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 8.1.5. This functionality has been described as *fine grained access control* or *row level security* or *virtual pr* essentially mean the same thing. We will come back to this shortly but before we do that lets get to what t meant as an overview; a taster in fact of what *row level security* can be used for and how it can be used, w illustrate. I want to also discuss some of the issues with row level security. Finally, I also want to show how components have been implemented in the database and also touch on how to view how the actual databa 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 it really matter to us, the users of this technology? Of course not! The API package interface is called DBMS 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 *trusted Oracle* product. Label security is used by the more security minded customers of Oracle who are int protection and is an add-on for Oracle enterprise edition. A GUI tool called *policy manager* was added in 9i and allows the setting up of label security. Label security is implemented on top of VPD. For label security t and are implemented on a single column value. It is this column that represents the label. Because Oracle 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 lev auditing. This has nothing to do with fine grained access other than it works in a similar way and is manage 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 c the data to be stored in one database for different departments or even for a hosting company to store dat database. Previously this could have been done with Oracle by using either database views or database trig 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 se would be created for each group of business users. These would have hard coded *where clauses* that imple Instead, database triggers would be utilized to cater for data manipulations. Grouping business users toget 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 ir 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 databa users to be audited a whole set of authentication, application roles and audit features are generally implem

Where is all of this code? It could be in the client applications, or more recently in a middle tier application big issue! So is security - what would happen if a user connected to the database with a tool such as *SQL*/ and used one of the shared accounts? When applications control the differing business roles from manager then there is an issue. The shared database account needs to be able to see and use all of the functionality

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 security can be a solution to some of these problems. Here are a few of the advantages of row level securit

- Oracle's row level security provides a great improvement for this type of application where many use 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 throu 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 close to the actual data as possible.
- Because row level security is implemented as close to the data as possible, the loophole of accessing 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 us segregated for the purpose of hard coding views onto the data. Row level security can be made to w 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 pot
- Row level security provides a protection against ad-hoc queries as the tool does not matter anymore everyone at the source.

To close this section, it is worth noting that row level security does not make the old methods totally redun 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 each time data in the table or view is queried or altered. This function returns an additional piece of SQL ca to the original SQL's *where clause* before the SQL is used. This is done in the query optimizer and is actuall and executed. When the SQL is executed it is actually the modified SQL that is executed on behalf of the us function controls which rows of data are returned. The process can be thought of as a system trigger that i accessed that has a policy defined. This also means that it is now possible to use this functionality to write

Application contexts can also be used within the policy function to define which predicate is returned to the contexts are stored in a namespace for each user and can be set in name/value pairs to identify which grou *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 creati grant some basic privileges needed. The user needs to be able to create a table and add data, create packa 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* 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. Thi details for a fictitious company. These financial transactions include dates, credits, debits, transactio

  In the real world an application would probably want to protect tables that hold critical data or data t divisions of an organisation or... at the design stage data that should be viewed by certain groups of 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 p
  access what data and how. It is possible to define differing rules on a table or view for reading data,
  it.

  In the case of our simple example we shall define the following rules. Any user in the *accounts* sectio
  accounts cost centre. Any user who is employed as a *clerk* can view only *cash* cost centre transactio
  and finally any user who is not any of the above three cannot view any transactions. Existing record
  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/val
  is through the PL/SQL package that is bound to that context. Using a package bound to the context
  setting of the context stops a malicious user from spoofing the context that she wants to gain better

  Creating the context is reasonably easy - it should be associated with the PL/SQL package that will b
  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 contex
  is the function that was *bound* to the context when it was created. This allows a trusted method to s
  context can only be set through this function so it is controlled. The context could be set at logon tim
  could be executed by a client application or from the middle tier on the application server or in any c

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

```
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 nur it is stored in the database then potentially it could be used to set the context. As discussed, in a rea that the correct role had been set for a user but for our simple example we will simply provide a pac 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 *ap* 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 th business rules defined above and implemented in the functions to set the security context. The funct executing the *select* statement or *update*, *insert* or *delete* returns a predicate. This predicate is a dy *where clause* of the executing SQL by the Oracle optimizer at the time the SQL is parsed and execut

It is possible to write and define separate policy functions for *select*, *insert*, *update* and *delete* for ea

security rules can be defined for each type of access to the data. It is also possible to have multiple 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. T *function*. The function has to accept two varchar parameters for the schema owner and the object na 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 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 functio database and specifically with the table being secured. In a real application this stage is just the sam to a sys owned package called DBMS_RLS. This package is the API interface to the Oracle kernel Ro package prototype can be found in `$ORACLE_HOME/rdbms/admin/dbmsrlsa.sql` where details of the

For the example used in this paper, the same policy function will be used for all access methods to k 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 b
  trigger. An example piece of code to achieve this using one of the example application context proce

```
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 planne
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
application role (security context) has been set for this user and the business security rules that were defin
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 an
would happen if the context were to be set directly by using dbms_session, and not the associated package

```
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

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 s
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 te how row level security works but we do not have space to be exhaustive here. I want to show just two mor application. The first demonstrates inserting a new record and the behaviour of one of the dbms_rls.add_p the same policy function for all access methods the same visibility rules apply. If we set the context to *acc* 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 t

```
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 update_check is defaulted to FALSE but if it is set to TRUE as we did when adding the policy function, no da that should not be viewable. If we change this parameter back to its FALSE setting and try inserting the tra 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 i 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 securi 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 Next week in Part Two we'll conclude this short article series by testing the policies that have been setup, c dictionary views that allow for management and monitoring, cover some other issues and features, and the 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 b
- Expert one-on-one - Thomas Kyte - Published by Wrox Press
- Internet security with Oracle Row-Level security - Roby Sherman - http://www.interealm.com/roby,

About the author

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

View more articles by Pete Finnigan on SecurityFocus.