Using VPD to secure Discoverer Reports

By Rod West, Cabot Consulting

Securing private and confidential information is vital to protect the integrity of an organisation's data. For most organisations the Data Protection Act requires that proper mechanisms are in place to protect the data. However, using flexible ad-hoc reporting tools such as Discoverer poses potential security problems because when you implement the security you often don't know what data the user will be accessing.

There is always a tension between keeping flexibility in the reporting tool, implementing security and maintaining the performance of the reports. The report writing investment is maximised by having flexible reports run by many different users where each user is restricted to the data they are allowed to see. Performance is often an issue with reporting tools and implementing security can adversely affect the performance of system. If implemented ineffectively you could be forgiven for thinking VPD stands for Vast Performance Drain.

This article uses a series of examples to demonstrate how VPD can be used to secure your reports. These examples have been written with Discoverer in mind but VPD is a database centric approach and so can be applied to any reporting tool. The database centric approach makes using VPD with Discoverer attractive because the data will be secured in the database independent of Discoverer. You can let users develop new reports and extend the Discoverer EUL (End User Layer) knowing that the underlying data is secure. The flexibility of VPD complements the flexibility of Discoverer.

So what is VPD and how does it work?

VPD (Virtual Private Database) was introduced in Oracle 8i and the functionality further extended in Oracle 10g. It is sometimes referred to in documentation as FGAC (Fine Grained Access Control) or RLS (Row Level Security). VPD provides a flexible mechanism for securing data in the database by preventing users from retrieving specific rows and columns.

For example, you may need to let some users see data only about their department or you may need to hide sensitive financial information about some employees from all but a few users. These and far more complex security requirements can all be implemented using VPD policies.

To define a VPD policy, a PL/SQL function needs to be created that returns the condition that you want to use to secure the data. This is known as the policy function, and the condition returned is the policy predicate. The policy function can return different predicates for different groups of users and database objects. The add policy function in the DBMS_RLS package can be used to apply the policy to one or more database tables, views or synonyms.

It is often useful to think of VPD as creating views 'on the fly'. These transient views can be specific to the user and secure the underlying data.

Multiple policies can be applied to the same database objects, each policy dealing with a particular security requirement. The policy predicates are then ANDed together so that all the security rules are applied.

VPD can be used to control inserts, updates and deletes but this article is about reporting and so only considers selects. VPD policies can also be grouped with an application but again policy grouping is beyond the scope of this document.

Example 1 - A simple example

This first example shows how to secure tables and views in a payroll system where you want to allow employees to access only their own records which are keyed by person_id. There is a view defined (USER_PERSON_V) that can be used to link the database user to the person_id. The VPD policy is implemented by creating a policy function that returns the predicate that you need to add to the query when the user queries the table or view.

The policy function can be created and added to a table using the SQL*Plus commands below.

```
CREATE OR REPLACE FUNCTION person_policy
 (p_object_schema VARCHAR2, p_object_name VARCHAR2) RETURN VARCHAR2
AS
v_person_id user_person_v.person_id%TYPE;
BEGIN
   SELECT person_id INTO v_person_id
   FROM user_person_v WHERE username = USER;
   RETURN 'person_id = '||v_person_id;
END;
/
exec DBMS_RLS.ADD_POLICY ('DISCO', 'EMP', 'PERSON_POLICY', 'DISCO', 'PERSON_POLICY')
```

This is a very simple example to demonstrate VPD but is also an example of how **NOT** to define VPD policies. There are several problems with this policy which could result in performance issues:

- **No Bind Variables:** The policy does not use bind variables and each person id is hardcoded into the query. Each access to the object by a different user will result in a new SQL statement. Each statement will be added to the SQL cache in the database filling up the cache with copies of the query with different person_ids.
- **Dynamic:** By default the policy is dynamic which means that the policy function is run every time the object is accessed.
- **No Error Handling:** Finally, if there is no person_id for the user the policy function will fail causing an “ORA-28112: failed to execute policy function” error when this user selects from the database object.

Example 2 – Application contexts

This example is similar to the previous in that the tables and views are secured by person_id but this time an application context is used to store the person_id. Application contexts help you define efficient VPD policies by providing a mechanism for storing parameters used in the policy predicate. The policy predicate and the resulting SQL can then be used by many users reducing the space used in your SQL cache.

