SecurityFocus™

Oracle Row Level Security: Part 1

by Pete Finnigan
last updated November 7, 2003

In this short paper I want to explore the rather interesting *row level security* feature added to Oracle 8i and above, starting with version 8.1.5. This functionality has been described as *fine grained access control* or *row level security* or *virtual private databases* but they all essentially mean the same thing. We will come back to this shortly but before we do that lets get to what this paper is about. This paper is meant as an overview; a taster in fact of what *row level security* can be used for and how it can be used, with some simple examples to illustrate. I want to also discuss some of the issues with row level security. Finally, I also want to show how to view what row level security components have been implemented in the database and also touch on how to view how the actual database queries are altered by the row level security functionality in the oracle optimizer.

The example program code used in this paper is available at http://www.petefinnigan.com/sql.htm.

So many names

There seem to be many conflicting names for Row Level Security throughout its existence in Oracle. Why is this? Or more importantly does it really matter to us, the users of this technology? Of course not! The API package interface is called DBMS_RLS (for row level security) so we will stick to that for now.

There is also another name bandied about - *label security* - this is in fact another piece of functionality and is a replacement for the old *trusted Oracle* product. Label security is used by the more security minded customers of Oracle who are interested in higher levels of data protection and is an add-on for Oracle enterprise edition. A GUI tool called *policy manager* was added in 9i as part of *enterprise manager* and allows the setting up of label security. Label security is implemented on top of VPD. For label security the programs are pre-created and are implemented on a single column value. It is this column that represents the label. Because Oracle has already coded the functionality atop of VPD, label security can be implemented without programming expertise.

One other name that gets associated with virtual private databases, fine grained access control and row level security is fine grained auditing. This has nothing to do with fine grained access other than it works in a similar way and is managed by an API package DBMS_FGA and allows auditing to work at the row level. Fine grained auditing will not be discussed further here.

What can row level security be used for?

Why would a user of Oracle's database products want to use the row level security functionality? Well one of the main uses is to allow all of the data to be stored in one database for different departments or even for a hosting company to store data for different companies in one database. Previously this could have been done with Oracle by using either database views or database triggers or a combination of both. In general this leads to complex applications with a lot of code repetition.

If an application needed to cater to a number of departments that should only be able to access differing sets of data then a set of views would be created for each group of business users. These would have hard coded *where clauses* that implemented the business rules. Instead, database triggers would be utilized to cater for data manipulations. Grouping business users together to be able to use these sets of views and triggers tended to lead to the use of shared accounts.

Maintenance becomes difficult as adding a new business group, or in the case of multiple company's data in the same database, leads to the need to replicate and alter a whole set of views, triggers and associated code.

Auditing of individual user actions becomes a problem with the built in audit features as users share database accounts. To enable business users to be audited a whole set of authentication, application roles and audit features are generally implemented by the developers.

Where is all of this code? It could be in the client applications, or more recently in a middle tier application server. Maintenance is clearly a big issue! So is security - what would happen if a user connected to the database with a tool such as *SQL*Plus* or even MS Excel via ODBC and used one of the shared accounts? When applications control the differing business roles from manager down to lowest data in-putter then there is an issue. The shared database account needs to be able to see and use all of the functionality for every application role. So, when a direct connection is made to the database using one of these accounts a security hole exists.

The scenario I have painted is just one possibility; there are many others and similar issues will be obvious in those as well. Row level security can be a solution to some of these problems. Here are a few of the advantages of row level security:

- Oracle's row level security provides a great improvement for this type of application where many users must connect to the same data but be segregated based on what parts of that data they are allowed to view and edit.
- Maintenance becomes easier as now the business rules and security implementations are done through one PL/SQL procedure per table instead of being spread throughout the applications code.
- It should be possible to retro-fit row level security to an existing application due to the fact that it is implemented on the server as close to the actual data as possible.
- Because row level security is implemented as close to the data as possible, the loophole of accessing the data with a share account from a tool such as *SQL*Plus* is solved.
- The issue of having to use shared accounts is no longer a problem as application roles / groups of users do not need to be segregated for the purpose of hard coding views onto the data. Row level security can be made to work with shared accounts though, if needed.
- Auditing can now be done more easily using Oracles built in features.
- Security policies can be associated with both database base tables and also database views.
- Using row level security makes the application more manageable due to simpler designs and less potential code.
- Row level security provides a protection against ad-hoc queries as the tool does not matter anymore; the data is secured for everyone at the source.

To close this section, it is worth noting that row level security does not make the old methods totally redundant, they can still be valid and be the best solution in some cases.

So how does it work - a brief example

Row level security is based around the idea of having a defined security policy function that is attached to a database table or view execute each time data in the table or view is queried or altered. This function returns an additional piece of SQL called a predicate that is attached to the original SQL's *where clause* before the SQL is used. This is done in the query optimizer and is actually done when the SQL is parsed and executed. When the SQL is executed it is actually the modified SQL that is executed on behalf of the user. This means that the policy function controls which rows of data are returned. The process can be thought of as a system trigger that is executed when a table is accessed that has a policy defined. This also means that it is now possible to use this functionality to write *select triggers* against tables.

Application contexts can also be used within the policy function to define which predicate is returned to the optimizer. These application contexts are stored in a namespace for each user and can be set in name/value pairs to identify which groups of users belong to which *where* clause! It is not mandatory though to use an application context.

