

# Implementing Row- and Cell-Level Security in Classified Databases Using SQL Server 2005

SQL Server Technical Article

Writers: Art Rask, Don Rubin, and Bill Neumann

Published: April 2005

Updated: September 2005

Applies to: SQL Server 2005

Summary: This paper describes how SQL Server 2005 can be used to support row- and cell-level security (RLS/CLS). The examples provided in this white paper show how RLS and CLS can be used to meet classified database security requirements.

## Copyright

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, email address, logo, person, place or event is intended or should be inferred.

 2005 Microsoft Corporation. All rights reserved.

Microsoft, Visual C#, and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

## Table of Contents

[Executive Summary 1](#_Toc117458442)

[Assumptions and Design Principles 1](#_Toc117458443)

[Fine-Grained Access Control 1](#_Toc117458444)

[A Brief Review of Security Labeling 2](#_Toc117458445)

[Terminology 4](#_Toc117458446)

[Security labels for row- and cell-level access control 6](#_Toc117458447)

[Overview of the Database Solution 7](#_Toc117458448)

[Row-Level Security 7](#_Toc117458449)

[Defining the labeling scheme 8](#_Toc117458450)

[Roles 10](#_Toc117458451)

[What can I see? 10](#_Toc117458452)

[Changes to the base table 15](#_Toc117458453)

[Defining the view 15](#_Toc117458454)

[Insert, update, and delete 16](#_Toc117458455)

[Foreign keys 18](#_Toc117458456)

[Pulling it All Together (Part 1) 18](#_Toc117458457)

[Cell-Level Security 19](#_Toc117458458)

[Encryption in SQL Server 2005 19](#_Toc117458459)

[Key access control 21](#_Toc117458460)

[Changes to the base table 23](#_Toc117458461)

[Defining the view 24](#_Toc117458462)

[Encrypting cell data on insert/update 24](#_Toc117458463)

[Physical Partitioning 25](#_Toc117458464)

[Pulling it All Together (Part 2) 25](#_Toc117458465)

[Performance 26](#_Toc117458466)

[Row-Level Disclosure 27](#_Toc117458467)

[Predicate evaluation order 27](#_Toc117458468)

[Mitigation 28](#_Toc117458469)

[Table-valued function 29](#_Toc117458470)

[Encryption 30](#_Toc117458471)

[Application error management 31](#_Toc117458472)

[Mitigation summary 32](#_Toc117458473)

[Summary 32](#_Toc117458474)

[Appendix A – Example INSTEAD OF Trigger 33](#_Toc117458475)

[Appendix B – Symmetric Key Encryption Options 36](#_Toc117458476)

[Conclusion 36](#_Toc117458477)

#### Executive Summary

This paper reviews the requirements for row- and cell-level security based on security labels. The paper presents a Microsoft® SQL Server™2005-based design that provides row-level security based on security labels and views. Additional design elements, which allow cell-level security, are introduced. Cell-level security is also controlled by security labels.

#### Assumptions and Design Principles

The approach described in this paper makes the architectural assumption that applications using the database will connect using a specific identity for each end user. This could be either Integrated Windows authentication or a SQL Server login. This precludes the use of middle-tier connection pooling with a single identity, which is commonly used for scalability and management benefits. However, systems which have auditing and other security requirements that are significant enough to justify row-level security labeling usually have associated auditing requirements which require that the user identity be carried all the way to the database connection. Therefore, connection pooling is usually not an option anyway.

This design is also guided by these principles:

1. Rely on the SQL Server and Microsoft Windows® security model. Avoid hand-rolled authentication mechanisms.
2. Keep it simple. In some common scenarios, the number of possible security label combinations can grow very large. Avoid proliferation of fine-grained security groups, while still allowing very fine-grained control of data.
3. The design must scale to very large datasets, containing tens of millions of rows.

In general, the objective is to constrain access to data for user accounts, which are administered by privileged administrators. This design does not impose row-level or cell-level security on administrators, in particular members of the **sysadmin** or **db\_owner** fixed roles. Users with these rights will always be able to gain access to all data stored in the database.

#### Fine-Grained Access Control

Access control of information based on user permissions is a fundamental part of most computer software. Microsoft Windows, for example, uses access control lists (ACLs) to control which users can access files and folders in the NTFS file system. Microsoft SQL Server enforces access controls on the server login, databases, and on objects within a database (such as tables). In both cases, the level at which information is controlled extends only to a certain level of granularity. Windows controls access to user files, but not to portions of user files. SQL Server 2000, like most other RDBMSs, controls access to tables but does not provide row-level or cell-level security within tables.

In some scenarios, however, there is a requirement to control access at a more granular level. A list of patients and diagnoses, for example, may be stored in a single file or table. Any one doctor, however, may only be permitted to review information related to their own patients. In such a case, merely setting an ACL on a file or issuing a GRANT/DENY SELECT on a table will not meet the business requirements.

Similar scenarios exist in many environments, including finance, law, government, and military applications. Consumer privacy requirements are yet another emerging driver for finer control of data.

The typical approach to meeting such requirements in database applications has been to implement the necessary logic in application code. The business logic layer of an n-tier application might apply the filtering, for example. Or, in a two-tier client-server application, the client might do it. This approach may be effective for the application, but the data is not actually secured. A user connecting to the back-end database with Microsoft Access or a SQL query tool will have unrestricted access to all rows in tables on which they have SELECT permission.

Another common approach, which mitigates the last issue mentioned, is to wrap all data access in stored procedures. Users are denied all permissions on the underlying tables, and are instead given execute permissions on the stored procedures that implement the filtering logic. This approach has its own drawbacks. For example, ad hoc user reporting against such a database is difficult or impossible.

What is needed is a way to present the actual tables (or views) to user accounts with the filtering logic applied automatically, based on the user. In this case, all users might have access to the Patient table but, for each user, SELECT \* FROM Patient returns only the data that user should see.

Before exploring a SQL Server 2005-based solution for achieving this, we will introduce security labels, a common paradigm for defining fine-grained access control on data.

#### A Brief Review of Security Labeling

A security label is a piece of information which describes the sensitivity of a data item (an object). It is a string containing markings from one or more categories. Users (subjects) have permissions described with the same markings. Each subject has a label of their own. The subject’s label is compared against the label on the object to determine access to that object.

For example, the following table fragment has rows annotated with security labels in the Classification column.

Table 1

|  |  |  |
| --- | --- | --- |
| ID | Name | Classification |
| 1 | John Doe | SECRET |
| 2 | Frank Jones | TOP SECRET |
| 3 | Sam Barnes | UNCLASSIFIED |

A system containing this data might have user accounts as shown in Table 2.

Table 2

|  |  |
| --- | --- |
| User | Clearance |
| Alice | SECRET |
| Bob | UNCLASSIFIED (no clearance) |

Each user’s clearance (expressed as a security label) determines which rows in the table they can access. If Alice issues a SELECT \* FROM <tablename> against this table, she should get the following results.