The person id must be stored in the application context at the start of the session and this can be accomplished using a Discoverer or database logon trigger. The context will be available for the duration of the session and can be used to bind the person_id into the query. Now the predicate returned by the policy will be the same for every user and therefore the policy can be defined as static. The policy function and logon trigger are shown below.

```
CREATE OR REPLACE CONTEXT VPD_CONTEXT USING disco_pkg
/
CREATE OR REPLACE PACKAGE disco_pkg
AS
PROCEDURE initialize_person_policy;
FUNCTION person_policy (p_object_schema VARCHAR2, p_object_name VARCHAR2) RETURN
VARCHAR2;
END disco_pkg;
/
CREATE OR REPLACE PACKAGE BODY disco_pkg
AS
PROCEDURE initialize_person_policy (p_user VARCHAR2 DEFAULT USER)
IS
v_person_id user_person_v.person_id%TYPE;
BEGIN
   SELECT person_id into v_person_id
   FROM user_person_v WHERE username = USER;
   DBMS_SESSION.set_context('VPD_CONTEXT', 'PERSON_ID', v_person_id);
EXCEPTION WHEN NO_DATA_FOUND THEN RETURN;
END initialize_person_policy;
FUNCTION person_policy (p_object_schema VARCHAR2, p_object_name VARCHAR2) RETURN
VARCHAR2
IS
BEGIN
   RETURN 'person_id = SYS_CONTEXT(''VPD_CONTEXT'', ''PERSON_ID'')';
END person_policy;
END disco_pkg;
/
CREATE OR REPLACE TRIGGER person_policy_trigger
AFTER LOGON ON DATABASE
CALL disco_pkg.initialize_person_policy
/
```

The predicate returned by this policy function can be used to secure any database view or table with a person_id column. Therefore this policy can be defined as a shared static policy because the policy predicate is the same for all views and tables. The shared static policy type is only available in Oracle 10g.

This type of policy is equivalent to defining a static view over the underlying database object, but without the maintenance overhead of the additional view. The policy function is called just once when any of the database objects is first accessed. This type of policy has the minimal overhead.

The policy can then be applied to tables and views using the DBMS_RLS.add_policy function. Tables and views used in a Discoverer reporting environment can be found by examining the Discoverer End User Layer (EUL). The EUL table EUL5_OBJS can be used to list the tables and views used in your EUL. Also, the DBA_TAB_COLUMNS view can be used to check whether the table or view has the required person_id column. This enables you, for example, to use the SQL below to search the EUL and quickly add the policy to all eligible tables and views that are referenced by EUL base folders.

```
BEGIN
FOR c IN (SELECT c.owner, c.table_name
          FROM disco.eul5_objs o
             , dba_tab_columns c
          WHERE obj_type = 'SOBJ'
          AND c.column_name = 'PERSON_ID'
          AND c.table_name = o.sobj_ext_table
          AND c.owner = o.obj_ext_owner)
LOOP
   DBMS_RLS.ADD_POLICY (
      object_schema    => c.owner,
      object_name      => c.table_name,
      policy_name      => 'person_policy',
      function_schema  => 'DISCO',
      policy_function  => 'DISCO_PKG.PERSON_POLICY',
      statement_types  => 'SELECT',
      policy_type      => DBMS_RLS.SHARED_STATIC);
END LOOP;
END;
/
```

Example 3 – Securing Columns for different users

In the previous examples VPD prevents a user from retrieving rows. In this example, VPD is used to hide payment details from all users except managers. Non-managers will be able to see the employee details but the sort code and account number columns will be null. This policy can be applied 'on top of' a row policy so that both rows and columns are secured.

This policy again uses an application context set at the start of the session to store whether the user is a manager. This allows the policy to be defined as a static policy because the policy predicate is the same for all users.

