IBM® DB2® for Linux®, UNIX®, and Windows®

Data security best practices
A practical guide to implementing row and column access control

Walid Rjaibi, CISSP
IBM Senior Technical Staff Member
Security Architect for
DB2 for Linux, UNIX, and Windows

Issued: April 2012

1. Introduction

To comply with various government regulations and industry standards from countries around the world, organizations need to implement procedures and methods to ensure that information is adequately protected. These regulations and standards stipulate that an individual is allowed access only to the subset of that information that is needed to perform their job. For example, according to the US Health Insurance Portability and Accountability Act (HIPAA), a doctor is authorized to view the medical records of their own patients but not the records of other patients. Similarly, according to the Payment Card Industry Data Security Standard (PCI DSS), access to cardholder data such as the credit card number must be restricted by business need-to-know. For information stored in relational databases, the ability to control data access at the row and column levels satisfies this requirement.

This paper starts by reviewing traditional methods for tackling the row and column access control problem and introduces the new row permission and column mask concepts as an elegant and more effective alternative to the traditional methods. After that, new permission and mask dependencies are discussed along with the introduction of secure functions and secure triggers. A use scenario illustrates how to use row permissions and column masks to meet required access controls. Lastly, this paper provides you with a set of best practices to follow when using row permissions and column masks.

2. Traditional row and column access control methods

Three traditional methods have been used to implement row and column access control: database views, application-based security, and label-based access control (LBAC).

2.1 Database views

For each database table that requires protection, the following steps summarize the typical steps to implement row and column access control by using database views.

1. The database developer creates either a single view that includes all the security rules that affects all the users of the table or creates separate views for each user or group of users.
2. The database developer grants the appropriate privileges on the view or views.
3. The database developer revokes access to the base table from all users and groups.
4. If multiple views are created for different users or groups, the application architect ensures that the application contains logic to route user queries to appropriate views based on their identity, their group membership, or both.
5. The database developer and the application architect test the implementation.

Database views work well when the number of different restrictions is small or the restrictions affect only large and easily identified groups of users. However, when these conditions are not true, then a number of issues arise with the use of views:

- View definitions might become complex when trying to contain all the restrictions in one view. This complexity can strain system limits and can make maintenance of the view difficult. If an alternate approach of defining many simple views that each implement restrictions for a specific set of users is used to ease view definition maintenance, then routing user requests to the correct view becomes an issue. In this case, database developers often choose to resolve the request in the application, not in the database.
- If a user can bypass the view when accessing data, for example by having direct access to the underlying tables, then the restrictions are not enforced.
- Users with DATAACCESS authority still have full access to the data.

2.2 *Application-based security*

For each database table that requires protection, the following steps summarize the typical steps to implement row and column access control in an application.

1. The application fetches all the data into the application memory then applies custom logic to filter out the result set based on the user identity. Alternatively, the application builds some or all of the filtering logic into the actual SQL statement to submit to the database so that some or all of that filtering logic is performed by the database.
2. The database developer and the application architect test the implementation.

While application-based security might seem attractive, this approach suffers from several drawbacks:

- The security policy is exposed to application programmers.
- The approach is error prone and requires extensive code reviews.
- Application changes are required to reflect changes in the security policy.
- Data is protected only when it is accessed through the application. This protection limitation hampers the ability to use tools like ad hoc query and report generation tools on the data.
- Users with DATAACCESS authority still have full access to the data.

2.3 *Label-based access control (LBAC)*

For each database table that requires protection, the following steps summarize the typical steps to implement row and column access control by using LBAC.

1. The database security administrator (a user with SECADM authority) creates the security label component and security policy objects that are needed to map the security requirements into security labels.
2. The database security administrator creates the security label objects needed for user access to the protected tables.

3. The database security administrator grants security labels and exemptions to appropriate users.
4. The database developer alters the table to add a security label column and associates a security policy object with the table.
5. The database developer and the application architect test the implementation.

While LBAC is a strong security model, it is rarely suitable for commercial customers because it requires data to be classified and has a set of fixed security rules, namely, the *no read up rule* and the *no write down rule*. LBAC and multilevel security (MLS) are generally targeted at intelligence and defense customers, which is where MLS originated. For more information about MLS and its associated rules, see the paper listed in the references section.

3. Row permissions and column masks

Row permissions and column masks are two new database concepts that are introduced to address the shortcomings of traditional row and column access control methods. They represent a second layer of security that complements the current table-privileges security model. More specifically, the table-privileges security model is applied first to determine whether a user is allowed to access the table. If the user is allowed access to the table, row permissions are applied next to determine what specific rows of the table the user has access to. Column masks are then applied to determine whether the user sees the actual or masked value in the column or a masked value thereof. For example, row permissions ensure that when Dr. Jane Smith queries the patients table, she sees only rows that represent patients under her care. Other patients are nonexistent as far as she is concerned. Similarly, a column mask defined on the phone number column of that same table ensures that Dr. Jane Smith sees only phone numbers for patients who consented to share their phone numbers with her. For other patients, the phone number would be set to NULL or masked out according to the column mask definition.

One key advantage of row permissions and column masks is that no database user is inherently exempted from them, not even higher level authorities such as users with DATAACCESS authority. The ability to manage row permissions and column masks within a database is vested solely in the user with database security administrator authority (SECADM). Thus, row permissions and column masks ensure that users with DATAACCESS authority can no longer freely access all data in the database.

