Fine-Grained Access Control

Fine Grained Access Control

- **Fine-grained access control** examples:
  - Students can see their own grades
  - Students can see grades of all students in courses they registered for
    - Variant: but not the associated student-ids
  - Public can see average grades for all courses
  - Faculty can see/update/insert/delete grades of courses they taught
- SQL does not support such authorization
  - SQL authorization at the level of table/column
    - not row level

Fine-Grained Access Control

- Usual solution: handled by application programs
- Application-layer access control limitations
  - Complex, redundant code
  - Malicious/careless programmers
  - SQL injection problems
  - Application code runs in “super-user” mode always
  - Repeated security logic
  - Can be bypassed
- Solution: access control inside database

Access Control Using Views

- Common solution: Views

v

```
create view ShawnGrades as
select * from Grades where student_id = 'Shawn'
```

q

```
select  grade from ShawnGrades
where course = 'CS262B'
```

- Per-user views – difficult to administer
- Solution: **parametrized views**
  - **create view** MyGrades **as**
    select * **from** Grades **where** student_id = $userid
- Authorization-conscious querying
  - Instead of grades, must use MyGrades for students, another view for faculty, etc,

Authorization-Transparent Querying

- View-level data independence
- Analogous to physical/logical data independence
  - Changes to underlying authorization should not directly affect queries
- Query base relations rather than views
  - Query rewritten internally
  - Minimal query processing overheads
- Easy to build applications
  - Views can be user-specific, for multi-user apps
  - Generated queries better not be user-specific

The View Replacement Approach

- AKA: **Filter model (Using query rewriting mechanisms)**
- Transparent query modification

q

```
select * from Grades
```

"Grades of all students"

↓

q_m

```
select * from Grades
where studeint_id = 'Shawn'
```

"Grades of current user (Shawn)"

- Used in Oracle's Virtual Private Database

Drawbacks of View Replacement

- May provide misleading information
  - Query executes in an artificial world
  - Inconsistencies between the answer and user's external information
  - Even if query is actually authorized!

q

select avg(grade) **from** Grades

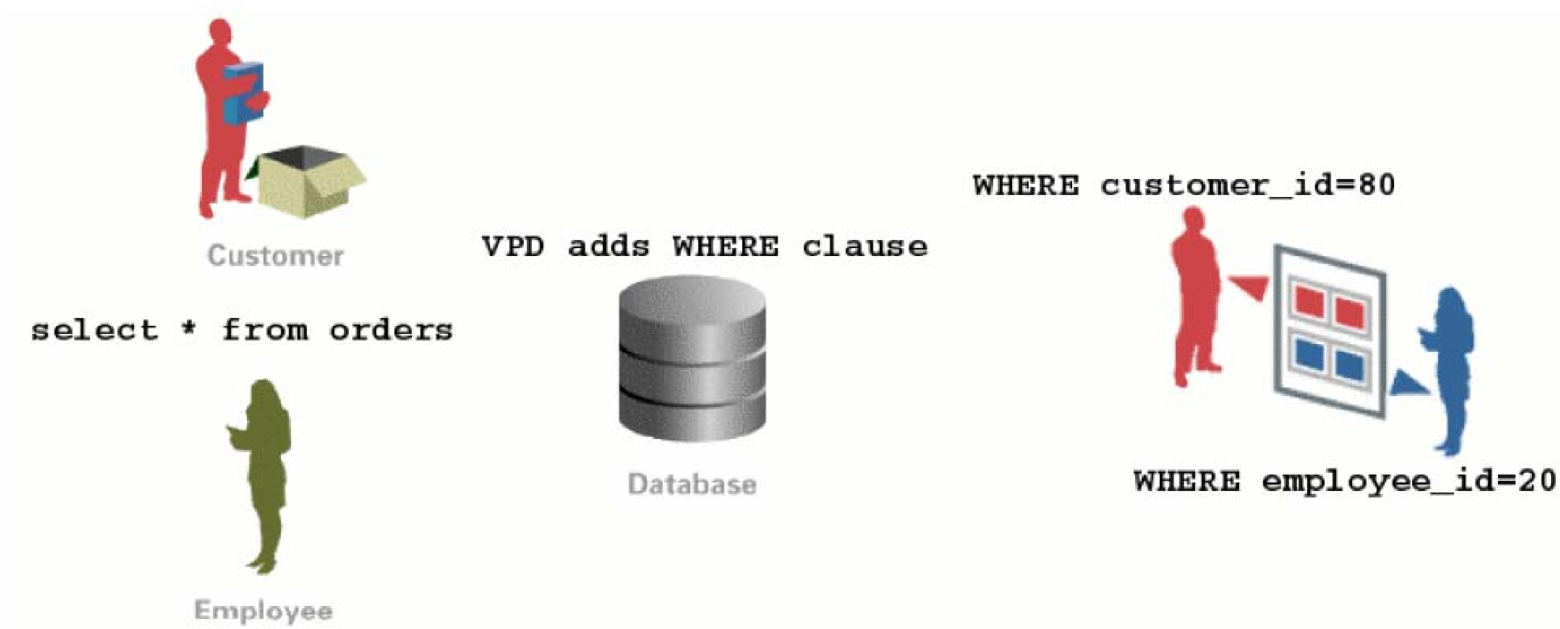
"Average grade across all courses and across all students"

