EXAM OBJECTIVE: APPLY A COLUMN LEVEL

In earlier versions of Oracle (9i), VPD was attached to the notion of a user accessing an object. In Oracle 9i, VPD now includes a notion of security-relevant columns. Only when certain relevant columns are accessed will the database append the VPD predicate to queries.

By security relevant columns we are referring to columns that hold sensitive data such as social security numbers, credit card numbers, salaries etc. If you do not specify certain relevant columns, VPD will rewrite all statements accessing the objects to contain the appropriate predicate. This enforcement of column-level privacy occurs whenever a command references the column either explicitly (column name specified as part of column list) or implicitly (when an asterisk * is used to refer to all columns).

Consider an EMPLOYEE_DATA table containing columns (EMPLOYEE_NO, EMPLOYEE_NAME, DEPARTMENT_NO, SALARY, MANAGER_NO).

Let us say that of the columns listed above the SALARY column could be considered a security-relevant column.

Consider a security policy "A manager can only access his/her subordinates' SALARY details"

The example below displays how you would associate a policy called "MANAGER_POLICY", with a table called EMPLOYEE_DATA belonging to the HR schema. The function that implements this policy is MANAGER_SECURITY. The policy is implemented only when a SELECT, INSERT or UPDATE is done on the security-relevant SALARY column.

```
BEGIN
DBMS_RLS.ADD_POLICY
(object_schema => 'hr',
 Object_name => 'EMPLOYEE_DATA',
 Policy_name => 'MANAGER_POLICY',
 Function_schema => 'HR',
 Policy_funciton -> 'MANAGER_SECURITY',
 Statement_types => 'SELECT, INSERT, UPDATE'),
 Sec_relevant_cols => 'SALARY');
END;
/
```

In this case fine grained access control will not be enforced when a statement of the following is issued.

SELECT EMPLOYEE_NO, EMPLOYEE_NAME
FROM EMPLOYEE;

Fine grained access control will be enforced when the statement issued are:

1) SELECT * FROM EMPLOYEE;
2) SELECT EMPLOYEE_NO, SALARY FROM EMPLOYEES;
3) UPDATE EMPLOYEE
   SET SALARY=SALARY+100
   WHERE EMPLOYEE_NO=999;

In the above cases 1)2)3) the queries will be rewritten after appending a predicate such as "WHERE MANAGER_NO=n;"   (*where n is the manager number of the individual accessing the records)*

EXAM OBJECTIVE: APPLY STATIC AND NON-STATIC POLICIES

To better understand static and non-static policies consider the following examples:

1) VPD can limit access to data in certain tables for all users, as required by a corporate policy.
- A company involved in trading might need to limit access to certain tables so they are accessible only during trading hours.
- The organization can implement a simple VPD policy that acquires the time or data and the day of week from the system's SYSDATE.

2) Consider an HR clerk who is allowed only to see employee records in the Engineering Division.

- When the user initiates the query "SELECT * FROM EMPLOYEE_DATA" the function implementing the security policy returns the predicate "division=MARKETING" and the database transparently rewrites the query so that the query actually executed becomes "SELECT *FROM EMP WHERE division =MARKETING".

STATIC POLICIES

The first example was an example of what is called as a STATIC policy. STATIC POLICIES: VPD always enforces the same predicate for access control, regardless of the user accessing the object. The predicate is cached in the SGA.

- Can be of two kinds: STATIC or SHARED_STATIC
- STATIC – is when the same policy is applied on a per object basis.
- SHARED_STATIC – is when the same policy is shared across multiple objects.
- To declare a static policy you use the DBMS_RLS.ADD_POLICY procedure. The **POLICY_TYPE** argument can take values of either STATIC or SHARED_STATIC, depending on whether you want to apply the predicate to a single object or to multiple objects.

NON-STATIC POLICIES

The second example is an example of what is called as a NON-STATIC policy. These policies change dynamically based on the user who may be accessing the object.

NON-STATIC POLICIES: are also known as Context-sensitive policies. This is useful when you need to apply different VPD predicates, depending on the user accessing the data. For example, if a HR personnel was accessing the data in an ORDERS table, the person should have a predicate of "WHERE employee_no =" and if a CUSTOMER was viewing his record the predicate should be "WHERE customer_no="

Non-static policies can be of three kinds:

- DYNAMIC
- CONTEXT_SENSITIVE
- SHARED_CONTEXT_SENSITIVE.

DYNAMIC policies execute the policy function every time a command accessing the security relevant columns of the object. It is useful when you have to base your VPD policy on changes in the system. THE VPD uses information in the application context for determining the appropriate predicate. The database does not cache the predicate in the SGA. Dynamic policies are also applied on a per object basis.