Another key advantage of row permissions and column masks is that they ensure that table data is protected regardless of how that table is accessed by using SQL such as through an application, through ad hoc query tools, or through report generation tools.

Lastly, a third key advantage of row permissions and column masks is that no application changes are required to take advantage of them. That is, row and column access control is transparent to existing applications. However, row permissions and column masks represent an important paradigm shift in the sense that it is no longer what is being asked but rather who is asking what. That is, result sets for the same query change based on the context in which the query was asked and no warnings or errors are returned. Application designers and database administrators need to be aware that database queries do not see all of the data in the table unless users are granted specific permissions to do so.

3.1 Row permissions definition

A row permission is a database object that expresses a row access-control rule for a specific table. The rule is an SQL search condition that describes what set of rows a user has access to.

3.1.1 Authorization

The database security administrator authority (SECADM) is required to create, alter, or drop a row permission object.

3.1.2 SQL syntax

The SQL syntax for creating a row permission follows.

```
>>-CREATE-+------------+-PERMISSION--permission-name--ON--table-name-------->
          '-OR REPLACE-'

>----+------------------------------+--------------------------------------->
     |  .--AS--.                    |
     '--+------+--correlation-name--'
                                                              .-DISABLE-.
>----FOR ROWS WHERE--search-condition--ENFORCED FOR ALL ACCESS-+---------+-><
                                                              '-ENABLE--'
```

The main parameters of this SQL syntax are:

- *table-name*: This is the name of the table for which the row permission applies. It cannot identify a nickname, a created or declared temporary table, a view, an alias, a synonym, a typed table, or a catalog table.
- *search-condition*: This is the actual row-level access control rule. It specifies a condition that can be either true or false for a row of the table. Generally, the *search-condition* follows the same rules as a WHERE clause in a subselect SQL statement.
- DISABLE or ENABLE: This represents the state of the row permission object. A disabled row permission is not enforced while an enabled row permission is enforced. For more information, see section 3.3.

Row permission definitions are recorded in the SYSCAT.CONTROLS catalog view. Depending on its specific definition, the row permission might have a dependency on one or more database objects. This dependency is also recorded. For example, if the row permission refers to a user-defined function in its *search-condition* expression, this dependency is recorded in the SYSCAT.CONTROLDEP catalog view.

Example 1

The following row permission creates a rule that limits access to rows in the PAYROLL table to only during normal business hours.

```
CREATE PERMISSION payrollp
ON PAYROLL
FOR ROWS WHERE CURRENT TIME BETWEEN ‘8:00’ AND ’17:00’
ENFORCED FOR ALL ACCESS ENABLE;
```

3.2 Column masks definition

A column mask is a database object that expresses a column access-control rule for a specific column in a specific table. The rule is an SQL CASE expression that describes what users should see when they access the column.

3.2.1 Authorization

The database security administrator authority (SECADM) is required to create, alter, or drop a column mask object.

3.2.2 SQL syntax

The SQL syntax for creating a column mask follows.

```
>>-CREATE-+------------+-MASK--mask-name--ON--table-name--------------->
          '-OR REPLACE-'

>----+------------------------------+---------------------------------->
     |  .--AS--.                    |
     '--+------+--correlation-name--'
                                                       .-DISABLE-.
>----FOR COLUMN--column-name--RETURN—cases-expression--+---------+----><
                                                       '-ENABLE--'
```

The main parameters of this SQL syntax are:

- *table-name*: This is the name of the table for which the column mask applies. It cannot identify a nickname, a created or declared temporary table, a view, an alias, a synonym, a typed table, or a catalog table.
- *column-name*: This is the specific column in table *table-name* for which the mask applies. It cannot identify a LOB column or a distinct type column that is based on a LOB, an XML column, or a column referenced in an expression that defines a generated column.
- *case-expression*: This is the actual column-level access control rule. It specifies a CASE expression to be evaluated to determine the value to return for the column. The result of the CASE expression is returned in place of the column value in a row. The result data type, null attribute, and length attribute of the CASE expression must be identical or promotable to (in the case of the result data type) to those of *column-name*. If the data type of *column-name* is a user-defined data type, the result data type of the CASE expression must be the same user-defined data type.

- DISABLE or ENABLE: This represents the state of the column mask object. A disabled column mask is not enforced while an enabled column mask is enforced. For more information, see section 3.4.

Like row permission definitions, column mask definitions are also recorded in the SYSCAT.CONTROLS catalog view. Depending on its specific definition, the column mask might have a dependency on one or more database objects. This dependency is also recorded. For example, if the column mask refers to a user-defined function in its *case-expression*, this dependency is recorded in the SYSCAT.CONTROLDEP catalog view.