q_m

select avg(grade) **from** Grades
where student_id = 'Shawn'

"Average grade across all courses for the current user"

Virtual Private Databases



Oracle VPD

- Sometimes referred to as Oracle Row-Level Security (RLS) or Fine Grained Access Control (FGAC)
- **FGAC**: associate security policies to database object
  - Predicates transparently added to query/update ***where*** clause for **each** relation used in query/update
  - User-defined functions (specified by application) generate the predicates
    - Functions encode security logic, can be in C/Java
    - *Secure application context* stores session parameters, which can be accessed by function and used in access control, e.g., for implementing temporal access control
- **Application Context**
  - Database user information is insufficient, need to know application user
  - Oracle provides mechanism for application to inform DB about end user
- Combining these two features, VPD enables administrators to define and enforce row-level access control policies based on session attributes

Oracle VPD (Cont.)

- Example applications
  - Application service providers (hosted applications)
    - E.g predicate: companyid = AppContext.comp_id()
  - Web applications
    - E.g. predicate   userid = AppContext.userid()

Why VPD?

- Scalability
  - Table Customers contains 1,000 customer records. Suppose we want customers to access their own records only. Using views, we need to create 1,000 views. Using VPD, it can be done with a single policy function.

- Simplicity
  - Say, we have a table T and many views are based on T. Suppose we want to restrict access to some information in T. Without VPD, all view definitions have to be changed. Using VPD, it can be done by attaching a policy function to T; as the policy is enforced in T, the policy is also enforced for all the views that are based on T.

- Security
  - Server-enforced security (as opposed to application-enforced).
  - Cannot be bypassed.

Oracle VPD

- How does it work?

  When a user accesses a table (or view or synonym) which is protected by a VPD policy (function),

1. The Oracle server invokes the *policy function*.
2. The policy function returns a predicate, based on session attributes or database contents.
3. The server dynamically *rewrites* the submitted query by appending the returned predicate to the WHERE clause.
4. The modified SQL query is executed.

Oracle VPD: Example

- Suppose Alice has/owns the following table.

  my_table(owner varchar2(30), data varchar2(30));

- Suppose we want to implement the following policy:
  - Users can access only the data of their own. But Admin should be able to access any data without restrictions.

Oracle VPD: Example

1. Create a policy function

```
Create function sec_function(p_schema varchar2, p_obj varchar2)
Return varchar2
As
    user VARCHAR2(100);
Begin
    if ( SYS_CONTEXT('userenv', 'ISDBA') ) then
            return ' ';
    else
            user := SYS_CONTEXT('userenv', 'SESSION_USER');
            return 'owner = ' || user;
    end if;
End;
```

// userenv = the pre-defined application context
// p_obj is the name of the table or view to which the policy will apply
// p_schema is the schema owning the table or view

SYS_CONTEXT

- In Oracle/PLSQL, the sys_context function is used to retrieve information about the Oracle environment.
- The syntax for the sys_context function is:

    sys_context(namespace, parameter, [length])

- *namespace is an Oracle namespace* that has already been created.
- If the namespace is 'USERENV', attributes describing the current Oracle session can be returned.
- *parameter is a valid attribute that has been set* using the DBMS_SESSION.set_context procedure.
- *length is optional*. It is the length of the return value in bytes. If this parameter is omitted or if an invalid entry is provided, the sys_context function will default to 256 bytes