Table 3

|  |  |  |
| --- | --- | --- |
| ID | Name | Classification |
| 1 | John Doe | SECRET |
| 3 | Sam Barnes | UNCLASSIFIED |

If Bob issues SELECT \* FROM <tablename>, he should see different results as shown in Table 4.

Table 4

|  |  |  |
| --- | --- | --- |
| ID | Name | Classification |
| 3 | Sam Barnes | UNCLASSIFIED |

Access controls can get more complex than this. There may be more than one access criterion expressed in a security label. For example, in addition to a classification level, a piece of data may only be visible to members of a certain project team. Assume this group is called PROJECT Q, and consider the following example.

Table 5

|  |  |  |
| --- | --- | --- |
| ID | Name | Classification |
| 1 | John Doe | SECRET, PROJECT Q |
| 2 | Frank Jones | TOP SECRET |
| 3 | Sam Barnes | UNCLASSIFIED |

Let’s modify our user permissions as well.

Table 6

|  |  |
| --- | --- |
| User | Clearance |
| Alice | SECRET, PROJECT Q |
| Bob | UNCLASSIFIED (no clearance) |
| Charlie | TOP SECRET |

We’ve added Charlie, a user with TOP SECRET clearance. We’ve also augmented Alice’s label with the PROJECT Q marking.

Now, if Alice issues SELECT \* FROM <tablename>, she should see the results in Table 7.

Table 7

|  |  |  |
| --- | --- | --- |
| ID | Name | Classification |
| 1 | John Doe | SECRET, PROJECT Q |
| 3 | Sam Barnes | UNCLASSIFIED |

If Charlie issues SELECT \* FROM <tablename>, he will see the following results.

Table 8

|  |  |  |
| --- | --- | --- |
| ID | Name | Classification |
| 2 | Frank Jones | TOP SECRET |
| 3 | Sam Barnes | UNCLASSIFIED |

Although Charlie has a TOP SECRET clearance, he does not have the PROJECT Q marking, so he cannot see row 1. Alice, however, satisfies both the SECRET marking and the PROJECT Q marking, so she can see row 1. Row 2, requiring a TOP SECRET clearance, is visible to Charlie only.

This basic approach can be extended to additional markings. In some real-world scenarios, security labels can include several markings from different categories, and the number of possible label combinations can be quite large.

##### Terminology

This section introduces terminology used to accurately describe labels, and the comparison of labels. A *label* is a string that describes either the sensitivity of an object, or the permissions of a subject. A label is a collection of *markings*. Each marking describes a particular permission. The markings in a label come from one or more *categories*.

Table 9 shows a breakdown of the previous example.

Table 9

|  |  |  |
| --- | --- | --- |
| Label | Markings | Categories |
| SECRET, PROJECT Q | SECRET | Classification |
| PROJECT Q | Project Team |

A subject can access an object if the subject label *dominates* the object label. Given two labels, A and B, label A is said to *dominate* label B if every category present in label B is satisfied by markings on label A. Determining whether the markings are satisfied depends on attributes of each category. For our purpose, each *category* can be characterized by the following attributes.

Table 10

|  |  |
| --- | --- |
| Attribute | Description |
| **Domain** | The possible markings in the category. |
| **Hierarchical** (yes or no) | Whether or not the category is hierarchical. Hierarchical categories have an ordering among values. This order determines access. A marking can satisfy any marking at or below its level in the hierarchy.  Nonhierarchical categories have no ordering among values. A marking is either present or not present. |
| **Cardinality** | How many values from the domain can be applied to the object. |
| **Comparison Rule** | Whether the subject must have any or all of the markings applied to the object from this category (referred to as the Any and All comparison rules, respectively). An alternative rule, InverseAll, can be used. This rule requires that each object must have all the markings held by the subject in order to be accessible. |

Here are a few examples to illustrate this. Assume we have a security labeling scheme with two categories as in Table 11.

Table 11

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Category | Domain | Hierarchical | Cardinality | Comparison Rule |
| Classification | TOP SECRET SECRET CONFIDENTIAL UNCLASSIFIED | Yes | 1..1  (exactly one) | Any |
| Compartment | Q BN G K | No | 0..\* (0, 1, or many) | All |

Now let’s look at some example labels. In each case, the question is: does label A dominate label B?

Example 1

|  |  |
| --- | --- |
| Label A | SECRET,Q |
| Label B | SECRET,Q,G |

To compare these labels, we must compare the markings in each category.

✓ Classification: The SECRET marking in label A satisfies the SECRET marking in label B.

🗶 Compartments: The Q compartment on label A does not satisfy the Q,G compartments on label B, since ALL compartments on B must be present in A.

So, label A does *not* dominate label B.

Example 2

|  |  |
| --- | --- |
| Label A | TOP SECRET,Q,G,BN |
| Label B | CONFIDENTIAL,Q,G |

✓ Classification: The TOP SECRET marking in label A satisfies the CONFIDENTIAL marking in label B.

✓ Compartments: The Q,G,BN compartments on label A do satisfy the Q,G compartments on label B, since ALL compartments on B are present on A.

So, label A *dominates* label B.

Example 3

|  |  |
| --- | --- |
| Label A | SECRET,Q,K |
| Label B | CONFIDENTIAL |

✓ Classification: The SECRET marking in label A satisfies the CONFIDENTIAL marking in label B.

✓ Compartments: Label B has no compartments listed, which means there are no compartment requirements.

So, label A *dominates* label B.

##### Security labels for row- and cell-level access control

The use of security labels is prevalent in many environments, in both the public and private sectors. The multilevel secure (MLS) information management field (in which a system securely manages information with multiple sensitivity levels) has grown up since the 1970s. So, the design presented here will be based on security labels as the paradigm for defining permissions on data and users.

#### Overview of the Database Solution

The design discussed in the following pages describes in detail an approach for adding label-based row- and cell-level security to a SQL Server 2005 database. This design is aimed at defining any arbitrary labeling scheme and enforcing it with SQL Server 2005.

The design does the following:

* Adds structures for defining arbitrary labeling categories and markings.
* Allows the user’s label to be defined intuitively through role memberships for basic markings (for example, Top Secret, Confidential; USA, UK, Taskforce Z).
* Provides for write-up, write-down, or other control models for writing data.
* Makes selective use of encryption within the database to provide cell-level security, exploiting the fully internal, self-managing certificate store in SQL Server 2005.
* Provides formulaic guidelines, such that tools could be developed to automate much of the implementation given basic input choices by developers or administrators.
* Provides strategies for protecting against certain narrow vulnerabilities in row-level security—vulnerabilities which are shared by multiple vendors’ row-level security solutions.

#### Row-Level Security

The mechanism we will use to enforce row-level security is SQL Server views. Views allow a predefined query to be presented to a user or application as if it were a table. Also, users can be granted access to a view, but denied access to underlying tables. This prevents the user from bypassing the view and going straight to the base table. We will use a specially constructed view which applies all the necessary logic to enforce row-level security based on labels.

