This requires a change in the policy function as follows.

```
create or replace function get_patient_id
(
   p_schema in varchar2,
   p_table in varchar2
)
return varchar
is
   l_ret varchar2(2000);
begin
   if (p_schema = USER) then
            l_ret := NULL;
   else
            l_ret := 'patient_id in ('||
            sys_context ('app_ctx','patient_id_list')||
            ')';
   end if;
```

```
   return l_ret;
end;
```

Here, the predicate is constructed from a list selected from the context attribute patient_id_list. This list can be constructed by this function.

```
create or replace function create_patient_id_list
(
   p_doctor_id     in number
)
return varchar2
is
   l_ret   varchar2(200) := null;
begin
   for pat_rec in
   (
           select patient_id
           from doctor_authority
           where doctor_id = p_doctor_id
   ) loop
           l_ret := l_ret || ','||pat_rec.patient_id;
   end loop;
   return ltrim(rtrim(l_ret,','),',');
end;
```

We also have to modify the set_app_ctx procedure to set the new attribute.

```
create or replace procedure set_app_ctx
(
   p_patient_id_list in varchar2
)
is
begin
   dbms_Session.set_context('app_ctx','patient_id_list',p_patient_id_list);
end;
```

The context should be set automatically and this can be done using a logon system trigger.

```
create or replace trigger set_patlist_ctx
after logon on database
declare
   l_doctor_id   number;
begin
   begin
      select doctor_id
      into   l_doctor_id
      from hospital.doctors
      where doctor_name = USER;
   exception
      when NO_DATA_FOUND then
         null;
   end;
   hospital.set_app_ctx (hospital.create_patient_id_list(l_doctor_id));
end;
```

The policy has to be changed to use the new policy function. First drop the policy

```
Connect HOSPITAL/HOSPITAL
begin
   dbms_rls.drop_policy
   (
       object_schema   => 'HOSPITAL',
       object_name     => 'PATIENTS',
       policy_name     => 'PATIENT_VIEW_POLICY'
   );
end;
```

Recreate the policy using the new function.

```
begin
   dbms_rls.add_policy
   (
      object_schema   => 'HOSPITAL',
      object_name     => 'PATIENTS',
      policy_name     => 'PATIENT_VIEW_POLICY',
      policy_function => 'GET_PATIENT_ID',
      function_schema => 'HOSPITAL',
      statement_types => 'SELECT, INSERT, UPDATE, DELETE',
      update_check    => true,
      enable          => true
   );
end;
```

After this is setup, test the actual workings. Logon as DRADAM.

```
SQL> select * from session_context
  2> /

NAMESPACE         ATTRIBUTE           VALUE
----------------- ------------------- --------------------
APP_CTX           PATIENT_ID_LIST     1,2,5
```

Voila! It worked. The application context was properly set. Now select from the table.

```
SQL> select * from hospital.patients;

PATIENT_ID  DOCTOR_ID PATIENT_NAME         DISEASE
---------- ---------- -------------------- ----------
         5          3 LOU                  GREED
         2          1 BILL                 CONTROL
         1          1 LARRY                EGO
```

And it worked! All the rows that are supposed to be shown are shown in this selection, as expected.

Oracle 10g Enhancements

Oracle 10g offers significant enhancements in this area, particularly in selective control based on columns and different types of policies to apply in different situations to enhance performance.

Relevant Columns

In the Oracle 9i and below, the policy is applied on the table unconditionally, regardless of which columns are being accessed. Consider these three queries

```
1. SELECT COUNT(*) FROM PATIENTS
2. SELECT PATIENT_ID FROM PATIENTS
3. SELECT SOCIAL_SEC_NO FROM PATIENTS
```

The first one does not access any specific record, just an aggregation for reporting or other purpose. In some situations, the user may be allowed to access all the records of the table PATIENTS in doing a COUNT(*), as it does not violate privacy regulations or practices. However, the same user should not be allowed unrestricted in the second query, where a specific element of the record, the column PATIENT_ID is accessed. In the third case, the query tries to fetch Social Security Number of a patient, a piece of information protected by a regulation known as HIPAA. In HIPAA compliant databases, the third query is restricted more than the second query.