Example 2

The following column mask creates a rule that limits access to the salary column in the PAYROLL table only to users in the HR role. The NULL value is returned for users who are not members in the HR role.

```
CREATE MASK salarym
ON PAYROLL
FOR COLUMN salary RETURN
CASE WHEN VERIFY_ROLE_FOR_USER(SESSION_USER, ‘HR’)= 1
     THEN salary
     ELSE NULL
END
ENABLE;
```

3.3 Supporting functions and variables

To provide more flexibility when defining row and column access control rules, the following built-in SQL functions give you different levels of control that depend on various user attributes:

- VERIFY_ROLE_FOR_USER: Use this function where access is allowed based on membership in a database role.
- VERIFY_GROUP_FOR_USER: Use this function where access is allowed based on membership in an external group.
- VERIFY_TRUSTED_CONTEXT_ROLE_FOR_USER: Use this function where access is allowed based on membership in a database role that is acquired through a trusted context.

Example 3

The following row permission creates a rule that limits access to rows in the PAYROLL table to only during normal business hours and only users who are members of the HR role.

```
CREATE PERMISSION payrollp
ON PAYROLL
FOR ROWS WHERE CURRENT TIME BETWEEN ‘8:00’ AND ’17:00’
               AND VERIFY_ROLE_FOR_USER(USER, ‘HR’)= 1
ENFORCED FOR ALL ACCESS ENABLE;
```

Additionally, the following database-managed session variables give you even more fine-grained control when defining row and column access control rules:

- SYSIBM.TRUSTED_CONTEXT: This variable represents the name of the trusted context associated with the current trusted connection.
- SYSIBM.CLIENT_IPADDR: This variable represents the current client IP address.
- SYSIBM.CLIENT_HOST: This variable represents the current client host name.
- SYSIBM.ROUTINE_SCHEMA: This variable represents the schema name of the currently executing routine.
- SYSIBM.ROUTINE_SPECIFIC_NAME: This variable represents the specific name of the currently executing routine.
- SYSIBM.ROUTINE_TYPE: This variable represents the type of the currently executing routine (P = Stored procedure, F = Function).
- SYSIBM.ROUTINE_MODULE: This variable represents the module name of the currently executing routine.
- SYSIBM.PACKAGE_NAME: This variable represents the name of the currently executing package.
- SYSIBM.PACKAGE_SCHEMA: This variable represents the schema of the currently executing package.
- SYSIBM.PACKAGE_VERSION: This variable represents the version of currently executing package.

Example 4

The following row permission creates a rule that limits access to rows in the PAYROLL table to only during normal business hours, by users who are members of the HR role, and using the HR application which is identified by a stored procedure called HRPROCS.PROC1.

```
CREATE PERMISSION payroll-table-rules
ON PAYROLL
FOR ROWS WHERE CURRENT TIME BETWEEN ‘8:00’ AND ’17:00’
               AND VERIFY_ROLE_FOR_USER(USER, ‘HR’)= 1
               AND ROUTINE_SPECIFIC_NAME = ‘PROC1’
               AND ROUTINE_SCHEMA = ‘HRPROCS’
               AND ROUTINE_TYPE = ‘P’
ENFORCED FOR ALL ACCESS ENABLE;
```

3.3 Row permissions enforcement

In order for a row permission R1 defined on a table T1 to be enforced, the following criteria must be met:

- Row-level access control must be activated for table T1.
- Row permission R1 must be in the enabled state.
- Table T1 must be accessed within a context where row permissions apply.

3.3.1 Activating row-level access control

Row-level access control must be explicitly activated on a table in order for enabled row permissions defined on that table to be enforced. The activation of row-level access control on a table is performed through an extension to the ALTER TABLE SQL statement as follows:

```
ALTER TABLE T1
ACTIVATE ROW ACCESS CONTROL;
```

The activation of row-level access control on a table results in the automatic creation of a database-managed row permission that represents a false predicate (“1 = 0”). This particular row permission is stored in the SYSCAT.CONTROLS catalog view and can be easily recognized by its schema and name. Its schema is the same as the schema of the table on which it is defined and its name contains the prefix SYS_DEFAULT_ROW_PERMISSION. The effect of this database-managed row permission is that after row-level access control is activated on a table, rows in that table are not accessible unless some row permission is defined and enabled on that table. This latter row permission then allows access to some or all of the rows in that table depending on the criteria specified in its search condition. When more than one single row permission is defined and enabled, a row access-control search condition is derived by applying the logical OR operator to the search condition in each of these row permissions. This derived search condition acts as a filter to the table before any user specified operations such as predicates, grouping, or ordering are processed.

Similarly, the deactivation of row-level access control on a table is performed through another extension to the ALTER TABLE SQL statement as follows:

```
ALTER TABLE T1
DEACTIVATE ROW ACCESS CONTROL;
```

The database security administrator authority (SECADM) is required to either activate or deactivate row-level access control on a table.

3.3.2 Enabling a row permission

The state of a row permission can be specified at the time when that row permission is created. It can be explicitly set to enabled or disabled at that time. The default state is disabled. Additionally, the state of a row permission can be changed to either enabled or disabled through the new ALTER PERMISSION SQL statement as follows.

```
>>--ALTER PERMISSION--permission-name--+--ENABLE--+---><
                                       '—DISABLE--'
```

The database security administrator authority (SECADM) is required to either enable or disable row permissions.

3.3.3 Row permissions application context

Row permissions defined on a table are applied when that table is accessed through the following SQL statements: SELECT, INSERT, UPDATE, DELETE, and MERGE.