In some scenarios, views are used to pre-cook a complex query—perhaps for a report—into a single database object that can be easily queried. We will not be doing this. Our intent is to simply wrap the base table in a view with a nearly identical definition. Users (or applications) will then query or update the view, and even join it to other tables, as if it were the table itself. Access to the base table will be denied.

Creating this view will require us to do the following four things.

1. Create some tables that define the label categories and markings, and which assign properties to each unique security label combination. This only needs to be done once for each database.
2. Create roles for the marking values. Membership in these roles will be used to express the label that is assigned to a user. This also only needs to be done once for each database.
3. Make some modifications to the base table.
4. Define the view.

##### Defining the labeling scheme

We start by creating a few tables that define the label metadata. These are shown in the ER diagram in Figure 1. The tblCategory table defines the categories which labels can include. For each category, there are a certain set of possible values—referred to as the domain of the category. These are defined in the tblMarking table. If the category is hierarchical, the parent-child relationships are defined with related records in the tblMarkingHierarchy table. The columns in these tables should be mostly self-explanatory based on the earlier discussion of security label categories.

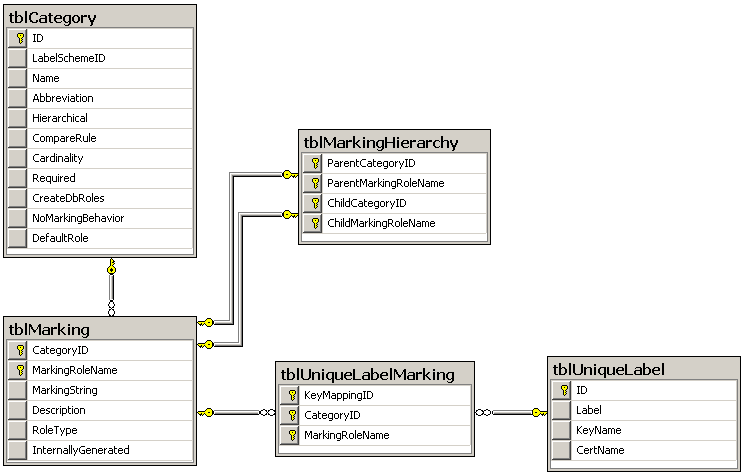


Figure 1

The following SQL statements show (roughly) how we would set up an example labeling scheme.

--Categories

INSERT tblCategory (ID, Name, CompareRule, DefaultRole)

VALUES (1, 'Classification', 'ANY', NULL)

INSERT tblCategory (ID, Name, CompareRule, DefaultRole)

VALUES (2, 'Compartment', 'ALL', 'public')

INSERT tblCategory (ID, Name, CompareRule, DefaultRole)

VALUES (3, 'Nationality', 'ANY', 'public')

INSERT tblCategory (ID, Name, CompareRule, DefaultRole)

VALUES (4, 'Need-to-Know', 'ANY', 'public')

GO

--Classification markings

INSERT tblMarking (CategoryID, MarkingRoleName, MarkingString,)

VALUES (1, 'T', 'T')

INSERT tblMarking (CategoryID, MarkingRoleName, MarkingString)

VALUES (1, 'S', 'S')

INSERT tblMarking (CategoryID, MarkingRoleName, MarkingString)

VALUES (1, 'C', 'C')

INSERT tblMarking (CategoryID, MarkingRoleName, MarkingString)

VALUES (1, 'U', 'U')

--Classification hierarchy

INSERT tblMarkingHierarchy (ParentCategoryID, ParentMarkingRoleName,

ChildCategoryID, ChildMarkingRoleName) VALUES (1, 'T', 1, 'S')

INSERT tblMarkingHierarchy (ParentCategoryID, ParentMarkingRoleName,

ChildCategoryID, ChildMarkingRoleName) VALUES (1, 'S', 1, 'C')

INSERT tblMarkingHierarchy (ParentCategoryID, ParentMarkingRoleName,

ChildCategoryID, ChildMarkingRoleName) VALUES (1, 'C', 1, 'U')

--Compartment markings

INSERT tblMarking (CategoryID, RoleName, MarkingString)

VALUES (2, 'Q', 'Q')

INSERT tblMarking (CategoryID, RoleName, MarkingString)

VALUES (2, 'G', 'G')

INSERT tblMarking (CategoryID, RoleName, MarkingString)

SELECT 2, DefaultRole, 'none' FROM tblCategory WHERE ID = 2

Etc.…

Next let’s turn our attention to the tblUniqueLabel table. This table is used to map a particular combination of markings—a specific security label instance—to a unique ID. For the sake of efficiency, this table is populated on demand. Whenever a piece of data with a given security label is added to the database, a stored procedure is called to get an ID that represents that unique label. If there is no corresponding row in tblUniqueLabel, a new row is added and the new ID returned. We want to have just as many rows in this table as we need—no more and no less.

Finally, the tblUniqueLabelMarking table associates individual marking values and the security label instances.

##### Roles

Having defined our labeling scheme, we need to do something to tie in to the SQL Server security model. For each label in every category we defined, we create a corresponding database role[[1]](#footnote-2). Database roles must adhere to the following guidelines.

* For nonhierarchical categories that use an Any or All comparison rule, create a role for each possible value. The name of the role must match the value in the tblMarking.MarkingRoleName column for that marking.
* For hierarchical categories, do the same thing. In addition, however, the roles must be nested to model the hierarchy. For each role you create, add the parent role that is defined in the tblMarkingHierarchy table as a member. This nesting ensures, for example, that users with Secret clearance can access data at that level and below.

These roles allow users to be granted permissions that precisely express the security labels to which they should have access. It is the job of the application or database administrator (DBA) to correctly maintain role memberships for the users of the system.

In the following sections we’ll see how the role memberships are used to identify which data users can access.

##### What can I see?

Before we turn our attention to the application table(s), we will create a helper view which encapsulates the actual row-level security logic. We’ll call it vwVisibleLabels. The starting point of the view definition is as in the following code example.

SELECT ID, Label.ToString()

FROM tblUniqueLabel WITH (NOLOCK)

WHERE ….

The WHERE clause definition is based on the category attributes in our labeling scheme. The important attribute is **Comparison Rule**. For each category that has a comparison rule of Any, add the following predicate to the WHERE clause.

ID IN (SELECT ID FROM tblUniqueLabelMarking WITH (NOLOCK)

WHERE CategoryID = <HardCodedCatID> AND IS\_MEMBER(MarkingRoleName) = 1)

This clause gets all the IDs of unique security labels which contain markings from the given category of which the current user is a member. Let’s look at that more closely. The subquery scans the tblUniqueLabelMarking table for rows in a certain category. From among these rows, it chooses the ones for which the current user is a member of the database role named in the MarkingRoleName column. This check is accomplished with the SQL Server intrinsic function IS\_MEMBER. For each one of these rows, the ID of the corresponding record in tblUniqueLabel is returned to the outer query.