USERENV Namespace Valid Parameters

Parameter	Explanation	Return Length
AUDITED_CURSORID	Returns the cursor ID of the SQL that triggered the audit	N/A
AUTHENTICATION_DATA	Authentication data	256
AUTHENTICATION_TYPE	Describes how the user was authenticated. Can be one of the following values: Database, OS, Network, or Proxy	30
BG_JOB_ID	If the session was established by an Oracle background process, this parameter will return the Job ID. Otherwise, it will return NULL.	30
CLIENT_IDENTIFIER	Returns the client identifier (global context)	64
CLIENT_INFO	User session information	64
CURRENT_SCHEMA	Returns the default schema used in the current schema	30
CURRENT_SCHEMAID	Returns the identifier of the default schema used in the current schema	30
CURRENT_SQL	Returns the SQL that triggered the audit event	64
CURRENT_USER	Name of the current user	30
CURRENT_USERID	Userid of the current user	30
DB_DOMAIN	Domain of the database from the DB_DOMAIN initialization parameter	256
DB_NAME	Name of the database from the DB_NAME initialization parameter	30
ENTRYID	Available auditing entry identifier	30
EXTERNAL_NAME	External of the database user	256
GLOBAL_CONTEXT_MEMORY	The number used in the System Global Area by the globally accessed context	N/A
HOST	Name of the host machine from which the client has connected	54
INSTANCE	The identifier number of the current instance	30

USERENV Namespace Valid Parameters

ISDBA	Returns TRUE if the user has DBA privileges. Otherwise, it will return FALSE.	30
LANG	The ISO abbreviate for the language	62
LANGUAGE	The language, territory, and character of the session. In the following format: language_territory.characterset	52
NETWORK_PROTOCOL	Network protocol used	256
NLS_CALENDAR	The calendar of the current session	62
NLS_CURRENCY	The currency of the current session	62
NLS_DATE_FORMAT	The date format for the current session	62
NLS_DATE_LANGUAGE	The language used for dates	62
NLS_SORT	BINARY or the linguistic sort basis	62
NLS_TERRITORY	The territory of the current session	62
OS_USER	The OS username for the user logged in	30
PROXY_USER	The name of the user who opened the current session on behalf of SESSION_USER	30
PROXY_USERID	The identifier of the user who opened the current session on behalf of SESSION_USER	30
SESSION_USER	The database user name of the user logged in	30
SESSION_USERID	The database identifier of the user logged in	30
SESSIONID	The identifier of the auditing session	30
TERMINAL	The OS identifier of the current session	10

Oracle VPD: Example

2. Attach the policy function to my_table

```
execute dbms_rls.add_policy (object_schema => ‘Alice’,
                             object_name => ‘my_table’,
                             policy_name => ‘my_policy’,
                             function_schema => ‘Alice’,
                             policy_function => ‘sec_function’,
                             statement_types => ‘select, update, insert’,
                             update_check => TRUE );
```

- The VPD security model uses the Oracle *dbms_rls package (RLS* stands for row-level security)
- update_check: Optional argument for INSERT or UPDATE statement types. The default is FALSE. Setting update_check to TRUE causes the server to also check the policy against the value after insert or update.

DBMS_RLS.ADD_POLICY syntax

- DBMS_RLS.ADD_POLICY (

```
object schema IN VARCHAR2 NULL,
object_name IN VARCHAR2,
policy_name IN VARCHAR2,
function_schema IN VARCHAR2 NULL,
policy_function IN VARCHAR2,
statement_types IN VARCHAR2 NULL,
update_check IN BOOLEAN FALSE,
enable IN BOOLEAN TRUE,
static_policy IN BOOLEAN FALSE,
policy_type IN BINARY_INTEGER NULL,
long_predicate IN BOOLEAN FALSE,
sec_relevant_cols IN VARCHAR2,
sec_relevant_cols_opt IN BINARY_INTEGER NULL);
```

Oracle VPD-Example

3. Bob accesses my_table

select * from my_table;

=> select * from my_table where owner = ‘bob’;

- only shows the rows whose owner is ‘bob’

insert into my_table values(‘bob’, ‘Some data’); OK!

insert into my_table values(‘alice’, ‘Other data’); NOT OK!

- because of the check option

Policy Commands

- ADD_POLICY – creates a new policy
- DROP_POLICY – drops a policy

DBMS_RLS.DROP_POLICY (

object schema IN VARCHAR2 NULL,

object_name IN VARCHAR2,

policy_name IN VARCHAR2);

- ENABLE_POLICY – enables or disables a fine-grained access control policy

DBMS_RLS.ENABLE_POLICY (

object schema IN VARCHAR2 NULL,

object_name IN VARCHAR2,

policy_name IN VARCHAR2,

enable IN BOOLEAN);

enable - TRUE to enable the policy, FALSE to disable the policy

Column-level VPD

- Instead of attaching a policy to a whole table or a view, attach a policy only to security-relevant columns
  - Default behavior: restricts the number of rows returned by a query.
  - Masking behavior: returns all rows, but returns NULL values for the columns that contain sensitive information.