Next let's look at a simple example of the use of *row level security* to see how it works. First start by creating a user to use for the test and grant some basic privileges needed. The user needs to be able to create a table and add data, create package procedures, create a context and access the row level security API and session packages.

```
SQL> connect system/manager@zulia
Connected.
SQL> create user vpd identified by vpd default tablespace users temporary
tablespace temp;

User created.

SQL> grant create session to vpd;

Grant succeeded.

SQL> grant create any context to vpd;

Grant succeeded.

SQL> grant create table to vpd;

Grant succeeded.

SQL> grant unlimited tablespace to vpd;

Grant succeeded.

SQL> grant create procedure to vpd;

Grant succeeded.

SQL> connect sys/change_on_install@zulia as sysdba
Connected.
SQL> grant execute on dbms_rls to vpd;

Grant succeeded.

SQL> grant execute on dbms_session to vpd;

Grant succeeded.

SQL>
```

In general there are a number of basic actions required in setting up and use of *fine grained access control*. These basic steps could be described as follows and be applied to our example:

- *Choose the tables or views to protect at the row level:*

  In our case we will use a simple test table called *transactions* that we will create for the purpose. This table holds some financial details for a fictitious company. These financial transactions include dates, credits, debits, transaction types and cost centres.

  In the real world an application would probably want to protect tables that hold critical data or data that pertained to different divisions of an organisation or... at the design stage data that should be viewed by certain groups of users should be identified and any tables or views that store or reveal that data should be candidates for *row level security*.

The test table can be created and data can now be populated:

```
SQL> connect vpd/vpd@zulia
Connected.
SQL>
SQL> create table transactions (
  2  trndate date,
  3  credit_val number(12,2),
  4  debit_val number(12,2),
  5  trn_type varchar2(10),
  6  cost_center varchar2(10)) tablespace users;

Table created.

SQL>
SQL> insert into transactions
(trndate,credit_val,debit_val,trn_type,cost_center)
  2  values (to_date('15-OCT-2003','DD-MON-
YYYY'),100.10,0.0,'PAY','CASH');

1 row created.

SQL> insert into transactions
(trndate,credit_val,debit_val,trn_type,cost_center)
  2 values (to_date('15-OCT-2003','DD-MON-YYYY'),50.23,0.0,'PAY','CASH');

1 row created.

SQL> insert into transactions
(trndate,credit_val,debit_val,trn_type,cost_center)
  2 values (to_date('15-OCT-2003','DD-MON-
YYYY'),0.0,230.20,'INV','ACCOUNTS');

1 row created.

SQL> insert into transactions
(trndate,credit_val,debit_val,trn_type,cost_center)
  2 values (to_date('15-OCT-2003','DD-MON-
YYYY'),15.24,0.0,'INT','ACCOUNTS');

1 row created.

SQL> commit;

Commit complete.

SQL> select * from transactions;

TRNDATE   CREDIT_VAL  DEBIT_VAL TRN_TYPE   COST_CENTE
--------- ---------- ---------- ---------- ----------
15-OCT-03      100.1          0 PAY        CASH
15-OCT-03      50.23          0 PAY        CASH
15-OCT-03          0      230.2 INV        ACCOUNTS
15-OCT-03      15.24          0 INT        ACCOUNTS

SQL>
```

Now that there is a table to protect and some data, let's move on to the next stage.

- *Define the business rules that will be followed for accessing data in these tables*

  Defining the rules to be implemented is the next stage. This should revolve around which groups of people / workers should be able to access what data and how. It is possible to define differing rules on a table or view for reading data, writing data, updating data or deleting it.

  In the case of our simple example we shall define the following rules. Any user in the *accounts* section can only view all transactions in the *accounts* cost centre. Any user who is employed as a *clerk* can view only *cash* cost centre transactions. *Managers* can view all transactions and finally any user who is not any of the above three cannot view any transactions. Existing records can be added and changed in the same groups with the same rules.

- *Create a security context to manage application sessions*

  The security context is called an *application context* in Oracle and is a *namespace* that has name/value pairs. The only way to set a context is through the PL/SQL package that is bound to that context. Using a package bound to the context that has predefined rules to control setting of the context stops a malicious user from spoofing the context that she wants to gain better access.

  Creating the context is reasonably easy - it should be associated with the PL/SQL package that will be used to set values in this context for users. The package does not have to exist yet to be able to create the context. Here is the code:

```
SQL> show user
USER is "VPD"
SQL> create or replace context vpd_test using set_vpd_context;

Context created.

SQL>
```

- *Create a procedure or function to manage setting of the security context for users*

  In a real application the function used to set the application or security context would set the context for the logged in user. Remember this is the function that was *bound* to the context when it was created. This allows a trusted method to set a user's context in the sense that the context can only be set through this function so it is controlled. The context could be set at logon time with a logon database trigger or could be executed by a client application or from the middle tier on the application server or in any other number of ways.

  The setting of the context can be based on many things such as the time of day, the identity of the user who is logged on, where the user is logged in from, either the IP address (only if TCP is used) or the terminal. Here is an example of retrieving the IP Address:

```
SQL> select sys_context('userenv','ip_address') from dual;

SYS_CONTEXT('USERENV','IP_ADDRESS')
--------------------------------------------------------------------------------
------
127.0.0.1

SQL>
```

Many other application-based values could also be used such as department numbers, employee numbers, company job grade, desk location... if it is stored in the database then potentially it could be used to set the context. As discussed, in a real application we would code logic to check that the correct role had been set for a user but for our simple example we will simply provide a package that allows the setting of the application role *accountant*, *manager* or *clerk*. Here it is:

```
SQL> create or replace package set_vpd_context
  2  is
  3   procedure set_manager;
  4   procedure set_accountant;
  5   procedure set_clerk;
  6  end;
  7  /

Package created.

SQL>
```

As stated the context consists of name/value pairs. In this case we will define a variable name of *app_role* and set the relevant values in the package body:

```
SQL> create or replace package body set_vpd_context
  2  as
  3   procedure set_manager
  4   is
  5   begin
  6    dbms_session.set_context('vpd_test','app_role','manager');
  7   end;
  8   --
  9   procedure set_accountant
 10   is
 11   begin
 12    dbms_session.set_context('vpd_test','app_role','accountant');
 13   end;
 14   --
 15   procedure set_clerk
 16   is
 17   begin
 18    dbms_session.set_context('vpd_test','app_role','clerk');
 19   end;
 20  end;
 21  /

Package body created.

SQL>
```

OK, the context is now sorted, let's write the predicate function.