And what about categories with a **Comparison Rule** of All? For each category that has a comparison rule of All, add the following predicate to the WHERE clause.

1 = ALL(SELECT IS\_MEMBER(MarkingRoleName) FROM tblUniqueLabelMarking (NOLOCK)

WHERE CategoryID = <HardCodedCatID> AND UniqueLabelID = tblUniqueLabel.ID)

This predicate results in comparisons that require the user to have all of the applied markings in order to have access. The subquery gets a list of values returned by IS\_MEMBER for every related marking for a given record in tblUniqueLabel. If all of these return values are 1, the predicate is satisfied.

Categories with a comparison rule of InverseAll should use this predicate in the WHERE clause:

ID IN

(SELECT KeyMappingID FROM tblUniqueLabelMarking(NOLOCK) Z

WHERE NOT EXISTS (SELECT MarkingRoleName

FROM tblMarking

WHERE IS\_MEMBER(MarkingRoleName) = 1

AND CategoryID = 1 AND InternallyGenerated =0

EXCEPT

SELECT MarkingRoleName

FROM tblUniqueLabelMarking

WHERE CategoryID = 1

AND KeyMappingID = Z.KeyMappingID))

All the predicates added should have an AND operator between them.

Pull all this together for our earlier example case and you get a definition for the intermediate view as follows:

CREATE VIEW vwVisibleLabels

AS

SELECT ID, Label.ToString()

FROM tblUniqueLabel WITH (NOLOCK)

WHERE

ID IN --Classification

(SELECT ID FROM tblUniqueLabelMarking WITH (NOLOCK)

WHERE CategoryID = 1 AND IS\_MEMBER(MarkingRoleName) = 1)

AND --Compartments

1 = ALL(SELECT IS\_MEMBER(MarkingRoleName) FROM tblUniqueLabelMarking

WHERE CategoryID = 2 AND UniqueLabelID = tblUniqueLabel.ID)

GO

This view is called vwVisibleLabels, but you could just as well think of it as “the list of all the security labels present in the database to which I (the current user) have access.”

Table 12 summarizes the design rules for setting up the labeling scheme, depending on the attributes of each category.

Table 12

|  |  |  |
| --- | --- | --- |
| Category Attribute | Design Implication | |
| **Domain** | Define all labels in the tblCategory, tblMarking, and tblMarkingHierarchy tables.  Create a database role to represent each marking, using role nesting for hierarchical categories. Membership in the role means the user has that access. | |
| **Hierarchical** | Yes | Nest roles to model hierarchy |
| No | No action |
| **Cardinality** | 1 | Constrain the number of markings on one label using triggers. Treat the **Comparison Rule** as Any. |
| 1..\* | No action |
| 0..1  or 0..\* | As above, but handle the zero markings case as described in **NoMarkingBehavior** in this table. |
| **Comparison Rule** | Any | Use the following WHERE-clause predicate in the vwVisibleLabels view.  ID IN (SELECT KeyMappingID FROM tblUniqueLabelMarking WITH (NOLOCK)  WHERE CategoryID = n AND IS\_MEMBER(MarkingRoleName) = 1) |
| All | Use the following WHERE-clause predicate in the vwVisibleLabels view.  1 = ALL(SELECT IS\_MEMBER(MarkingRoleName) FROM tblUniqueLabelMarking (NOLOCK)  WHERE CategoryID = n AND KeyMappingID = tblUniqueLabel.ID) |
| Inverse All | Use the following WHERE-clause predicate in the vwVisibleLabels view.  ID IN  (SELECT KeyMappingID FROM tblUniqueLabelMarking(NOLOCK) Z WHERE NOT EXISTS (SELECT MarkingRoleName  FROM tblMarking  WHERE IS\_MEMBER(MarkingRoleName) = 1 AND CategoryID = 1 AND InternallyGenerated =0  EXCEPT  SELECT MarkingRoleName  FROM tblUniqueLabelMarking  WHERE CategoryID = 1  AND KeyMappingID = Z.KeyMappingID)) |
| **NoMarking Behavior** | No Restriction | For each tblUniqueLabel that has zero markings in the category, add a row in tblUniqueLabelMarking assigning the public role to the label. This makes the category nonrestrictive for that label. |
| NoAccess | If **Comparison Rule** is All, add an additional WHERE-clause predicate to the vwVisibleLabels view.  EXISTS(SELECT MarkingRoleName  FROM tblUniqueLabelMarking (NOLOCK)  WHERE CategoryID = n  AND UniqueLabelID = tblUniqueLabel.ID)  If **Comparison Rule** is InverseAll, add the following code.  EXISTS(SELECT MarkingRoleName FROM tblMarking  WHERE IS\_MEMBER(MarkingRoleName) = 1  AND CategoryID = 1 AND InternallyGenerated = 0) |

Note that to handle some scenarios, there is an additional category attribute called **NoMarkingBehavior**. This controls how access is evaluated against a label with no markings from a given category. In the great majority of scenarios, no markings means that category can be ignored—it imposes no restriction. In some scenarios, however, the absence of markings in the category can mean no one is able to view the data (except an account explicitly granted permission to bypass RLS).

##### Changes to the base table

Now let’s look at the changes that must be made to the base table to which we will add row-level security. These changes are minor. (There will be more when we get to the topic of cell-level security). Two columns must be added to the base table: RowLabel and RLSMappingID. RowLabel is the raw security label for that row. RLSMappingID is the integer ID from tblUniqueLabel that corresponds to that label. Strictly speaking, only RLSMappingID is needed. In fact, having the RowLabel column is a violation of third normal form. In stringent security scenarios, however, it is often required that the literal security label that accompanied the data when it came into the database always be retained with the data. So, we have RowLabel in the base table. It may also be necessary to add additional columns for other raw source metadata about the row. Whatever the policy may be for a particular application, it should be understood that the only field that is *required* in the base table is RLSMappingID.

When a row is inserted into the base table with a security label, the ID corresponding to that label in tblUniqueLabel must be retrieved (or generated) and placed in the RLSMappingID column of the new row. Similarly, if a row is updated so that its security label changes, the RLSMappingID should change accordingly. (The updateability of the security label is dependent on the security requirements in each situation; this may be expressly forbidden).

Finally, create a foreign key relationship between RLSMappingID on the base table and the ID column in tblUniqueLabel.

For performance reasons, create a nonclustered index on the RLSMappingID column. Do not skip this or performance will suffer!

##### Defining the view

Now we’re ready for the last step. We will create a view over the base table that will, in essence, replace the table in the eyes of the end user or application. Here is what the view definition should look like.