SELECT statement

When the table for which row-level access control is activated is referenced in a SELECT statement, all enabled row permissions that were created for the table, including the database-managed row permission, are implicitly applied by the DB2 database manager to control which rows in the table are accessible. A row access-control search condition is derived by applying the logical OR operator to the search condition in each enabled row permission. This derived search condition acts as a filter to the table before any user specified operations such as predicates, grouping, or ordering are processed.

INSERT statement
When you issue an INSERT statement against a table for which row-level access control is activated, the rules specified in all the enabled row permissions defined on that table determine whether the row can be inserted. To be inserted, the row must conform to the enabled row permissions that are defined on the table. A conformant row is a row that, if inserted, can be retrieved back by using a SELECT statement by the same user. This behavior is identical to how an insert into a symmetric view works. In other words, you cannot insert a row that you cannot select.

UPDATE statement
When you issue an UPDATE statement against a table for which row-level access control is activated, the rules specified in all the enabled row permissions that are defined on that table determine whether the row can be updated. Enabled row permissions are used as follows during UPDATE operations:

1. The enabled row permissions filter the set of rows to be updated. In other words, you cannot update rows that you cannot select.
2. The updated rows (if any) must conform to the enabled row permissions. A conformant updated row is a row that can be retrieved back using a SELECT statement by the same user. This is identical to how an update of a symmetric view works. In other words, you cannot update a row such that you can no longer select that row.

DELETE statement
When a DELETE statement is issued against a table for which row-level access control is activated, the rules specified in all the enabled row permissions that are defined on that table determine which rows can be deleted. The enabled row permissions filter the set of rows to be deleted. In other words, you cannot delete rows that you cannot select.

MERGE statement
A MERGE statement can be thought of as both an INSERT and an UPDATE operation. The processing of a MERGE follows the processing of INSERT and UPDATE.

3.4 Column masks enforcement

In order for a column mask M1 defined on column C1 of table T1 to be enforced, the following criteria must be met:

- Column-level access control must be activated for table T1.
- Column mask M1 must be in the enabled state.
- Table T1 must be accessed within a context where column masks apply.
- Column C1 must be accessed within a context where column masks apply.

3.4.1 *Activating column-level access control*
Column-level access control must be explicitly activated on a table in order for enabled column masks defined on that table to be enforced. The activation of column-level access control on a table is performed through an extension to the ALTER TABLE SQL statement as follows:

```
ALTER TABLE T1
ACTIVATE COLUMN ACCESS CONTROL;
```

Similarly, the deactivation of column-level access control on a table is performed through another extension to the ALTER TABLE SQL statement as follows:

```
ALTER TABLE T1
DEACTIVATE COLUMN ACCESS CONTROL;
```

The database security administrator authority (SECADM) is required to either activate or deactivate column-level access control on a table.

3.4.2 *Enabling a column mask*
The state of a column mask can be specified at the time when that column mask is created. It can be explicitly set to enabled or disabled at that time. The default state is disabled. Additionally, the state of a column mask can be changed to either enabled or disabled through the new ALTER PERMISSION SQL statement as follows.

```
>>--ALTER MASK--mask-name--+--ENABLE--+---><
                           '—DISABLE--'
```

The database security administrator authority (SECADM) is required to either enable or disable row permissions.

3.4.3 *Column masks application context*
Column masks defined on a table are applied when that table is accessed through a SELECT statement. They determine the values in the final result table. If a column has a column mask and the column appears in the outermost select list, the column mask is applied to the column to produce the values for the final result table. If the column does not appear in the outermost select list but it participates in the final result table, the column mask is applied to the column in such a way that the masked value is included in the result table of the materialized table expression or view so that it can be used in the final result table.

In general, a column mask is applied in the following contexts:

- The outermost SELECT clause or clauses of a SELECT or SELECT INTO statement, or if the column does not appear in the outermost select list but it participates in the final result table, the outermost SELECT clause or clauses of the corresponding materialized table expression or view where the column appears.
- The outermost SELECT clause or clauses of a SELECT FROM INSERT, SELECT FROM UPDATE, or SELECT FROM DELETE operation.

- The outermost SELECT clause or clauses that are used to derive the new values for an INSERT, UPDATE, or MERGE statement, or a SET transition-variable assignment statement. The same applies to a scalar-fullselect expression that appears in the outermost SELECT clause or clauses of the preceding statements, the right side of a SET host-variable assignment statement, the VALUES INTO statement, or the VALUES statement.

The application of column masks does not interfere with the operations of other clauses within the statement such as the WHERE, GROUP BY, HAVING, SELECT DISTINCT, and ORDER BY. The rows returned in the final result table remain the same, except that the values in the resulting rows might be masked by the column masks. As such, if the masked column also appears in an ORDER BY *sort-key*, the order is based on the original column values and the masked values in the final result table might not reflect that order. Similarly, the masked values might not reflect the uniqueness enforced by SELECT DISTINCT.

In general, column masks are not applied when the masked column appears in the following contexts:

- WHERE clauses
- GROUP BY clauses
- HAVING clauses
- SELECT DISTINCT
- ORDER BY clauses

3.5 Application exceptions

It is important that row permissions and column masks do not compromise database integrity. Consequently, row permissions and column masks are not applied during any internal operation performed by the database system itself for its own housekeeping. These internal operations include:

- Populating temporal history tables.
- Populating EXPLAIN tables.
- Populating EVENT monitor tables.
- Refreshing a materialized query table (MQT).
- Populating a staging table.
- Accessing a temporal history table to service an AS OF query.
- Accessing an MQT as part of a query reroute optimization.
- Accessing a table as part of evaluating a row-permission search condition.
- Primary key, unique key, and check constraint scans.
- Referential integrity (RI) scans of a parent or child table.
- Transition variables and transition tables in triggers.