It is often useful to use views to store the queries in policy functions. This allows the query to be checked and maintained separately from the policy function. In this example, an IS_MANAGER_V view is used to check when the user has been assigned the PAYMENT_MANAGER database role. Note that the USER_ROLE_PRIV view cannot be used because the policy function is run with the rights of the policy function owner.

```
CREATE OR REPLACE VIEW is_manager_v AS
SELECT 'Y' Y FROM dba_role_privs
WHERE granted_role = 'PAYMENT_MANAGER' AND grantee = USER
/
```

The initialization and policy functions are shown below. For non-managers the context is not set so that the policy predicate returns false.

```
PROCEDURE initialize_payment_policy
IS
v_dummy VARCHAR2(1);
BEGIN
   SELECT Y INTO v_dummy FROM is_manager_v;
   DBMS_SESSION.set_context('VPD_CONTEXT', 'IS_MANAGER', v_dummy);
   EXCEPTION WHEN NO_DATA_FOUND THEN RETURN;
END initialize_payment_policy;
FUNCTION payment_policy (p_object_schema VARCHAR2, p_object_name VARCHAR2) RETURN
VARCHAR2
IS
BEGIN
   RETURN 'SYS_CONTEXT(''VPD_CONTEXT'', ''IS_MANAGER'')= ''Y''';
END payment_policy;
```

The policy can be applied using the SQL below.

```
BEGIN
DBMS_RLS.ADD_POLICY (
   object_schema         => 'DISCO',
   object_name           => 'EMP_PAYMENT_METHODS',
   policy_name           => 'PAYMENT_POLICY',
   function_schema       => 'DISCO',
   policy_function       => 'DISCO_PKG.PAYMENT_POLICY',
   statement_types       => 'SELECT',
   policy_type           => DBMS_RLS.STATIC,
   sec_relevant_cols     => 'SORT_CODE,ACCOUNT_NUMBER',
   sec_relevant_cols_opt => DBMS_RLS.ALL_ROWS );
END;
/
```

However, there are a number of restrictions that apply to column VPD policies:

- Securing columns is only available in Oracle 10g
- The policy predicate must be a simple expression
- The policy cannot be applied to a synonym.

Example 4 – Using Global Temporary Tables

Typically the security requirements in a reporting environment are more complex than in the previous examples with groups of users at different levels of the organization having different levels of security applied.

In this example, in an HR system pay teams are responsibility for a group of employees where a pay team can only see details of the employees in their team. Managers can see the payment details for all employees.

In this policy the predicate needs to select from a list of employees. There are a number ways that this can be achieved:

- If the number of people in the team is small then multiple contexts can be used:
  `person_id IN (SYS_CONTEXT('VPD_CONTEXT', 'PERSON_ID_1'), SYS_CONTEXT('VPD_CONTEXT', 'PERSON_ID_2'), SYS_CONTEXT('VPD_CONTEXT', 'PERSON_ID_3'))`
- A subquery can also be used:
  `IN (SELECT person_id FROM PERSON_USER_V WHERE USERNAME = USER)`
- However the most efficient policy may be to load a global temporary table (GTT) with the list of person ids for the user as shown in the example below.

In this example, the policy has different predicates for different users and therefore the policy type cannot be static; it has different predicates for different objects so cannot be shared. Therefore the policy should be defined as context sensitive. This is not available in Oracle 9i where a dynamic policy must be used.

A context sensitive policy function is run when the database object is first used in the session. Also the policy package is loaded when any policy database object is first used in the session. Therefore the package initialisation rather than a logon trigger can be used to load the GTT.

```
CREATE GLOBAL TEMPORARY TABLE team_person_gtt (person_id NUMBER
, CONSTRAINT team_person_gtt_pk PRIMARY KEY (person_id) USING INDEX)
ON COMMIT PRESERVE ROWS
/
CREATE OR REPLACE PACKAGE disco_pkg
AS
FUNCTION team_policy (p_object_schema VARCHAR2, p_object_name VARCHAR2) RETURN
VARCHAR2;
END disco_pkg;
/
CREATE OR REPLACE PACKAGE BODY disco_pkg
AS
FUNCTION team_policy (p_object_schema VARCHAR2, p_object_name VARCHAR2) RETURN
VARCHAR2
IS
v_dummy   VARCHAR2(1);
BEGIN
   SELECT Y INTO v_dummy FROM is_manager_v;
   RETURN NULL; -- do not apply policy
EXCEPTION WHEN NO_DATA_FOUND THEN
   RETURN 'EXISTS (SELECT NULL FROM team_person_gtt WHERE person_id='
          ||p_object_name||'.person_id)';
END team_policy;
BEGIN -- package initialization
INSERT INTO team_person_gtt
SELECT person_id FROM user_person_v
WHERE username = USER;
COMMIT;
END disco_pkg;
/
BEGIN
DBMS_RLS.ADD_POLICY (
   object_schema            => 'DISCO',
   object_name              => 'EMP',
   policy_name              => 'TEAM_POLICY',
   function_schema          => 'DISCO',
   policy_function          => 'DISCO_PKG.TEAM_POLICY',
   statement_types          => 'SELECT',
   policy_type              => DBMS_RLS.CONTEXT_SENSITIVE);
END;
/
```

A useful feature is that user does not need privileges to insert or select from the GTT because the policy function and predicate use the rights of the owner of the policy.