- *Write a package to generate the dynamic access predicates for access to each table*

This is the core functionality of the *row level security* implementation. This function is what checks the context for the current user in line with the business rules defined above and implemented in the functions to set the security context. The function then, based on the rights of the user executing the *select* statement or *update*, *insert* or *delete* returns a predicate. This predicate is a dynamic piece of SQL that is appended to the *where clause* of the executing SQL by the Oracle optimizer at the time the SQL is parsed and executed.

It is possible to write and define separate policy functions for *select*, *insert*, *update* and *delete* for each table or view. This means that differing security rules can be defined for each type of access to the data. It is also possible to have multiple policies for each object and it is possible to define policy groups for objects - we will not go to this level of detail in this paper.

Policy functions always have to have the same *signature* as they are called by the optimizer for us. They can be thought of as a *call back function*. The function has to accept two varchar parameters for the schema owner and the object name and return a varchar string. The contents of the parameters can be used in anyway by the function. The prototype should be:

```
function policy_function_name(owner in varchar2, object_name in varchar2)
               return varchar2
```

For our example we will use just one function for all four types of access for now; this is to keep the example simple. The function can now be written as follows:

```
SQL> create or replace package vpd_policy
  2  as
  3     function vpd_predicate(schema_name in varchar2, object_name in
varchar2)
  4     return varchar2;
  5  end;
SQL> /

Package created.

SQL>
```

Now create the package body for the policy:

```
SQL> create or replace package body vpd_policy
  2  as
  3     function vpd_predicate(schema_name in varchar2,object_name in
varchar2)
  4     return varchar2
  5     is
  6             lv_predicate varchar2(1000):='';
  7     begin
  8             if sys_context('vpd_test','app_role') = 'manager' then
  9                     lv_predicate:=''; -- allow all access
 10             elsif sys_context('vpd_test','app_role') = 'accountant'
then
 11                     lv_predicate:='cost_center=''ACCOUNTS''';
 12             elsif sys_context('vpd_test','app_role') = 'clerk' then
 13                     lv_predicate:='cost_center=''CASH''';
 14             else
 15                     lv_predicate:='1=2'; -- block access
 16             end if;
 17             return lv_predicate;
 18     end;
 19  end;
SQL> /

Package body created.

SQL> show errors package body vpd_policy
No errors.
SQL>
```

- *Register the policy function / package with Oracle using the DBMS_RLS package.*

  Almost all of the pieces are now in place before we can test the functionality. Next the policy function needs to be registered with the database and specifically with the table being secured. In a real application this stage is just the same as the example with one call needed to a sys owned package called DBMS_RLS. This package is the API interface to the Oracle kernel Row Level Security functionality. The package prototype can be found in `$ORACLE_HOME/rdbms/admin/dbmsrlsa.sql` where details of the API calls can be seen.

  For the example used in this paper, the same policy function will be used for all access methods to keep the example and testing reasonably simple. Here is the call to register the policy:

```
SQL> begin
  2     dbms_rls.add_policy(
  3             object_schema => 'VPD',
  4             object_name => 'TRANSACTIONS',
  5             policy_name => 'VPD_TEST_POLICY',
  6             function_schema => 'VPD',
  7             policy_function => 'VPD_POLICY.VPD_PREDICATE',
  8             statement_types => 'select, insert, update, delete',
  9             update_check => TRUE,
 10             enable => TRUE,
 11             static_policy => FALSE);
 12 end;
 13 /

PL/SQL procedure successfully completed.

SQL>
```

- *Automate the setting of the security context*

  As mentioned above, the security context can be set automatically and one useful way to do this is by using a *database logon trigger*. An example piece of code to achieve this using one of the example application context procedures would be:

```
create or replace trigger vpd_logon_trigger
after logon on database
begin
        set_vpd_context.set_accountant;
end;
/
```

Testing the example

Finally it is time to test the policies that have been set up and see if they actually restrict access as planned. First of all we will log in as the owner of the table and see if we can see the records:

```
SQL> connect vpd/vpd@zulia
Connected.
SQL> select * from transactions;

no rows selected
```

Strange, shouldn't we see four records as the user *vpd* is the owner of this table? Well actually the result is correct as currently no application role (security context) has been set for this user and the business security rules that were defined stated that when no application role is set then no data should be accessed. Next set the *clerk* role and examine what is visible:

```
SQL> exec set_vpd_context.set_clerk;

PL/SQL procedure successfully completed.
```

Use the following SQL to confirm that the application role was correctly set.

```
SQL> col namespace for a15
SQL> col attribute for a15
SQL> col value for a15
SQL> select * from session_context;

NAMESPACE       ATTRIBUTE       VALUE
--------------- --------------- ---------------
VPD_TEST        APP_ROLE        clerk
```

Now check what records can be viewed in the transactions table.

```
SQL> select * from transactions;

TRNDATE   CREDIT_VAL  DEBIT_VAL TRN_TYPE   COST_CENTE
--------- ---------- ---------- ---------- ----------
15-OCT-03      100.1          0 PAY        CASH
15-OCT-03      50.23          0 PAY        CASH

SQL>
```

Great! We are on track as we have correctly been only able to see the CASH transactions as defined for anyone working in as a *clerk*. What would happen if the context were to be set directly by using dbms_session, and not the associated package? Here is what happens:

```
SQL> exec dbms_session.set_context('vpd_test','app_role','manager');
BEGIN dbms_session.set_context('vpd_test','app_role','manager'); END;

*
ERROR at line 1:
ORA-01031: insufficient privileges
ORA-06512: at "SYS.DBMS_SESSION", line 78
ORA-06512: at line 1


SQL>
```

This shows that Oracle will only allow the context to be set using the correctly associated function - for this example *set_vpd_context*.