Database integrity can be compromised if row permissions are applied in these contexts. For example, deleting a parent row typically requires deleting any child rows in a parent/child RI constraint. If a child row is not visible due to the database system applying a row permission for its own internal RI scan, that invisibility would result in an orphan row that is a violation of the RI constraint. Similarly, updating a row in a temporal table implies inserting the old version of that row in the corresponding history table. If the old version of the row cannot be inserted due to the database system applying a row permission for this internal insert operation database integrity would be compromised.

Row permissions and column masks apply to SQL operations only. This restriction means that row permissions and column masks are not applied for non-SQL utilities such as LOAD, REORG and RUNSTATS. Row permissions and column masks are applied for utilities that use SQL such as EXPORT.

4. Managing dependencies and secure objects

Row permissions and column masks definition and enforcement must coexist in harmony with other key database objects to maintain a balance between security, integrity, and performance. This section discusses how row permissions and column masks coexist in harmony with SQL packages, user-defined functions, and triggers.

4.1 SQL packages

Defining row permissions or column masks on a table, or activating row-level or column-level access control on a table have implications on any SQL package or cached dynamic SQL sections that depend on that same table. Enabled row permissions and column masks are enforced by having the SQL compiler incorporate them into the section (that is, access plan) that it generates for any SQL statement that affects the table on which those row permissions and column masks are defined and for which row-level or column-level access control is activated. Consequently, it is paramount that those sections remain in sync with row permissions and column masks at all times to ensure that SQL access to the table always follows the security policy. The following actions against a table T1 invalidate any SQL package or cached dynamic SQL sections that depend on that table:

- Activating row-level access control on T1.
- Activating column-level access control on T1.
- Deactivating row-level access control on T1.
- Deactivating column-level access control on T1.

Additionally, the following actions that affect row permissions and column masks associated with a table for which row-level or column-level access control is activated invalidate any SQL package or cached dynamic SQL sections that depend on that table:

- Creating an enabled row permission.
- Creating an enabled column mask.
- Dropping an enabled row permission.
- Dropping an enabled column mask.
- Altering the state of a row permission.
- Altering the state of a column mask.

Creating row permissions and column masks before activating row-level or column-level access control for a table is the recommended sequence to avoid multiple invalidations of packages and dynamic cached statements that reference the table.

4.2 Secure user-defined functions

When a user-defined function (UDF) appears in a predicate that affects a table for which row-level access control is activated, table data could be leaked through that UDF if the row permissions defined on that table are evaluated after the UDF predicate. For example, a user might not be authorized to see some row R1 in the table. By the time the row permissions are evaluated to discard row R1 for that user, the UDF could have read that row and done something with it such as sending it to the user in an email. To avoid such data leakage, the database system always evaluates any UDF predicate affecting a table for which row level access control is activated after the row permissions defined on that table are evaluated.

While executing the UDF predicate after the row permissions is appropriate for security reasons, it might not necessarily be appropriate for performance since forcing the UDF predicate to execute last limits the optimizer plan-selection options. To provide a balance between security and performance, a user with database security administrator authority (SECADM) or a delegate can declare the function as secure if they trust the UDF. This way, the DB2 database manager trusts the function and does not force it to be evaluated last. The UDF is evaluated in whatever order the optimizer deems necessary for best performance.

The CREATE FUNCTION SQL statement now allows declaring a function secure. Additionally, the new ALTER FUNCTION SQL statement alters the secure attribute of an existing function.

Example 5

The following SQL statement alters an existing function to mark it secure.

```
ALTER FUNCTION Func1
SECURED;
```

Changing the secure attribute of a function on which an SQL package or a cached dynamic SQL section depends might result in the invalidation of that package. This invalidation happens when that function is used in a predicate that affects a table for which row-level access control is activated as the secure attribute affects the optimizer plan selection.

4.3 Secure triggers

Triggers are used to maintain database integrity. To do so, trigger transition variables and transition tables must not be subject to row permissions defined on a triggering table for which row-level access control is activated. Otherwise, the trigger would not be able to serve its database integrity purpose. However, not subjecting those transition variables and transition tables to row permissions defined on the triggering table might open opportunities for data leakage. For example, an SQL statement in the trigger action might access sensitive data in the transition variables or transition tables. Consequently, a balance between security and integrity is needed. To provide such balance, the concept of a secure trigger is introduced. A user with database security administrator authority (SECADM) or a delegate can declare the trigger as secure to indicate that it is acceptable to create such a trigger on a table for which row-level access control is activated.

You can declare a trigger secure with the CREATE TRIGGER SQL statement. Additionally, you can alter the secure attribute of an existing trigger with the new ALTER TRIGGER SQL.

Example 6

The following SQL statement alters an existing trigger to mark it secure.

```
ALTER TRIGGER Trigg1
SECURED;
```

4.4 Automatic data movement

In some situations, data can be moved automatically by the database system from one table to another as a result of a database operation. For example, a refresh of a materialized query table (MQT) moves data from the base tables to that MQT. When a staging table is defined for an MQT, data from the base tables can also be moved to that staging table.