CREATE VIEW UserTable

AS

SELECT <base table column list which does not include RLSMappingID, or any columns from vwVisibleLabels>

FROM tblBaseTable (READCOMMTTED), vwVisibleLabels

WHERE tblBaseTable.RLSMappingID = vwVisibleLabels.ID

GO

GRANT SELECT ON UserTable TO <app\_users>

DENY ALL ON tblBaseTable TO <app\_users>

GO

There are two points of interest in this view definition as follows:

* By joining the security label identifier in the base table against the Visible Labels view, the label dominance logic—which depends on user membership in security groups—controls access to rows in the base table.
* The READCOMMITTED locking hint applied to the base table prevents dirty reads against the base table. This prevents a reader from viewing rows within another user’s pending transaction which may have been updated with sensitive data, but whose security label identifier(s) have not yet changed. The locking hint in the view overrides the reader’s transaction isolation level or explicit locking hints they use when querying the view. REPEATABLE READ and SERIALIZABLE are acceptable here as well. Note that this assumes that the other user (the updater) has a transaction isolation level other than READ UNCOMMITTED.

We now have a view which transparently enforces label-based row-level security without any application logic, and in a way that remains in effect even if the application layer is bypassed.

Of course, this view is only good for selecting rows from the table. If the application using this table needs to insert, update, or delete rows, there is a little more work to do.

##### Insert, update, and delete

Everything discussed so far pertains to SELECTing rows from the underlying table. Many applications need to write information to the table as well. Inserting, updating, or deleting rows in a table with label-based row-level security raises some questions. Which rows can the user update? Are there any restrictions on the level of security label the user can apply to newly inserted rows?

The answers to these questions depend on the scenario. Some systems allow “read-down, write-down” behavior only, while others only allow “read-down, write-up” behavior. Both scenarios are supportable, using much the same techniques.

To support inserts and updates against the base table, we need to be able to do these things:

* Allow inserts or updates to the user-accessible view.
* Require valid row labels.
* Generate new label mapping IDs as needed, or resolve labels to existing IDs.
* Enforce write-down or write-up logic, as the scenario demands.
* Perform the insert/update to the base table.

This is achieved by defining INSTEAD OF triggers for insert and update. The triggers check the validity of labels, generate or retrieve label mapping IDs, enforce write permission checks, and handle the actual insert or update against the base table.

An example of an INSTEAD OF trigger for doing this shown in Appendix A. There are a few things to point out in this code.

First, take a look at the following two lines of code.

DECLARE @RowClassification SecurityLabel

SELECT @RowClassification = row\_label FROM inserted

This gets the row label that was included in the INSERT statement into an instance of a user-defined type called **SecurityLabel**. **SecurityLabel** encapsulates the values which make up a particular label. **SecurityLabel** instances can be compared with the **Dominates** function, a Microsoft Visual C#® user-defined function. This method compares two labels to see if one dominates the other. We use this method in the trigger to compare the permissions of the current user to the label(s) on the data they are inserting.

Before we can make that comparison, however, we must get an instance of **SecurityLabel** that describes the current user’s permissions. The following lines do that.

EXECUTE AS CALLER

SELECT @CallerName = CURRENT\_USER

REVERT

DECLARE @UserClearance [SecurityLabel]

exec usp\_GetUserLabel @CallerName, @UserClearance OUTPUT

**usp\_GetUserLabel** is a stored procedure that examines the role memberships of the current user, and generates a label describing their permission level. The label is returned as a **SecurityLabel** instance. With that we can compare the user’s permissions to any label we want. In the case of our example, we are implementing a write-down-only requirement, which we enforce as follows.

IF dbo.Dominates(@UserClearance, @RowClassification) = 0

RAISERROR('user rights not sufficient to write this data', 12, 1)

Similar code could be used to enforce write-up requirements. (Alternatively, if the requirements called for data to be silently labeled on insert with the user’s clearance level, we could skip the **Dominates** check and simply use the label returned from **usp\_GetUserLabel**.)

Assuming the user’s permissions are appropriate to write the data, we need to get the mapping ID for the row label. The stored procedure **usp\_GetRLSMappingID** handles this work. This procedure handles the work of retrieving the existing ID, or generating a new one, as needed.

With the mapping ID, we can proceed to do the actual insert to the base table.

The code for an INSTEAD OF update or delete trigger would use similar logic.

##### Foreign keys

Before leaving the topic of row-level security, we should address the question of tables that reference an RLS-protected table. Allowing users to see rows in a table that references restricted rows in another table constitutes a leak of information. This can be addressed when modifying the base table. When adding the RLSMappingID, this column can be added to the primary key. Whenever the primary key is referenced in another table, the row-level control will go with it, and can be used in the definition of a view over that table.

#### Pulling it All Together (Part 1)

The various aspects of the design presented here combine into a layered design pattern for outfitting an existing table with label-based row-level security. Figure 2 depicts this graphically. The bottom layer is the base table itself. Over the base table we create the view, which enforces row-level security for reads against the table. This depends on the supporting meta-information shown on the side of the diagram.

Above this RLS view, an outer view is created. This outer view selects directly from the first view. This view also includes INSTEAD OF triggers to support inserts, updates, and deletes on the table. It is not strictly necessary to separate these two views, but as we add cell-level security this model will be necessary.



Figure 2

Only the uppermost view and the vwVisibleLabels view (shaded in the diagram) are exposed to user logins.

This layered design pattern has several advantages:

* It breaks the major design points into simple pieces that are easily maintained.
* The process of applying the design to a given table is formulaic and mechanical.
* It encapsulates security-label logic for the write cases in the core constructs, rather than application code or stored procedures.

In the following sections, we improve on this pattern to include cell-level label security and to physically partition data of different sensitivities across distinct I/O devices.

#### Cell-Level Security

It is possible that data may need control at a finer level of detail than the entire row. Most of a row might need to be visible a one set of users, while certain more sensitive cells might require additional permissions to view. This is shown notionally in Figure 3. Different patterns represent the different labels that are applied to the data. While the next-to-last row is controlled with row-level security, there are several cells scattered through the table that have their own distinct labels.



Figure 3

We would like to be able to inherently control the visibility of the data based on user permissions, and do so as close to the data itself as possible. Ideally, the database engine could simply show data from labeled cells, or not, based on the identity of the connected user. Until now there has been no easy way to support such a model.

SQL Server 2005 introduces encryption capabilities in the database engine itself. These can be used to encrypt and decrypt any arbitrary data, using a certificate and key infrastructure that is internally managed by the database engine. There is no need to pass in certificates or keys from an external source.

This paper proposes a design which would accomplish this ideal level of cell-level control, in a way that is relatively easy to implement and administer.

The basic objectives of the design are:

* Support for the arbitrary labeling of cells of data.
* Dynamic evaluation of the user’s label, to show only the appropriate cells.
* Acceptable performance impact at high volume.

##### Encryption in SQL Server 2005

SQL Server provides internal functions to easily encrypt and decrypt data using a certificate, asymmetric key, or symmetric key. It manages all of these in an internal certificate store. The store uses an encryption hierarchy that secures certificates and keys at one level with the layer above it in the hierarchy. This feature area of SQL Server 2005 is called Secret Storage.