Next let's set the application role to *accountant* and test that only ACCOUNTS transactions can be viewed.

```
SQL> exec set_vpd_context.set_accountant;

PL/SQL procedure successfully completed.

SQL> select * from session_context;

NAMESPACE       ATTRIBUTE       VALUE
--------------- --------------- ---------------
VPD_TEST        APP_ROLE        accountant


SQL> select * from transactions;
```

```
TRNDATE   CREDIT_VAL  DEBIT_VAL TRN_TYPE   COST_CENTE
--------- ---------- ---------- ---------- ----------
15-OCT-03          0      230.2 INV        ACCOUNTS
15-OCT-03      15.24          0 INT        ACCOUNTS
```

Again success, only ACCOUNTS transactions can be viewed. This is correct behaviour for our applications security rules. Next set the application role to *manager* and check that all of the transactions can be viewed.

```
SQL> exec set_vpd_context.set_manager;

PL/SQL procedure successfully completed.

SQL> select * from session_context;

NAMESPACE       ATTRIBUTE       VALUE
--------------- --------------- ---------------
VPD_TEST        APP_ROLE        manager


SQL> select * from transactions;

TRNDATE   CREDIT_VAL  DEBIT_VAL TRN_TYPE   COST_CENTE
--------- ---------- ---------- ---------- ----------
15-OCT-03      100.1          0 PAY        CASH
15-OCT-03      50.23          0 PAY        CASH
15-OCT-03          0      230.2 INV        ACCOUNTS
15-OCT-03      15.24          0 INT        ACCOUNTS

SQL>
```

The last results show that all transactions can be seen, which is again the correct behaviour. Many more tests could be performed to show how row level security works but we do not have space to be exhaustive here. I want to show just two more tests with our simple application. The first demonstrates inserting a new record and the behaviour of one of the dbms_rls.add_policy parameters. Because we set the same policy function for all access methods the same visibility rules apply. If we set the context to *accountant* and then try and insert a CASH transaction it should fail. Let's try:

```
SQL> exec set_vpd_context.set_accountant;

PL/SQL procedure successfully completed.

SQL> select *  from session_context;

NAMESPACE       ATTRIBUTE       VALUE
--------------- --------------- ---------------
VPD_TEST        APP_ROLE        accountant

SQL> insert into
transactions(trndate,credit_val,debit_val,trn_type,cost_center)
  2  values (to_date('15-OCT-2003','DD-MON-YYYY'),120.0,0.0,'PAY','CASH');
insert into
transactions(trndate,credit_val,debit_val,trn_type,cost_center)
            *
ERROR at line 1:
ORA-28115: policy with check option violation
```

This is the correct behaviour as access for an accountant should be restricted to only inserting ACCOUNTS transactions. Let's try:

```
SQL> insert into
transactions(trndate,credit_val,debit_val,trn_type,cost_center)
  2  values (to_date('15-OCT-2003','DD-MON-
YYYY'),120.0,0.0,'INV','ACCOUNTS');

1 row created.

SQL>
```

Success! The behaviour for inserts and updates can be modified when specifying the policy functions to the kernel. The parameter update_check is defaulted to FALSE but if it is set to TRUE as we did when adding the policy function, no data can be inserted or updated that should not be viewable. If we change this parameter back to its FALSE setting and try inserting the transaction above again it should not fail!

```
SQL>  begin
  2    dbms_rls.drop_policy(
  3     object_schema => 'VPD',
  4     object_name => 'TRANSACTIONS',
  5     policy_name => 'VPD_TEST_POLICY');
  6  end;
  7  /

PL/SQL procedure successfully completed.

SQL> begin
  2      dbms_rls.add_policy(
  3              object_schema => 'VPD',
  4              object_name => 'TRANSACTIONS',
  5              policy_name => 'VPD_TEST_POLICY',
  6              function_schema => 'VPD',
  7              policy_function => 'VPD_POLICY.VPD_PREDICATE',
  8              statement_types => 'select, insert, update, delete',
  9              update_check => FALSE, -- set back to FALSE
 10              enable => TRUE,
 11              static_policy => FALSE);
 12 end;
 13 /

PL/SQL procedure successfully completed.

SQL> insert into transactions
(trndate,credit_val,debit_val,trn_type,cost_center)
  2 values (to_date('15-OCT-2003','DD-MON-YYYY'),120.0,0.0,'PAY','CASH');

1 row created.

SQL>
```

This parameter allows a *slight* circumvention of the policy rules defined! Beware of this parameter and set it cautiously or at least understand its behaviour.

One last demonstration in this test section:

```
SQL> connect system/manager@zulia as sysdba
Connected.
SQL> select * from vpd.transactions;

TRNDATE   CREDIT_VAL  DEBIT_VAL TRN_TYPE   COST_CENTE
--------- ---------- ---------- ---------- ----------
15-OCT-03      100.1          0 PAY        CASH
```

```
15-OCT-03      50.23          0 PAY        CASH
15-OCT-03          0      230.2 INV        ACCOUNTS
15-OCT-03      15.24          0 INT        ACCOUNTS
15-OCT-03        120          0 INV        ACCOUNTS
15-OCT-03        120          0 PAY        CASH

6 rows selected.
```

As you can see the SYS user or any user such as *system* connected as *sysdba* bypasses all row level security policies. Beware of access by any user as SYS or as *sysdba* if protection of data from **all users** is important.