When row-level or column-level access control is activated for a base table, it is important to ensure that the sensitive data in that base table does not suddenly lose that protection when it is automatically moved to another table. To ensure that the protection is not lost, the database system automatically activates row-level access control on the subject table. This automatic activation ensures that direct access to the subject table sees no rows until either a user with database security administrator authority (SECADM) explicitly creates row permissions that allow access or deactivates the row-level access control on the subject table. More specifically, the automatic activation of row-level access control for a subject table happens in these situations:

- The creation of an MQT that is based on one or more tables for which row-level or column-level access control is activated. The MQT is the subject table.
- The creation of a staging table for an MQT that is based on one or more tables for which row-level or column-level access control is activated. The staging table is the subject table.
- The activation of row-level or column-level access control on a base table that is used in the definition of an MQT. The MQT and the staging table (if one is defined) are the subject tables.
- The creation of a history table for a temporal table for which row-level or column-level access control is activated. The history table is the subject table.

- The activation of row-level or column-level access control on a temporal table for which a history table exists. The history table is the subject table.
- The detaching of a partition from a partitioned table for which row-level or column-level access control is activated. The detached partition is the subject table.

Do not deactivate row-level access control on the subject table. Rather, define row permissions that ensure that the data continues to be protected as it was in the source tables. After all, the data was protected in the source tables for a reason. Additionally, it is recommended that you follow the same approach for data that you explicitly move yourself from one table to another to ensure continuous protection as the data moves from one table to another. This approach also applies to cases where the data moves from one database to another. The db2look utility can be used to extract row permission definitions in order to mimic them in another database such as a replicated database.

5. Usage scenario

This scenario presents ExampleBANK, a banking institution with a large customer base spanning many branches, as a user of row permissions and column masks. ExampleBANK uses row permissions and column masks to ensure that their database policies reflect company requirements for privacy and security, as well as management business objectives. These requirements can be summarized as follows:

- Tellers can see only their own branch customers
- Customer service representative and telemarketers can see all customers
- The account number is accessible by customer service representatives only when they are using the account update application. This application is identified through stored procedure ACTPROCS.PROCUPD. Otherwise, only the last four digits of the account number are visible. The rest of the digits are replaced with X.

Customer information is stored in a table called CUSTOMER and bank employee information is stored in a table called INTERNAL_INFO. The SQL statements for creating these two tables follow:

```
CREATE TABLE EXAMPLEBANK.CUSTOMER (
      ACCOUNT VARCHAR(19),
      NAME VARCHAR(20),
      INCOME INTGER,
      BRANCH CHAR(1)
);

CREATE TABLE EXAMPLEBANK.INTERNAL_INFO (
      HOME_BRANCH CHAR(1),
      EMP_ID VARCHAR(10)
);
```

Tellers, customer service representatives, and telemarketers are members in database roles TELLER, CSR, and TELEMARKETER respectively. SELECT privilege to the

CUSTOMER table is granted to these three roles. Users Amy, Pat, and Haytham are a teller, a customer service representative, and a telemarketer respectively. Additionally, EXECUTE privilege on procedure ACTPROCS.PROCUPD is granted to the CSR role. The SQL statements for setting up these roles are as follow:

```
CREATE ROLE TELLER;
GRANT SELECT ON EXAMPLEBANK.CUSTOMER TO ROLE TELLER;
GRANT ROLE TELLER TO USER AMY;

CREATE ROLE CSR;
GRANT SELECT ON EXAMPLEBANK.CUSTOMER TO ROLE CSR;
GRANT EXECUTE ON PROCEDURE ACTPROCS.PROCUPD TO ROLE CSR;
GRANT ROLE CSR TO USER PAT;

CREATE ROLE TELEMARKETER;
GRANT SELECT ON EXAMPLEBANK.CUSTOMER TO ROLE TELEMARKETER;
GRANT ROLE TELEMARKETER TO USER HAYTHAM;
```

We assume that tables CUSTOMER and INTERNAL_INFO are populated and their content is as follows:

ACCOUNT	NAME	INCOME	BRANCH
1111-2222-3333-4444	Alice	22,000	A
2222-3333-4444-5555	Bob	71,000	B
3333-4444-5555-6666	Carl	123,000	B
4444-5555-6666-7777	David	172,000	C

Table 1: CUSTOMER table

EMP_ID	HOME_BRANCH
Amy	A
Pat	B
Haytham	C

Table 2: INTERNAL_INFO table

To implement the first rule that states that tellers can see only customers of the teller's own branch, a user with database security administrator authority (SECADM) creates the following row permission object. The user with SECADM authority uses the VERIFY_ROLE_FOR_USER function to ensure that the user is a member of the TELLER role.

```
CREATE PERMISSION EXAMPLEBANK.TELLER_ROW_ACCESS
ON EXAMPLEBANK.CUSTOMER
FOR ROWS WHERE
VERIFY_ROLE_FOR_USER (USER, 'TELLER') = 1 AND
BRANCH = (SELECT HOME_BRANCH FROM EXAMPLEBANK.INTERNAL_INFO
          WHERE EMP_ID = USER)
ENFORCED FOR ALL ACCESS
ENABLE;
```