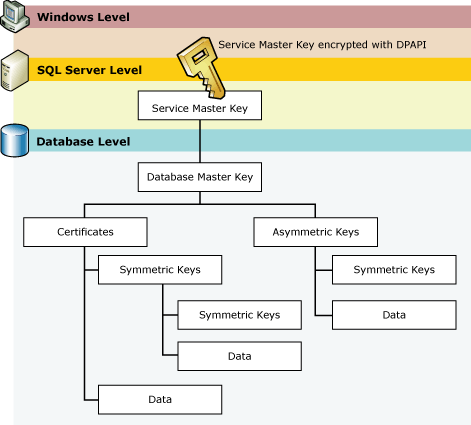


Figure 4[[2]](#footnote-3)

The fastest mode of encryption supported by the internal API is symmetric key encryption. This mode is suitable for handling large volumes of data. The symmetric keys themselves are stored encrypted by X509.v3 certificates.

SQL Server 2005 supports several symmetric key encryption algorithms. The algorithms are implemented in the Windows Crypto API. Appendix B lists the supported algorithms and the key size for each algorithm.

Within the scope of a database connection, SQL Server 2005 can maintain multiple open symmetric keys. By “open” we mean they are retrieved from the store and ready to be used for decrypting data. When a piece of data is decrypted, there is no need to specify the symmetric key to use. Instead, the engine matches the encrypted byte stream to an open symmetric key, *if* the correct key has been decrypted and is open. This key is then used to perform decryption and return the data. If the correct key is not open, NULL is returned.

The ability of a key to be “open” depends directly on the ACL on the key.

Given these mechanics of SQL Server 2005 encryption support, suppose the following approach.

1. Create a symmetric key for each unique label that is used to mark data in the database.
2. Encrypt data in labeled cells with the corresponding key.
3. Control access to keys in such a way that exactly the keys which map to labels dominated by the user’s label can be opened. Provide a simple way to have all these keys opened when the connection is established.
4. Use a view over the base table to include calls to the decryption API in the SELECT statement that defines the view.

The view definition shown below gives a simple example of what such a view might look like.

CREATE VIEW MyTable

AS

SELECT ID,

DecryptByKey(SensitiveData),

DecryptByKey(OtherSensitiveData),

NonSensitiveData,

FROM BaseTable

GO

Given this approach, let’s look at what happens when a user selects against the view. All the keys which map to labels that the user can access will have been opened. Therefore, all the cells with labels the user can access will be decrypted when the SELECT statement is executed. Conversely, all the keys that map to labels the user cannot access will not be opened. When the SELECT statement is executed, cells with these labels come back NULL, providing the user with no information on whether there is even data present in the cell.

This approach accomplishes the granular, dynamic control over data in a relational table that we are seeking.

##### Key access control

Of course the suitability of the design hinges on control of the keys. SQL Server 2005 defines permissions on keys in terms of a single SQL Server principal. The permission to open a key, for example, can be granted to a user named Bob, or to a group named AppUsers. In our scenario, however, we want to control rights to the key based on arbitrary *combinations* of principals – to be specific, based on arbitrary combinations of role memberships which define the user’s label. We want to avoid defining a principal for every possible label combination. The semantics for deciding which keys a user can access are identical to those for controlling row access.

This type of ACL’ing requires a more subtle approach. Instead of granting key permissions to users, or to roles, we grant permissions to a system-defined broker user account. We’ll call it the KeyBroker account. KeyBroker can open keys. Users and user roles are denied all permissions on keys. We get a list of the labels to which the user should have access, and ask KeyBroker to open the corresponding keys.

Using new impersonation features of SQL Server 2005, we can define a stored procedure which can be called by users, but which will ‘execute as’ KeyBroker. This procedure is shown below. A cursor is defined on the vwVisibleLabel view. Note two columns in that view which we have not mentioned yet: KeyName and CertName. The cursor selects these two columns from the view. Using another new impersonation feature, we temporarily revert to the identity of the calling user and open the cursor. This ensures that the rows returned from vwVisibleLabels are based on the calling user’s label (that is, their combination of role memberships). We then immediately revert back to the KeyBroker identity. The rest of the procedure loops through the rows in the cursor, opening each key by using the dedicated certificate used to encrypt it.

CREATE PROCEDURE usp\_EnableCellVisibility

WITH EXECUTE AS 'KeyBroker'

AS

DECLARE @KeyName nvarchar(256)

DECLARE @CertName nvarchar(256)

DECLARE Key\_Cursor CURSOR LOCAL FORWARD\_ONLY STATIC FOR

SELECT KeyName, CertName

FROM vwVisibleLabels

EXECUTE AS CALLER

--Since the cursor is STATIC, it is fully

--populated here based on the caller’s identity

OPEN Key\_Cursor

REVERT

FETCH NEXT FROM Key\_Cursor INTO @KeyName, @CertName

WHILE @@FETCH\_STATUS = 0

BEGIN

open symmetric key @KeyName using certificate @CertName

FETCH NEXT FROM Key\_Cursor INTO @KeyName, @CertName

END

CLOSE Key\_Cursor

DEALLOCATE Key\_Cursor

GO

Exit from the stored procedure automatically reverts the user context back to the calling user. This approach opens exactly those keys associated with labels that are dominated by the user’s label. The user never has any access to keys, and so cannot open any others. And, there is no input to this procedure which can be spoofed. Evaluation of rights is based only on membership in SQL Server database roles.

With the correct symmetric keys open, selecting from the view causes labeled cells to be visible if the user’s label dominates. All other labeled cells appear as NULL.

The expectation is that this stored procedure (**usp\_EnableCellVisibility**) is called once by an application or end user immediately after establishing the database connection. The keys opened would remain open for the life of the connection. A corresponding procedure (**usp\_DisableCellVisibility**) is provided to close the keys if needed. This is not strictly necessary, as closing the connection does the necessary cleanup.

##### Changes to the base table

Changes to the base table to support cell-level security are fairly minor. Most important, the data type of the column to be protected must be compatible with the encrypted data values. The intrinsic function **EncryptByKey** returns **varbinary**. This can be stored in a character or binary field (for example, **varchar**, **nvarchar**, or **varbinary**). If the original data type is not compatible with varbinary content—a numeric type such as **int**, for example—the column data type must be changed. In the next section, we’ll see how to make the data type appear unchanged to the user.

Ensuring the correct data type is the only required change. Another change that may be desirable in some scenarios is the addition of a column to hold the label that applies to the cell. The main purpose of this would be to comply with policies requiring that the label metadata live with the cell data. This could be stored as the ID from the **tblUniqueLabel** table, an instance of the **SecurityLabel** user-defined type (UDT), or the raw label string from source data.

##### Defining the view

Finally, we need to redefine the user-accessible view to include logic for decrypting data in protected cells. The next example code shows what the view definition should look like. It is almost identical to the view definition we showed previously when discussing row-level security. The only difference is that some columns are wrapped in a SQL Server 2005 intrinsic function called **DecryptByKey**. To keep the example simple, we assume just a few columns in the base table.