Concluding part one

We've seen some of the advantages of Oracle's row level security, what it can be used for, and looked at a simple example of how it works. Next week in Part Two we'll conclude this short article series by testing the policies that have been setup, demonstrate a few of the data dictionary views that allow for management and monitoring, cover some other issues and features, and then see if the data can be viewed by hackers or malicious users through the use of trace files.

References

- Oracle documentation - http://tahiti.oracle.com
- Oracle 8i Virtual Private Databases - Tim Gorman - http://www.evdbt.com/VPD.pps
- Practical Oracle 8i - Building efficient databases - Jonathan Lewis - Published by Addison Wesley
- Oracle security handbook - Aaron Newman and Marlene Theriault - published by Oracle Press.
- Oracle in a nutshell - A desktop Quick reference - Rick Greenwald and David C Kreines - Published by O'Reilly
- Expert one-on-one - Thomas Kyte - Published by Wrox Press
- Internet security with Oracle Row-Level security - Roby Sherman - http://www.interealm.com/roby/technotes/8i-rls.html

About the author

Pete Finnigan is the author of the book "Oracle security step-by-step - A survival guide to Oracle security" published in January 2003 by the SANS Institute (see http://store.sans.org/). Pete Finnigan is the founder and CTO of PeteFinnigan.com Limited (http://www.petefinnigan.com/) a UK based company that specialises in auditing the security of client's Oracle databases world-wide and provides consultancy in all areas of Oracle security design, configuration and development.

View more articles by Pete Finnigan on SecurityFocus.

Oracle Row Level Security: Part 2

by Pete Finnigan
last updated November 17, 2003

Continuing from Part One

In part one of this short article series we looked at some of the advantages of Oracle's row level security, what it can be used for, and looked at a simple example of how it works. We'll conclude this series by testing the policies that have been setup, demonstrate a few of the data dictionary views that allow for management and monitoring, cover some other issues and features, and then see if the data can be viewed by hackers or malicious users through the use of trace files.

Please take a look at part one before continuing on, review the example that was previously created and you'll see where we left off.

Looking around

Now we will demonstrate a few of the data dictionary views that allow management and monitoring of policies and contexts.

The dictionary view v$vpd_policies lists all security policies and predicates for cursors that are at present in the library cache. This is an interesting view when combined with v$sqlarea so that the predicate and the "original SQL" can be seen. Here is an example query:

```
SQL> col sql_text for a25
SQL> col predicate for a20
SQL> col policy for a15
SQL> col object_name for a15
SQL> select substr(sql_text,1,25) sql_text,
  2     predicate,
  3     policy,
  4     object_name
  5  from v$sqlarea ,v$vpd_policy
  6  where hash_value = sql_hash;

SQL_TEXT                  PREDICATE            POLICY          OBJECT_NAME
------------------------- -------------------- --------------- ------------
---
select count(*) from tran 1=2                  VPD_TEST_POLICY TRANSACTIONS
select * from transaction 1=2                  VPD_TEST_POLICY TRANSACTIONS
```

Quite clearly access to this view should not be granted to any user except administrators.

There are a number of data dictionary views that can be used to manage and view policies set up in the database. As is usual with Oracle there are three levels of views - these are USER_%, ALL_% and DBA_%. The user views show only the records owned by the user, The ALL_% views show records owned by the user and any records that have had permissions granted to the user. The DBA_% views show all records in the database. The records in general can represent objects or privileges - obviously it depends on the view in question. The views of relevance are as follows:

View name	Description
%_POLICY_GROUPS	Lists groups for different policies
%_POLICY_CONTEXTS	Lists policies and associated contexts
%_POLICIES	Lists details of each policy

For instance there is a view called DBA_POLICY_GROUPS. For this article the interesting views are the %_POLICIES family. These show the actual policies implemented in the database. For example using the USER_POLICIES view for the sample user *vpd* shows:

```
SQL> col object_name for a15
SQL> col policy_name for a15
SQL> col function for a15
SQL> col sel for a3
SQL> col ins for a3
SQL> col upd for a3
SQL> col del for a3
SQL> col chk for a3
SQL> col enb for a3
SQL> select     object_name object_name,
  2     policy_name policy_name,
  3     function function,
  4     sel sel,
  5     ins ins,
  6     upd upd,
  7     del del,
  8     chk_option chk,
  9     enable enb
 10 from user_policies;

OBJECT_NAME     POLICY_NAME     FUNCTION        SEL INS UPD DEL CHK ENB
--------------- --------------- --------------- --- --- --- --- --- ---
TRANSACTIONS    VPD_TEST_POLICY VPD_PREDICATE   YES YES YES YES YES YES
```

The source code for the policy functions and any functions written to manage application contexts is available through another set of data dictionary views. The main ones are %_SOURCE views. For example to get the source for procedures owned by the current user use the USER_SOURCE view (to find a particular function add "where name={function_name}" to the SQL):

```
SQL> col name for a15
SQL> col text for a64
SQL> set pages 0
SQL> break on name
SQL> select name,text
  2  from user_source
  3  order by name,type,line;

SET_VPD_CONTEXT package set_vpd_context
                is
                 procedure set_manager;
                 procedure set_accountant;
                 procedure set_clerk;
                end;
                package body set_vpd_context
                as
                 procedure set_manager
                 is
                 begin

dbms_session.set_context('vpd_test','app_role','manager');
                {output snipped}
```

Clearly these views, particularly the ALL_SOURCE and DBA_SOURCE views can be used by hackers or less than ethical employees to gain knowledge of the security policies in the database. The dictionary view SOURCE$ owned by the SYS user can also be used to view source code held in the database. Audit existing access and do not allow access to any of these views. Use the *wrap* command to obfuscate the PL/SQL source code from prying eyes.