Example 5 – Temporarily removing access

VPD is normally used to secure data in the database but it can also be useful for restricting access to the database for other reasons. In this example there are a set of tables that are loaded through an interface and must be manually validated before users can access the data. VPD can be used to raise an application error when the user tries to access a table while the table is temporarily unavailable.

In this example there is a view (TABLE_MESSAGE_V) that returns the application message when the table is unavailable. The policy returns a predicate that containing a call to raise an application message. If the view returns a null message then the table is available and policy returns a null predicate.

```
CREATE OR REPLACE PACKAGE disco_pkg
AS
FUNCTION raise_exception (p_message VARCHAR2) RETURN VARCHAR2;
FUNCTION table_policy (p_object_schema VARCHAR2, p_object_name VARCHAR2) RETURN
VARCHAR2;
END disco_pkg;
/
CREATE OR REPLACE PACKAGE BODY disco_pkg
AS
FUNCTION raise_exception (p_message VARCHAR2) RETURN VARCHAR2
IS
BEGIN
   RAISE_APPLICATION_ERROR(-20001, p_message);
END raise_exception;
FUNCTION table_policy (p_object_schema VARCHAR2, p_object_name VARCHAR2) RETURN
VARCHAR2
IS
v_message   table_messages_v.message%TYPE;
BEGIN -- if table owner return null predicate
   IF p_object_schema = USER THEN RETURN NULL; END IF;
   SELECT message INTO v_message FROM table_messages_v
   WHERE table_name = p_object_name AND message IS NOT NULL;
   RETURN 'disco_pkg.raise_exception('''||v_message||''') IS NULL';
EXCEPTION WHEN NO_DATA_FOUND THEN RETURN NULL;
END table_policy;
END disco_pkg;
/
BEGIN -- apply policy to all tables
FOR c IN (SELECT table_name FROM table_messages_v)
LOOP
DBMS_RLS.ADD_POLICY (
   object_schema         => 'DISCO',
   object_name           => c.table_name,
   policy_name           => 'TABLE_POLICY',
   function_schema       => 'DISCO',
   policy_function       => 'DISCO_PKG.TABLE_POLICY',
   statement_types       => 'SELECT',
   policy_type           => DBMS_RLS.DYNAMIC);
END LOOP;
END;
/
```

When the Discoverer user runs a report that accesses a table which is unavailable the database raises an exception showing the application message set for this table. As this is a dynamic VPD policy when the message is removed the Discoverer user can just refresh their report to retrieve data from the table.

Example 6 – Securing Summary and Detail information

A requirement frequently requested is that users can see summary information, for example, totals of employees, but only certain users can drill down to the detail data.

For example all users can see summary information but only managers can drill down to the detail.

This requirement can be implemented in a Discoverer environment using summary management. A detail view and a summary view need to be created over the detailed data and the Discoverer report created using the detail view. The VPD policy should be applied only to the detail view so that the user can still see the summary data. An external summary folder based on items in the detail folder can be created in Discoverer that maps the items to the summary view.

Then when the user runs the report at the summary level, Discoverer redirects the workbook to the summary folder and the user will see the summarised data. You can achieve a similar effect in the database using Materialized Views and Materialized View query rewrite.

Testing and debugging your VPD policies

Testing and debugging VPD policies can be a challenge especially where there are multiple, complex policies applied to different sets of users. An organisation's password policy often prevents you from logging into the database as another user to check their security. Also as a Discoverer user you will not be able to freely enter SQL to diagnose problems and therefore it is vital to implement sufficient instrumentation so that you have some tools that you can use to test your VPD.

A big advantage of using application contexts to control the policies is that you can let privileged users set the contexts and cause VPD to behave as though they were logged on as a different user.