CREATE VIEW UserTable

AS

SELECT ID,

DecryptByKey(SensitiveData),

CONVERT(money, CONVERT(varchar(50), DecryptByKey(SensitiveMoneyData))),

NonSensitiveData,

FROM tblBaseTable (READCOMMITTED), vwVisibleLabel

WHERE tblBaseTable.RLSMappingID = vwVisibleLabel.ID

GO

GRANT SELECT ON UserTable TO <app\_users>

DENY ALL ON tblBaseTable TO <app\_users>

GO

As mentioned in the previous section, the encryption functions use character or binary data as input and output. If the original data type of a protected column is numeric, for example, the view definition should include a conversion of the **varbinary** decryption output to the original data type. An example of this is shown in the third column of the view above.

##### Encrypting cell data on insert/update

The encryption of cell data can be handled in much the same way we handled write-up/write-down checks for row-level security. INSTEAD OF triggers defined on the view handle the write-up/write-down checks, and also encrypt the cell(s) based on their label. The code in Appendix A includes an example of what this logic might look like.

#### Physical Partitioning

When designing a system that might combine data from multiple levels of classification, the issue of commingling data in physical storage must be addressed. Part of our design must allow the distribution of data with different classifications across distinct physical storage units. As with other aspects presented here, we would like to encapsulate the solution to the requirement in a unified, prescriptive reference design which lives as architecturally close to the data as possible. Applications built atop the data management solution should not have to deal with these issues.

This requirement can be addressed by using partitioned tables, another feature of SQL Server 2005. It could also be addressed by using partitioned views in the previous version of SQL Server, but with more difficulty.

Table partitioning allows the definition of a *partition function*, which uses a column in the table to divide the data into disjoint logical partitions. The partition functions are mapped to physical storage units (that is, files). These files can be placed on distinct physical devices. For an additional measure of control, each physical volume can use Windows Encrypting File System (EFS[[3]](#footnote-4)) support to apply high-strength encryption at the physical media level. (This use of EFS is only prudent if significant I/O hardware resources are available to offset the performance impact of encryption at the file system level.)

#### Pulling it All Together (Part 2)

Combining the cell-level security and physical partitioning with the previously presented row-level security model, we have the design as shown in Figure 5.



Figure 5

As discussed earlier, the techniques laid out to this point could be implemented in code in a variety of ways. However, the layered design pattern has several advantages:

* It breaks the major design points into simple, easily maintained pieces.
* The process of applying the design to a given table is formulaic and mechanical. It might be desirable to auto-script the implementation based on metadata.
* It encapsulates security-label logic for the write cases in the core constructs, rather than application code or stored procedures.

#### Performance

A key question in assessing this design is its impact on performance. An acceptable solution must implement the required security, yet have an acceptable performance impact with tables containing millions of rows.

A reference implementation of this design has been tested on a dataset containing one million rows. The test server specs were:

|  |  |
| --- | --- |
| CPU | 2 x 550 MHz |
| RAM | 512 MB |
| Disk | 4 x 18 GB @ 10,000 RPM  1 x 20 GB on Fiber channel SAN  (No EFS) |
| Operating System | Windows 2003 Standard |
| Database | SQL Server 2005 Beta 2 |
| .NET Framework | 1.1 for application code  2.0 for SQL Server code |

A detailed description of the test is beyond the scope of this document. In summary, the following results were obtained.

Impact on insert performance: ~40 percent degradation

Impact on selective queries: < 10 percent degradation

Impact on aggregate queries: ~10-50 percent degradation

The performance profile of every application is unique, and these results should not be taken as a fixed guideline. However, the testing does indicate that this design impacts performance to an acceptable degree. Note that the measured impact is on a low end (550 MHz) server with limited RAM. Assuming a similar performance impact for production applications, additional hardware resources could make up for the degradation.

#### Row-Level Disclosure

The view-based row-level security design described previously has some specific, narrow vulnerabilities. The design prevents the return of rows to which the user should not have access, whether the data is queried from an application, a reporting tool, or a direct connection with a SQL query tool. However, given the right conditions, data may be exposed *by inference* if an error message is thrown. This section describes the conditions, the potential disclosure, and presents two approaches to mitigation.

##### Predicate evaluation order

If a specifically formed query is issued against the user-accessible view that implements row-level security, an error may be returned which reveals the existence of information in the table to which the user is not supposed to have access.

The query might look like the following:

SELECT \* FROM UserTable --actually a view

WHERE LastName = ‘Smith’ AND LEN(LastName)/0 > 10

Suppose that there are no rows where LastName equals Smith to which the current user has access, but that there is at least one to which the user does *not* have access. Internally, the database server’s query optimizer will decompose any view in the statement into its parts. It will then build a query plan that is based on all of the WHERE clause conditions in the view(s) and in the actual statement’s WHERE clause. If the statement executes successfully, no information will leak. However, if the WHERE clause includes invalid predicates (such as division by zero), there is a risk of leakage. *Depending on the query plan generated by the query optimizer*, the invalid predicate may be evaluated before the predicate which restricts row visibility. If this happens, an error will be thrown that implies the existence of a row that should not be ‘seen’.

Here is how it might work with the example statement above. Suppose UserTable is actually a view defined per the design presented earlier. The actual query as seen by the optimizer will be something like:

SELECT \*

FROM BaseTable (READCOMMITTED)

WHERE (LastName = ‘Smith’ AND LEN(LastName)/0 > 10)

AND BaseTable.RLSMappingID IN

(SELECT ID FROM tblUniqueLabel

WHERE ID IN .. /\* category predicates \*/)

Suppose the optimizer chooses to first use an index on LastName to narrow the result set before the other conditions are evaluated. It will find the row(s) where LastName =’Smith’, then evaluate LEN(LastName)/0 > 10. This will immediately throw an exception, prior to the evaluation of the row security predicates. The result may be that the presence of at least one row with LastName = ‘Smith’ is revealed, even if the user should not see any of those rows.

This issue is not unique to view-based row-level security with SQL Server. Other vendors’ row-level security solutions which inject predicates behind the scenes are subject to the same risk, since they inject the predicate before the query plan is generated.

It is important to observe the conditions which must be met for this vulnerability to be an issue:

1. A user must be able to directly submit a maliciously formed SQL query, either by using a SQL query tool, or SQL injection against a poorly written front-end application.
2. The user must know enough about the table definition to write the query.
3. The query optimizer must choose a plan which allows the invalid predicate to evaluate before the row access predicate(s). This is a function of several variables, including what indexes exist, the distribution of data in the table, and the nature of the other predicates in the statement.
4. The user must be able to directly see the error that is generated. This is a given with a SQL query tool, but a well-written application will not return raw error information.