There are two useful methods to view the current application contexts set for a user; these are to use the public view *session_context* or to use the procedure dbms_session.list_context. We have already seen the use of the *session_context* view so we won't repeat it here; instead we will demonstrate the dbms_session method:

```
SQL> connect vpd/vpd@zulia
Connected.
SQL> exec set_vpd_context.set_manager;

PL/SQL procedure successfully completed.

SQL> set serveroutput on size 1000000
  1  declare
  2   lv_context dbms_session.appctxtabtyp;
  3   lv_size number;
  4  begin
  5   dbms_session.list_context(lv_context,lv_size);
  6   dbms_output.put_line(lv_context(1).namespace||'.'
  7    ||lv_context(1).attribute||' = '
  8    ||lv_context(1).value);
  9 end;
 10 /
VPD_TEST.APP_ROLE = manager

PL/SQL procedure successfully completed.

SQL>
```

If more than one context is stored then use a loop rather than hard coding the array index of 1.

Issues and other features

There are a few issues and gotchas with row level security that are now worth discussing. Some of these affect the way it works, and in general a user of row level security should be at least aware that some of these issues exist. I will keep this relatively brief as these issues are included here for completeness.

- *Accessing the base table from the policy function*

  One issue with row level security policy functions is that it is not possible to access the base table that the policy function is attached to from the policy function. Let us demonstrate by changing the previous policy function to this:

```
create or replace package body vpd_policy
as
   function vpd_predicate(schema_name in varchar2,object_name in varchar2)
   return varchar2
   is
           lv_predicate varchar2(1000):='';
           lv_trndate   date:=null;
   begin
           select trndate into lv_trndate
           from transactions
           where rownum=1;
           if sys_context('vpd_test','app_role') = 'manager' then
                   lv_predicate:='';
           elsif sys_context('vpd_test','app_role') = 'accountant' then
                   lv_predicate:='cost_center=''ACCOUNTS''';
           elsif sys_context('vpd_test','app_role') = 'clerk' then
                   lv_predicate:='cost_center=''CASH''';
           else
```

```
                lv_predicate:='1=2';
          end if;
          return lv_predicate;
   end;
end;
end;
/
```

Running as follows gives:

```
SQL> exec set_vpd_context.set_accountant;

PL/SQL procedure successfully completed

SQL> select * from transactions;
{hanging session}
```

Accessing the table that the policy function is attached to results in indefinite library cache pin locks - a dead lock of sorts!

The solution if access to the base table is needed is to create a view on the base table such as "create view our_view as select * from base_table". Then attach the security policy to the view and not the table.

- *Overriding row level security*

  There is a system privilege EXEMPT ACCESS POLICY that if granted allows a user to be exempt from all access policies defined on any table or view. This privilege can be useful for administering a database with row level security policies or label security if implemented. This is also a possible solution to resolve the export and import problems discussed below. First here is an example of this privilege in use:

```
SQL> sho user
USER is "VPD"
SQL> select * from transactions;

no rows selected

SQL> connect system/manager@zulia
Connected.
SQL> grant exempt access policy to vpd;

Grant succeeded.

SQL> connect vpd/vpd@zulia
Connected.
SQL> select * from transactions;

TRNDATE   CREDIT_VAL  DEBIT_VAL TRN_TYPE   COST_CENTE
--------- ---------- ---------- ---------- ----------
15-OCT-03      100.1          0 PAY        CASH
15-OCT-03      50.23          0 PAY        CASH
15-OCT-03          0      230.2 INV        ACCOUNTS
15-OCT-03      15.24          0 INT        ACCOUNTS

SQL>
```

This example shows that the user *vpd* should not be able to see any of the transactions as he has not had any application roles granted. After granting the system privilege EXEMPT ACCESS POLICY the user *vpd* can now view all transactions.

An audit should be undertaken regularly for any users who have been granted this privilege in a database that uses row level security. Also, clearly this privilege should not be granted with admin options. It should be noted that this privilege does not override any object privileges such as select, delete, insert or update on objects.

- *Issues with referential integrity*

  There are issues with referential integrity where it can be possible to update records where this should not be allowed by the security policy, by using the ON UPDATE SET NULL integrity constraint. It is also possible to delete records that again should not be allowed to be deleted by using the ON DELETE CASCADE integrity constraint. There is also a case whereby it is possible to infer the existence of records in a table without being able to access those records by using a foreign key. This can be done by attempting to insert records and watching the result. If an integrity constraint error occurs where the parent key is not found we know the record doesn't exist in the parent table. Conversely when the insert succeeds the parent record must exist. These issues exist when there is no access to a parent or child table due to row level security. More details and examples these three issues are discussed in detail in Tom Kytes' book, "Expert One-on-one Oracle".