To implement the second rule that states that customer service representatives and telemarketers can see all customers, a user with database security administrator authority (SECADM) creates the following row permission object. The user with SECADM authority uses the VERIFY_ROLE_FOR_USER function to ensure that the user is a member of the CSR or TELEMARKETER roles.

```
CREATE PERMISSION EXAMPLEBANK.CSR_ROW_ACCESS
ON EXAMPLEBANK.CUSTOMER
FOR ROWS WHERE
VERIFY_ROLE_FOR_USER (USER, ‘CSR’) = 1 OR
VERIFY_ROLE_FOR_USER (USER, ‘TELEMARKETER’) = 1
ENFORCED FOR ALL ACCESS
ENABLE;
```

To implement the third rule that states that the account number column can be accessed only by customer service representatives and only when they are using the account update application, a user with database security administrator authority (SECADM) creates the following column mask object. The user with SECADM authority uses the VERIFY_ROLE_FOR_USER function to ensure that the user is a member of the CSR role and the routine session variables to ensure that the user is using the account update application.

```
CREATE MASK EXAMPLEBANK.CSR_COLUMN_ACCESS
ON EXAMPLEBANK.CUSTOMER
FOR COLUMN account RETURN
CASE WHEN (VERIFY_ROLE_FOR_USER(SESSION_USER, ‘CSR’)= 1
           AND ROUTINE_SPECIFIC_NAME = ‘PROCUPD’
           AND ROUTINE_SCHEMA = ‘ACTPROCS’
           AND ROUTINE_TYPE = ‘P’)
     THEN account
     ELSE 'xxxx-xxxx-xxxx-‘ || SUBSTR(ACCOUNT,13,4)
END
ENABLE;
```

Now that both rules are implemented, the user with database security administrator authority (SECADM) must activate row-level access control and column-level access control on the CUSTOMER table so that these three rules are enforced:

```
ALTER TABLE EXAMPLEBANK.CUSTOMER
ACTIVATE ROW ACCESS CONTROL
ACTIVATE COLUMN ACCESS CONTROL;
```

Now that row-level access control and column-level access control are both activated, any SQL access to the CUSTOMER table automatically respects the three rules. For example, when Amy issues SELECT * FROM EXAMPLEBANK.CUSTOMER, she can see only rows for customers from branch A, which is where Amy works. The account number is masked out.

ACCOUNT	NAME	INCOME	BRANCH
XXXX-XXXX-XXXX-4444	Alice	22,000	A

Table 3: Output for user Amy

However, when Haytham issues the exact same query as Amy, he can see all the rows in the table which is in accordance with the second rule (that is, Haytham is a telemarketer). The account number is still masked out.

ACCOUNT	NAME	INCOME	BRANCH
XXXX-XXXX-XXXX-4444	Alice	22,000	A
XXXX-XXXX-XXXX-5555	Bob	71,000	B
XXXX-XXXX-XXXX-6666	Carl	123,000	B
XXXX-XXXX-XXXX-7777	David	172,000	C

Table 4: Output for user Haytham

Lastly, when the same query is issued by user Pat through the account update application, all the rows are returned and the account number is not masked out.

ACCOUNT	NAME	INCOME	BRANCH
1111-2222-3333-4444	Alice	22,000	A
2222-3333-4444-5555	Bob	71,000	B
3333-4444-5555-6666	Carl	123,000	B
4444-5555-6666-7777	David	172,000	C

Table 5: Output for user Pat using the account update application

Any other user who is not authorized by the rules defined on the CUSTOMER table sees no rows when they attempt to access that table.

6. Best practices

This section outlines a set of recommendations to follow when implementing row-level access control by using the new row permission concept.

6.1 Dependencies

Creating, altering, and dropping row permissions or column masks as well as the activation and deactivation of row-level access control or column-level access control on a table have a consequence on existing SQL packages and dynamic cached SQL statements. To avoid multiple invalidations of these objects, create row permissions and column masks before activating row-level access control and column-level access control on the table for which those row permissions and column masks were created. Ideally, create, verify, and test the enforcement of row permissions and column masks on a test system, then use the db2look tool to extract the row permission and column masks definitions from the test system and apply them to the production system.

6.2 Data movement

Data can be moved from a source table to a target table as part of its lifecycle. When row-level access control or column-level access control is activated for the source table, take care to ensure that the sensitive data in that table does not suddenly lose that protection when it is moved to the target table. When this movement is done automatically by the database system, the database system itself automatically activates row-level access control on the target table. Do not deactivate row-level access control on this target table.

Rather, define row permissions and column masks to ensure that the data continues to be protected as it was in the source table. Additionally, follow the same approach for data that you explicitly move yourself from one table to another to ensure continuous protection. This also applies to cases where the data moves from one database to another. Use the db2look utility to extract row permission and column mask definitions in order to mimic them in another database such as a replicated database.