- Restrictions
  - Applies only to ‘select’ statements
  - The predicate must be a simple Boolean expression.

Column-level VPD: Example

- Suppose Alice has (owns) the following table.

Employees(e_id number(2), name varchar2(10), salary nubmer(3));

e_id	Name	Salary
1	Alice	80
2	Bob	60
3	Carl	99

- Policy: Users can access e_id's and names without any restriction. But users can access only their own salary information.

Column-level VPD: Example

1. Create a policy function

```
Create function sec_function(p_schema varchar2, p_obj
   varchar2)
Return varchar2
As
   user VARCHAR2(100);
Begin
   user := SYS_CONTEXT('userenv', 'SESSION_USER');
   return 'name = ' || user;
End;
```

Column-level VPD: Example

2. Attach the policy function to Employees (default behavior)

```
execute dbms_rls.add_policy (object_schema => ‘Alice’,
                             object_name => ‘employees’,
                             policy_name => ‘my_policy’,
                             function_schema => ‘Alice’,
                             policy_function => ‘sec_function’,
                             sec_relevant_cols=>’salary’);
```

Column-level VPD: Example

3. **Bob accesses table Employees (default behavior).** REMEMBER: default behavior restricts the number of rows returned by a query

select e_id, name from Employee;

e_id	Name
1	Alice
2	Bob
3	Carl

select e_id, name, salary from Employee;

e_id	Name	Salary
2	Bob	60

Column-level VPD: Example

2’. Attach the policy function to Employees (masking behavior)

```
execute dbms_rls.add_policy (object_schema => ‘Alice’,
                             object_name => ‘employees’,
                             policy_name => ‘my_policy’,
                             function_schema => ‘Alice’,
                             policy_function => ‘sec_function’,
                             sec_relevant_cols=>’salary’,
    sec_relevant_cols_opt=>dbms_rls.ALL_ROWS);
```

Column-level VPD: Example

3. Bob accesses table Employees (masking behavior).

REMEMBER: Masking behavior returns all rows, but returns NULL values for the columns that contain sensitive information.

select e_id, name from Employee;

e_id	Name
1	Alice
2	Bob
3	Carl

select e_id, name, salary from Employee;

e_id	Name	Salary
1	Alice	
2	Bob	60
3	Carl	

Application Context

- Application contexts act as secure caches of data that may be used by a fine-grained access control policy.
  - Upon logging into the database, Oracle sets up an application context in the user's section.
  - You can define, set and access application attributes that you can use as a secure data cache.
- There is a pre-defined application context, "*userenv*".
  - See Oracle Security Guide.

Application Context

- One can create a customized application context and attributes.
  - Say, each employee can access a portion of the Customers table, based on the job-position.
  - For example, a clerk can access only the records of the customers who lives in a region assigned to him. But a manager can access any record.
  - Suppose that the job-positions of employees are stored in a LDAP server (or in the Employee table).
  - Such information can be accessed and cached in an application context when an employee logs in.

Multiple Policies

- It is possible to associate multiple policies to a database object.
  - The policies are enforced with AND syntax.
  - For example, suppose table T is associated with {P1, P2, P3}.
  - When T is accessed by query Q = select A from T where C.
  - Q’ = select A from T where C $\wedge$ (c1 $\wedge$ c2 $\wedge$ c3).

Issue 1: Inconsistencies

- Suppose the policy authorizes each employee to see his/her own salary
- Alice issues the following query:

  SELECT AVG(*) FROM Employee

- The query will be rewritten to

  SELECT AVG(*) FROM Employee where name = “Alice”;

- What’s the problem?

Issue 2: Recursion

- Although one can define a policy against a table, one *cannot* select that table from within the policy that was defined against the table
  - That is, a policy function of an object should not access the object.
  - Suppose that a policy function PF that protects a table T accesses T.
  - When T is accessed, PF is invoked. PF tries to access T, and another PF is invoked. This results in endless function invocations.
- This cyclic invocation can occur in a longer chain.
  - For example, define a policy function for T, that accesses another table T_1. If T_1 is protected by another policy function that refers to T, then we have a cycle.
  - It is hard to check. (A policy function can even invoke a C program.)

Summary

- FGAC is a powerful access control
- Oracle VPD implements FGAC using query rewriting mechanisms
- It is difficult, if not impossible, to verify whether or not a particular user has access to a particular data item in a particular table in a particular state.
  - Such verification requires checking all policy functions.
  - As policy functions are too “flexible”, it is computationally impossible to analyze them.