- *Cached cursors*

  This is a rather esoteric issue. In earlier versions of Oracle with row level security (8.1.5 and 8.1.6) if the context was changed after setting it at the beginning of a session and the tool used to connect to the database caches cursors, then it is possible for the SQL affected to execute based on a previous context and not the changed one. This can mean that an incorrect predicate is used. This issue is fixed from version 8.1.7. as bind variables are used for the context. There is a second issue with cached cursors that is still a problem in later versions. If the predicate condition is retrieved dynamically from a database table for instance and that table is altered during a session and again a cursor is cached then an incorrect predicate will be used in the SQL executed. That is, the cached predicate will not be re-parsed in this case and the new predicate used. Again there is an excellent discussion of this issue in Tom Kytes' book, "Expert One-on-one Oracle"

- *Export and import*

  When export is used to export data the row level security policy rules still apply. The default mode of export is to use conventional path which means SQL is used to extract the data. Here is what happens when the transactions table is exported as the vpd user:

```
C:\oracle\admin\zulia\udump>exp vpd/vpd@zulia file=exp.dat
tables=transactions

Export: Release 9.0.1.1.1 - Production on Sun Oct 19 18:56:18 2003

(c) Copyright 2001 Oracle Corporation.  All rights reserved.


Connected to: Oracle9i Enterprise Edition Release 9.0.1.1.1 - Production
With the Partitioning option
JServer Release 9.0.1.1.1 - Production
Export done in WE8MSWIN1252 character set and AL16UTF16 NCHAR character set

About to export specified tables via Conventional Path ...
EXP-00079: Data in table "TRANSACTIONS" is protected. Conventional path may
only
 be exporting partial table.
. . exporting table                   TRANSACTIONS          0 rows exported
Export terminated successfully with warnings.
```

Export warns that it may not export all of the data requested because of row level security rules. When export is run in direct path mode the following happens:

```
About to export specified tables via Direct Path ...
EXP-00080: Data in table "TRANSACTIONS" is protected. Using conventional
mode.
EXP-00079: Data in table "TRANSACTIONS" is protected. Conventional path may
only
 be exporting partial table.
. . exporting table                    TRANSACTIONS          0 rows exported
```

Export drops back to conventional mode. The solutions to this issue if all data should be exported is to export as the user SYS or as a user using "as sysdba" so that all the expected data is exported. The issue with the *import* utility is when the update_check parameter is set to true (described above) then any rows that are attempted to be inserted that would not normally be viewable by the user running import because of the policy will be rejected. The same solution as the *export* utility applies, either import the data as SYS or as a user connected "as sysdba". In both cases the policy could also be disabled for the duration of the export or import.

All of the above issues should be noted by the Oracle security practitioner as knowledge of these could be used to an attackers benefit.

Can the SQL including predicate be extracted or traced?

One other interesting area to look into is the issue of whether the real SQL that is executed for the user can be viewed in anyway by a hacker or malicious employee. This is of course the SQL including the additional part of the *where* clause, the predicate. First we will try standard SQL trace. There are many ways to set sql trace - we will not go into them all here. Here is an example run to generate a trace file:

```
SQL> sho user
USER is "VPD"
SQL> exec set_vpd_context.set_accountant;

PL/SQL procedure successfully completed.

SQL> alter session set sql_trace=true;

Session altered.

SQL> select * from transactions;

TRNDATE   CREDIT_VAL  DEBIT_VAL TRN_TYPE   COST_CENTE
--------- ---------- ---------- ---------- ----------
15-OCT-03          0      230.2 INV        ACCOUNTS
15-OCT-03      15.24          0 INT        ACCOUNTS
15-OCT-03        120          0 INV        ACCOUNTS

SQL>
```

This generates a raw trace file in the directory pointed at by the *user_dump_dest* initialisation parameter. The format of the name is platform dependent for instance on Windows XP it is ORA{PID}.TRC for Unix it can be {SID}_ora_{PID}.trc. The PID can be found by querying the dictionary views v$process and v$session. The raw trace file can be read to see where the SQL was executed:

```
PARSING IN CURSOR #3 len=26 dep=0 uid=80 oct=3 lid=80 tim=3703688816
hv=2895734505 ad='79fff7a0'
select * from transactions
END OF STMT
PARSE
#3:c=1241785,e=2153000,p=14,cr=231,cu=6,mis=1,r=0,dep=0,og=4,tim=3703688816
=====================
PARSING IN CURSOR #4 len=764 dep=1 uid=80 oct=47 lid=80 tim=3703698816
hv=2107271493 ad='79fc88e0'
declare p varchar2(32767);                    begin  p :=
VPD_POLICY.VPD_PREDICATE(:sn, :on);                    :v1 :=
substr(p,1,4000); :v2 :=
substr(p,4001,4000);                    :v3 := substr(p,8001,4000); :v4 :=
substr(p,12001,4000);                    :v5 := substr(p,16001,4000); :v6
:=
substr(p,20001,4000);                    :v7 := substr(p,24001,4000); :v8
:=
substr(p,28001,4000);                    :v9 := substr(p,32001,767);
:v10 := substr(p, 4000, 1); :v11 := substr(p,8000,1);
:v12 :=
substr(p, 12000, 1); :v13 := substr(p,16000,1);                    :v14 :=
substr(p, 20000, 1); :v15 := substr(p,24000,1);                    :v16 :=
substr(p, 28000, 1); :v17 := substr(p,32000,1);                    end;
```

The trace shows the SQL that was typed in not the SQL including the dynamic predicate. This is not useful if we hoped to see the *real SQL* that was executed. But it is possible to see from the trace that row level security is in use as the next cursor is one that executed the policy function in a piece of dynamic PL/SQL. Even though the predicate can not be read it is possible to see that row level security is used on this table! A better way to view the trace file is to use the utility *tkprof* as follows:

```
Oracle:Jupiter> tkprof input_trace_file output_file sys=yes
```