6.3 Three-tier application models

In a three-tier application model, a generic user ID is typically used to access the database for all requests by all users. Using a generic ID creates a challenge for database security implementations since all the database sees for access control and auditing purposes is that single generic user ID. For an effective database security implementation in three-tier environments, use row permissions with trusted context database objects. With trusted contexts, a mid-tier application can assert the user identity to the database so that it is that user identity that is used for access control and auditing purposes. Trusted contexts are supported by IBM WebSphere Application Server and IBM Cognos software. In those environments, leveraging trusted contexts is a configuration setting on the application server side. For more information about trusted contexts, see the trusted contexts paper listed in the Further reading section.

Another potential security issue in three-tier application models is data caching. Some mid-tier applications might cache a query result and reuse that result to service the same query in the future. This caching is problematic when the user identity is not taken into account when reusing the query result as different users might not necessarily have access to the same information in the database. Even when the user identity is taken into account, data caching could still be problematic if the query results cache is not kept in sync with changes to row permissions and column masks in the database. Carefully think through any data caching at the mid-tier layer when designing three-tier applications.

6.4 Performance

What is the performance impact of activating row level access control or column level access control on a table? There is no easy answer to this question as the performance varies depending on the row permissions and column masks associated with the table. For example, suppose that you have a simple table T1 and you create a row permission that represents a simple predicate like A = 5. Now, when you activate row-level access control on T1, a SELECT statement on T1 would run internally with the additional predicate A =5, which could actually be faster than when row-level access control is deactivated. So, in this case the performance improves with row-level access control active. However, if the predicate was some complex expression involving some user-defined functions, performance is reduced. So, performance depends on the nature of the row permissions defined on the table.

A better comparison to make is to compare the performance of row permissions and column masks to enforcing the same security rules in an alternative way such as enforcing them in the application. Consider the following recommendations regardless of how you determine the performance impact:

- User-defined function: Secure user-defined functions do not limit the optimizer search space. Thus, a user with database security administrator authority (SECADM) or their delegate should mark such functions secure if they are trusted.
- Objects in a row permission search condition or in a column mask case expression: Treat these objects (such as tables, indexes, and functions) following the usual performance best practices. For example, run the RUNSTATS command regularly on tables referenced in the search condition or case expression so that the statistics are up to date. Similarly, create appropriate indexes on such tables and run REORG operations on those tables and indexes should be done as required.

6.5 Problem determination

Often, database administrators are required to examine the access plans generated by the optimizer to investigate performance issues. They typically use the EXPLAIN facility to perform this investigation. By default, the output of the EXPLAIN facility represents the optimized statement, including any applicable row permissions and column masks associated with tables for which row-level access control or column-level access control is activated. In situations where you want to examine the optimized statements without any applicable row permissions and column masks, use the new NORCAC option of the EXPLAIN facility. When this new explain mode is set, the EXPLAIN facility explains the access plan as if row permissions and column masks were not present.

Additionally, database administrators might sometimes need to know what row permissions and column masks are applicable for a particular table or a particular SQL statement. This information can be easily derived from the catalog tables. For example, to determine what row permissions are applicable for the EXAMPLEBANK.CUSTOMER table, the following SQL query can be used:

```
SELECT CONTROLSCHEMA, CONTROLNAME, RULETEXT
FROM SYSCAT.CONTROLS WHERE
TABSCHEMA = ‘EXAMPLEBANK’ AND
TABNAME = ‘CUSTOMER’ AND
CONTROLTYPE = ‘R’ AND
ENABLE = ‘Y’
```

You can determine what row permissions and column masks are applicable for a particular SQL statement by first determining what tables are involved in the statement and then determining what row permissions and column masks apply for each table.

7. Conclusion

Row permissions and column masks are new powerful database security features that address the shortcomings of traditional row and column access control methods. Row permissions and column masks are based on SQL expressions and therefore provide greater flexibility. They are also transparent to applications and provide data-centric security that is managed solely by users with database security administrator authority (SECADM). Row permissions and column masks can be applied to meet the security and privacy requirements that arise in several areas, including regulatory compliance, multi-tenancy, database hosting, and data consolidation. They should be an important part of any database security strategy.

Further reading

- DB2 Trusted Contexts: Making Security and Compliance Easier, IDUG Solutions Journal, Volume 14, Number 2, 2007

- A Multi-Purpose Implementation of Mandatory Access-control in Relational Database Management Systems, in Proceedings of the international conference on Very Large Databases (VLDB), 2004

- Information Management best practices: http://www.ibm.com/developerworks/data/bestpractices/

- DB2 for Linux, UNIX, and Windows best practices: http://www.ibm.com/developerworks/data/bestpractices/db2luw/

Reviewers

Tim Vincent
IBM Fellow
VP and CTO, Information Management

Paul Bird
Senior Technical Staff Member
Optim & DB2 for Linux, UNIX, and Windows

Eric Alton
Software Developer
DB2 for Linux, UNIX, and Windows

Mihai Iacob
Software Developer
DB2 for Linux, UNIX, and Windows

Greg Stager
Software Developer
DB2 for Linux, UNIX, and Windows

Yu-Ping Ding
Software Developer
DB2 for Linux, UNIX, and Windows

Harley Boughton
Software Developer
DB2 for Linux, UNIX, and Windows

Mokhtar Kandil
Software Development Manager
DB2 for Linux, UNIX, and Windows

Paolo Cirone
Information Developer
DB2 for Linux, UNIX, and Windows

Michael Tiefenbacher
Data Management Specialist
LIS.TEC GmbH

Notices

Trademarks