As you can see, there is a need for differing restrictiveness based on the column being accessed. In the first case we need no restriction as no columns are accessed; and in the second and third cases we want to have progressively more restrictive policies. Oracle 10g allows that through the use of column declaration in policies.

```
begin
    dbms_rls.add_policy (
        object_schema     => 'HOSPITAL',
        object_name       => 'PATIENTS',
        policy_name       => 'PATIENT_VIEW',
        function_schema   => 'HOSPITAL',
        policy_function   => 'GET_PATIENT_ID',
        statement_types   =>  'INSERT, UPDATE, DELETE, SELECT',
        update_check      => TRUE,
        sec_relevant_cols => 'PATIENT_ID'
    );
end;
```

Note the new parameter `sec_relevant_cols => 'PATIENT_ID'`, which indicates that this policy should be applied only when the query accesses the column PATIENT_ID, not every time. With this type of policy in place, when DRADAM issues the query SELECT PATIENT_NAME FROM PATIENTS, he sees 5 records, which is actually the number of records in the table. However, when he issues SELECT PATIENT_ID FROM PATIENTS, he sees only 2, the only two he is authorized to see.

This is a very important feature in FGAC and can be used in myriads of cases where data must be made accessible only selectively, not universally. At the same time, you must also be careful to avoid confusion and mistakes. Since the policy restrictions are based on the columns being accessed, the results will vary and may give wrong results inadvertently.

Policy Types

In the example given for Oracle 9i, the policy function is executed every time the policy is invoked. Sometimes this may not be necessary, as in case of a hosting company where the outcome of the policy function for a user is same, regardless of any other factor such as time of the day, the value of some other attributes, etc. In this case, re-evaluation of the policy function is not necessary and causes performance degradation. To address this issue, Oracle 10g has introduced a new type of policy called **static policy**, where the function is not executed during every call; it is executed only once, during parse phase and then the result is cached for future reference.

Static policy is indicated by the new parameter in `dbms_rls.add_policy` procedure, `policy_type => dbms_rls.static`. The parameter `static_policy` present in Oracle 9i is not required anymore. However, if the value of parameter policy_type is not set, the policy is set as per the setting of the other parameter static_policy. If static_policy parameter is set to TRUE, then the policy is static and dynamic otherwise. If both static_policy and policy_type are set, the policy_type takes precedence. Here is a complete listing of policy types that can be passed to the parameter policy_type. All these should be prefixed by dbms_rls, since they are actually constants defined inside the package.

- **dynamic**
  The policy function is always executed, during every call. This is the default.
- **context_sensitive**
  The policy function is executed only is the program context changes. An excellent example is in an web application where the single database connection is used by multiple sessions of the application. Every time the database session passes from one session to the other, the context changes and the policy function is evaluated.
- **shared_context_sensitive**
  The behavior is the same as the above, but it can be applied to more than one object.
- **static**
  This has been described above.
- **shared_static**
  Same as above, but it can be applied to more than one object.

As of writing of the article, Oracle 10g was yet to be released and these names were being finalized. It is quite possible that some of the names may be different in the version released for production.

Conclusion

Fine Grained Access Control allows an existing database to be partitioned into several virtual databases that can be shown differently to different users. The filtering of rows inside the table can be controlled by a user defined policy function, and the string generated by the function is applied automatically to every query on the table, regardless where the user connects from or with. This adds security to the application by showing only those rows the users are authorized for, and also helps in maintaining an application where a separate data store is not needed.

In my book Oracle Privacy Security and Auditing, by Rampant TechPress, I have discussed these features in more detail and explained how to build a secure application authentication system using contexts and FGAC.

About the Author

Arup Nanda (arup@proligence.com) has been an Oracle DBA for more than 10 years, working on almost kinds of features, problems, challenges and nuances the Oracle database offers. He is the founder of Proligence (www.proligence.com) , a New York area company offering highly specialized Oracle database solutions, security training and auditing to customers. He has written several articles in magazines such as Oracle Magazine, Select Journal; spoken at many conferences including Oracle World and IOUG Live and is the co-author of the book *Oracle Privacy Security and Auditing*. His involvement in Oracle user community spans from being an editor of Select Journal – the International Oracle User Group publication – to being a Director of Connecticut Oracle User Group. Recognizing his professional accomplishments and contributions to the community, Oracle honored him with the *DBA of the Year* award in 2003.