

## Oracle Row Level Security: Part 2

*Pete Finnigan* 2003-11-17

### 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.appctxtyp;
  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;
/

```

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.

[Privacy Statement](#)

Copyright 2006, SecurityFocus