Another, better way to view what row level security is up to is to use the event 10730. This can be done as follows:

```
SQL> sho user
USER is "VPD"
SQL> exec set_vpd_context.set_accountant;

PL/SQL procedure successfully completed.

SQL> alter session set events '10730 trace name context forever';

Session altered.

SQL> set autotrace on explain
SQL> select * from transactions;

TRNDATE   CREDIT_VAL  DEBIT_VAL TRN_TYPE   COST_CENTE
--------- ---------- ---------- ---------- ----------
15-OCT-03          0      230.2 INV        ACCOUNTS
15-OCT-03      15.24          0 INT        ACCOUNTS
15-OCT-03        120          0 INV        ACCOUNTS


Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   TABLE ACCESS (FULL) OF 'TRANSACTIONS'
```

This also generates a trace file in the same location as the normal trace again with the same naming conventions. The contents of the trace are short and show the row level security details. Here is what was generated without the header info for the above run:

```
*** 2003-10-19 16:30:59.000
*** SESSION ID:(7.64) 2003-10-19 16:30:59.000
------------------------------------------------------------
Logon user     : VPD
Table or View  : VPD.TRANSACTIONS
Policy name    : VPD_TEST_POLICY
Policy function: VPD.VPD_POLICY.VPD_PREDICATE
RLS view :
SELECT  "TRNDATE","CREDIT_VAL","DEBIT_VAL","TRN_TYPE","COST_CENTER" FROM
"VPD"."TRANSACTIONS"  "TRANSACTIONS" WHERE (cost_center='ACCOUNTS')
```

This is exactly what is needed for the curious hacker or malicious employee. It shows details of the policy function and also a modified SQL statement with the additional predicate.

One other trace method is available using event 10060 which dumps the cost-based optimizer predicates. Details of this and other events can be found on the internet. This is more complicated as it needs the use of a summary table that has to be created, statistics need to be created and the shared pool needs to be flushed to force the trace. Here is an example:

```
SQL> sho user
USER is "VPD"
SQL> analyze table transactions estimate statistics;

Table analyzed.

SQL> create table kkoipt_table(c1 int, c2 varchar2(80));

Table created.

SQL> connect system/manager@zulia
Connected.
SQL> alter system flush shared_pool;

System altered.

SQL> connect vpd/vpd@zulia
Connected.
SQL> exec set_vpd_context.set_accountant;

PL/SQL procedure successfully completed.

SQL> alter session set events '10060 trace name context forever';

Session altered.

SQL> set autotrace on explain;
SQL> select * from transactions;

TRNDATE   CREDIT_VAL  DEBIT_VAL TRN_TYPE   COST_CENTE
--------- ---------- ---------- ---------- ----------
15-OCT-03          0      230.2 INV        ACCOUNTS
15-OCT-03      15.24          0 INT        ACCOUNTS
15-OCT-03        120          0 INV        ACCOUNTS


Execution Plan
------------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=3 Bytes=63)
```

```
   1    0   TABLE ACCESS (FULL) OF 'TRANSACTIONS' (Cost=1 Card=3 Bytes
          =63)



SQL> set autotrace off
SQL> select c2 from (select distinct c1,c2 from kkoipt_table) order by c1;

C2
--------------------------------------------------------------------------------
-----
Table:
TRANSACTIONS
frofand
 "TRANSACTIONS"."COST_CENTER"='ACCOUNTS'

SQL>
```

This is a long winded way to check the predicates generated as event 10730 is easier to use but it works. From Oracle 9iR2 there are additional predicate columns on the explain plan so this becomes redundant.

The result from this query by selecting from the summary table *kkoipt_table* shows the predicate again.

The lesson from this section is that if you do not want your users, hackers or malicious employees to know row level security is used or how the SQL is modified, then do not allow them to set trace or events in anyway.

In this section we were interested from a security perspective as to what details, if any, can be viewed from the database on the operation of the optimizer in respect of row level security. Performance analysts are also interested in these trace files to know what is executing and why it might be slow.

Conclusions

Oracle's row level security implementation provides security at the actual data level that cannot be easily bypassed by use of administration tools such as *SQL*Plus* or *TOAD* or whatever else an employee or hacker may choose to use. In this respect it is excellent at securing the data.

Can the data still be hacked? The answer is probably, if an attacker is determined and skilled enough, and enough time is available. For most organisations though this is an excellent piece of functionality that should be considered for segregating data or for protecting the more secure data in a general database from a larger group of users. Use it but remember it isn't the golden global solution. The servers and database still need to be audited and hardened and an overall security policy should be in place that includes the Oracle data. Always remember the *least privilege principle* and use it to ensure that every user has only the access they need irrespective of row level security. Audit for privileges and access levels in the areas discussed above. Again the code from this paper is available at http://www.petefinnigan.com/sql.htm.

References

- Oracle documentation - http://tahiti.oracle.com
- Oracle 8i Virtual Private Databases - Tim Gorman - http://www.evdbt.com/VPD.pps
- Practical Oracle 8i - Building efficient databases - Jonathan Lewis - Published by Addison Wesley
- Oracle security handbook - Aaron Newman and Marlene Theriault - published by Oracle Press.
- Oracle in a nutshell - A desktop Quick reference - Rick Greenwald and David C Kreines - Published by O'Reilly
- Expert one-on-one - Thomas Kyte - Published by Wrox Press
- Internet security with Oracle Row-Level security - Roby Sherman - http://www.interealm.com/roby/technotes/8i-rls.html

About the author

Pete Finnigan is the author of the book "Oracle security step-by-step - A survival guide to Oracle security" published in January 2003 by the SANS Institute (see http://store.sans.org/). Pete Finnigan is the founder and CTO of PeteFinnigan.com Limited (http://www.petefinnigan.com/) a UK based company that specialises in auditing the security of client's Oracle databases world-wide and provides consultancy in all areas of Oracle security design, configuration and development.

View more articles by Pete Finnigan on SecurityFocus.