In CONTEXT_SENSITIVE policies, the policy function is re-executed whenever the session context changes. It is applied on a per object basis.

SHARED_CONTEXT_SENSITIVE policies execute the policy function whenever the session context changes. They can be shared across multiple objects.

To declare a static policy you use the DBMS_RLS.ADD_POLICY procedure. The **POLICY_TYPE** argument can take values of either DYNAMIC, CONTEXT_SENSITIVE or SHARED_CONTEXT_SENSITIVE, depending on whether you want to apply the predicate to a single object or to multiple objects.

EXAM OBJECTIVE: SHARE VPD POLICY FUNCTIONS

Using this feature you can apply a single VPD policy static and non-static to multiple objects.

Users should be able to see only the data that is relevant to their area of business on several objects using a single policy. This is a preferred method since we do not have to repeatedly create one policy per object.

A static policy that can be shared across multiple objects can be defined by setting the POLICY_TYPE to SHARED_STATIC when specifying the DBMS_RLS.ADD_POLICY procedure. Consider the example below, the MANAGER_POLICY can be applied to other tables and views, since you have specified the POLICY_TYPE to SHARED_STATIC, the same policy can be applied to another table or view.

A non-static policy that can be shared across multiple objects can be defined by setting POLICY_TYPE to SHARED_CONTEXT_SENSITIVE when specifying the DBMS_RLS.ADD_POLICY procedure.

```
BEGIN
DBMS_RLS.ADD_POLICY
(object_schema => 'hr',
 Object_name => 'EMPLOYEES',
 Policy_name => 'MANAGER_POLICY',
 Function_schema => 'HR',
 Policy_function -> 'MANAGER_SECURITY',
 Statement_types => 'SELECT, INSERT, UPDATE'),
 Policy_type => DBMS_RLS.SHARED_STATIC,
 Sec_relevant_cols => 'SALARY');
END;
/
```

EXAM OBJECTIVE: UNIFORM AUDIT TRAILS

OVERVIEW OF AUDITING

Auditing is the monitoring of database actions performed by certain users and may also be used to investigate suspicious database activity or even gather information about specific database activities, such as which tables are being updated, logical I/O are being performed, concurrent users connections as peak times. Auditing can be done by session or access.

UNIFORM AUDIT TRAILS

In Oracle 10g, the columns referenced in the DBA_AUDIT_TRAIL and DBA_FGA_AUDIT_TRAIL have been standardized. There are some new attributes in the two tables that complement each other.

While performing fine-grained auditing, certain new columns added to DBA_FGA_AUDIT_TRAIL include

1) A serial number for each audit record.
2) A statement number that links the multiple audit entries that generate from a statement.
3) An action code, which is a numeric code for each type of action. The SELECT statement has an action code of 3.
4) An object identifier, which is the OBJECT ID given to every object in the database.

EXAM OBJECTIVE: USING FINE-GRAINED AUDITING IN SELECT AND DML STATEMENTS

In Oracle 9i, fine-grained auditing leant its support only to SELECT statements. The Oracle 10g database extends this support to include INSERT, UPDATE and DELETE statements as well. A DBA can be called or paged when the database determines a violation of a fine-grained audit policy. This can be done by an EVENT HANDLER.

Oracle 10g audits DML statements when the manipulated rows meet the audit policy. If the audit policy specifies a certain column, an audit record will be generated only when a statement references that column.

Consider the example given below used to enforce fine-grained auditing on an object EMPLOYEES belonging to the HR schema. The name of the policy is HR_POLICY. It does not have an audit condition (NULL) specified indicating all statements that access the object are audited. The specific security-relevant column is SSNO. A procedure called NOTICY_US created in the ALL_DBAS schema gets executed when a user issues an INSERT or UPDATE on the SSNO column. Note that DELETE statements are always audited, since the entire row is referenced. In case you omit the HANDLER_SCHEMA argument it would default to the schema of the Object.

```
BEGIN
      DBMS_FGA.ADD_POLICY
      (object_schema => 'HR',
       object_name => 'EMPLOYEES',
       policy_name => 'HR_POLICY',
       audit_condition => NULL,
       audit_column => 'SSNO',
       handler_schema => 'ALL_DBAS',
       handler_module => 'NOTIFY_US',
       enable => TRUE
       statement_types => 'INSERT, UPDATE');
END;
/
```

An audit record will be generated if a user issues a statement like:

```
UPDATE EMPLOYEES
SET SSNO='XXXXX'
WHERE EMPLOYEE_ID=10;
```

Or
```
INSERT INTO EMPLOYEES(SSNO)
VALUES('YYYYYY');
```

Or
```
DELECT FROM EMPLOYEES
WHERE EMPLOYEE_NO=11;
```