In a Discoverer environment, you can interact with the VPD environment by creating a workbook that allows a privileged user to enter as a parameter the username of another user. The workbook uses a calculation to call a function (an example is shown below) that is mapped into the EUL and sets the application contexts for the VPD. Once the workbook has been run further Discoverer reports can be run to check the security is correctly applied.

```
FUNCTION set_user (p_username VARCHAR2) RETURN VARCHAR2
IS
v_person_id   user_person_v.person_id%TYPE;
BEGIN
  -- check privileged user
   IF USER <> 'DISCO_ADMIN' THEN RETURN 'FALSE'; END IF;
   initialize_person_policy(p_username);
   RETURN 'TRUE';
EXCEPTION WHEN NO_DATA_FOUND THEN RETURN 'FALSE';
END set_user;
```

You should also consider creating a diagnostic view over V$VPD_POLICY to show the current predicates for the policy and mapping the view in the EUL.

Using VPD with the e-Business Suite

Great care must be exercised when adding VPD policies to an e-Business Suite environment because the policy will be applied to all processes accessing the database object and therefore the VPD policy may have unforeseen side effects on Applications 11i processes.

However it is technically possible and in fact VPD is used internally within the e-Business Suite in a number of areas. If you are just using VPD for securing reports then many of the technical complexities can be avoided.

Familiarity with the applications security is necessary to set up an effective VPD policy. The current user and responsibility can be obtained by calling the FND_GLOBAL.user_id and FND_GLOBAL.resp_id functions.

An in-depth discussion of VPD with the e-Business Suite is beyond this article but following the guidelines below will help avoid most of the pitfalls:

- Only apply the policy to tables and views used in the reporting environment ideally only to custom views created for reporting. Policies applied to custom views do not affect Applications 11i processes.
- Use application contexts to bind variables in the predicates and use NVL to ensure that Applications 11i processes are not affected if the context is not set
  `NVL(SYS_CONTEXT('VPD_CONTEXT', 'PERSON_ID'), person_id) = person_id`
- Implement policies that only return a predicate if the session is a Discoverer session. In Oracle 10g a dynamic or context-sensitive policy function can use the statement

below to return a null predicate if the session is not a Discoverer session ensuring that Application 11i processes are not affected.

```
IF SYS_CONTEXT('USERENV', 'MODULE') NOT LIKE 'Disco%' THEN RETURN NULL; END IF;
```

- Use the "Initialization SQL Statement Custom" system profile attached to a reporting responsibility or a Discoverer logon trigger rather than a database logon trigger to set the policy contexts. For example, enter a call to initialise the policy into the system profile as shown below

```
BEGIN disco_pkg.initialize_person_policy; END;
```

Conclusion

Oracle VPD is a versatile tool and can be used to implement both simple and complex security requirements. Discoverer security is often implemented by building mandatory conditions into views or the EUL, but using VPD has many advantages over this approach.

Application independent: VPD is a database centric approach and so can be used by any reporting tools. Users can develop complex Discoverer ad-hoc reports with confidence that the security policies are being applied. Discoverer Administrators can extend the EUL and automatically inherit the security rules from the underlying database tables and views.

Better performance: VPD provides the best mechanism for efficiently implementing security. Many other methods include a condition calling a validate_access function that checks whether the user can access the row. This involves for each row processed a switch to PL/SQL to check whether the user can access the row. When VPD is used with application contexts, the predicates are usually defined at the start of the session. Therefore the business logic is processed once and the query in the reports can be processed without the use of any PL/SQL.

More flexible implementation: Each VPD policy can be directly related to a business security requirement and can be applied to all the database objects that need to be secured. The database will apply all the security rules if multiple policies apply to a single database object. There is no need to create additional database objects and grant privilege to implement security.

Easier to maintain: Views and tables can be added and removed easily from the VPD policies. New tables and views can be automatically added to the policy when they are created using a database trigger. Any change to the security requirement can be made by adjusting the related VPD policy, often by just changing an underlying view that defines the user data relationship. No changes are required to the reports or the Discoverer EUL and if you have several hundred reports and folders in your EUL then this really is a big advantage.

About the Author

Rod West has been using Oracle databases since 1985 and is principal consultant at Cabot Consulting. He specialises in Oracle Applications 11i and Discoverer. Rod can be contacted at rodwest@cabotconsulting.co.uk.