##### Mitigation

In many cases, the chances of exploiting the vulnerability previously described will be deemed small enough that no action is needed. If, however, it is absolutely necessary to ensure that this vulnerability is mitigated, three options are recommended. All use the same basic design explained in this paper, but add an additional piece.

###### Table-valued function

One way to eliminate the vulnerability is to ensure that the row-access predicate is fully applied to the underlying data before any user-supplied predicates. This can be achieved using a table-valued function (TVF). The role of this TVF can easily be seen in a modified version of Figure 5.



Figure 6

Figure 6 is identical to Figure 5, except that the view which implements row-level security is wrapped in a TVF. The TVF is then queried by the user-accessible view. The definition on the TVF would be as shown in the following code.

CREATE FUNCTION [dbo].[fn\_MyTable\_tvf]()

RETURNS @ret TABLE

( <column specs from underlying RLS view> )

AS

BEGIN

INSERT @ret

SELECT <columns>

FROM vwRLSView

RETURN

END

GO

Note that the TVF must be a multi-statement table-valued function. An inline table-valued function will not have the intended effect.

Wrapping the row-access view will force it to be fully materialized as a table variable before any user-supplied predicates are evaluated. This eliminates the vulnerability.

The drawbacks to this approach are:

1. Additional complexity.
2. Performance. This requires the entire underlying table, with row-level security applied, to be buffered in a table variable which is then queried by the user-accessible view. For small tables, the impact is minor. For large datasets (millions of rows), the performance impact is severe.

###### Encryption

The second mitigation involves using the same encryption strategy as that described for cell-level security. Again, assume that all other aspects of the design are as shown in Figure 5. However, we will add an additional level of encryption to the data in each row. In addition to the cell-level encryption, every user-accessible column will be encrypted with the symmetric key that is associated with the row label. This ensures that, regardless of the query plan, the user will only read data values if their label allows access.

For example, assume a simple table with four columns. The table is protected with row- and cell-level security as depicted in Figure 5. The contents of one row might be shown as in the following table.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Column 1** | **Column 2** | **Column 3** | **Column 4** | **RLSMappingID** |
| Data1 | E(Data2, K1) | Data3 | E(Data4, K2) | 19 |

This row is labeled for row-level access control. The label is mapped to the record in tblUniqueLabel with an ID of 19. Columns 2 and 4 are protected with cell-level security. In the example row shown, column 2 is encrypted with a key for label 1 (K1) and column 4 is encrypted with a key for label 2 (K2). What the labels are doesn’t matter for this example.

This mitigation approach would apply additional encryption, so that the data stored in the underlying table looks like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Column 1** | **Column 2** | **Column 3** | **Column 4** | **RLSMappingID** |
| E(Data1, K19) | E(E(Data2, K1), K19) | E(Data3, K19) | E(E(Data2, K2), K19) | 19 |

Every user-accessible column is encrypted with the symmetric key mapped to label 19 (K19). This work is done in the INSTEAD OF triggers. The user-accessible view in Figure 5, which includes decrypt statements, would be modified to include additional decrypts as shown in the following code.

CREATE VIEW UserTable

AS

SELECT DecryptByKey(Column1),

DecryptByKey(DecryptByKey(Column2)),

DecryptByKey(Column3),

DecryptByKey(DecryptByKey(Column4))

FROM vwRLSView

WHERE ………… --omitted for clarity

The same principles described for managing open keys for cell-level security apply here. Only if the user’s label (role memberships) were appropriate would K19 be open. Therefore, no potential query plan could access impermissible data and disclose information.

This approach has drawbacks as well:

1. Additional complexity
2. Performance. More CPU time will be spent on encryption/decryption. Also, since all searchable columns are encrypted, SQL indexes cannot be used to speed up queries. Despite these issues, it is likely that this approach would scale to large datasets better than the TVF approach.
3. Loss of DBMS integrity enforcement. The database cannot enforce integrity constraints such as unique indexes or foreign key constraints on columns that contain encrypted data.

###### Application error management

The first two mitigation approaches addressed the issue solely at the database layer. A third option is to address it in other parts of the overall application. In this approach, some small risk is accepted that errors may occur which could constitute covert channels or inference channels. At an intervening layer of the solution (for example, the object-oriented data access layer) careful error trapping and logging is implemented. By trapping specific errors from the database, and propagating only a general failure message to the user, the amount of detail that is available to support inference is reduced. And, logging of errors provides an audit trail, which can be monitored for suspicious patterns that would indicate a user is attempting to exploit the system.

###### Mitigation summary

Whether or not either mitigation approach should be used is a judgment call based on the situation. It is possible that the threat scenario for a given application makes the chances of exploitation so insignificant that the penalty in complexity and performance is not worthwhile. These approaches are available to eliminate the vulnerability, if necessary.

#### Summary

The design presented in this paper enables a SQL Server 2005 database to support row- and cell-level security based on an arbitrary security label scheme. Access restriction on rows and cells is enforced inside the database by using intrinsic structures such as views and SQL Server data encryption. The evaluation of user access is based on the intrinsic SQL Server security model. This assembly of thoroughly tested, secure elements is superior to approaches that use ad hoc, custom code to do the primary authorization and filtering steps.

The ability to control access at the *cell level* based on security labels, intrinsically within the relational engine, is an unprecedented capability. It has the potential to markedly improve the management of sensitive information in many sectors, and to enhance the ability to leverage data quickly and flexibly for operational needs.

#### Appendix A – Example INSTEAD OF Trigger

This code illustrates example logic only. In the interests of clarity, the example deliberately only works with single-row operations.

CREATE TRIGGER dbo.IO\_Insert\_titles ON titles

INSTEAD OF INSERT

AS

DECLARE @RLSMappingID int

DECLARE @KeyMappingID int

DECLARE @KeyName nvarchar(256)

DECLARE @CertName nvarchar(256)

DECLARE @KeyGUID uniqueidentifier

DECLARE @KeyAlreadyOpen bit

DECLARE @CallerName sysname

BEGIN TRY

--Bail if @@ROWCOUNT > 1 (temporary)

IF @@ROWCOUNT > 1

RAISERROR('Only one row at a time for sample!', 12, 1)

--Ensure row\_label and advance\_label are NOT NULL

DECLARE @RowClassification SecurityLabel

DECLARE @AdvanceClassification SecurityLabel

SELECT @RowClassification = row\_label,

@AdvanceClassification = advance\_label

FROM inserted

IF @RowClassification IS NULL

RAISERROR('row\_label is required', 12, 1)

IF @AdvanceClassification IS NULL

RAISERROR('advance\_label is required', 12, 1)

--Check write-down logic

EXECUTE AS CALLER

SELECT @CallerName = CURRENT\_USER

REVERT

DECLARE @UserClearance SecurityLabel

exec usp\_GetUserLabel @CallerName, @UserClearance OUTPUT

IF @UserClearance IS NULL

RAISERROR('user rights not sufficient to write this data', 12, 1)

IF dbo.Dominates(@UserClearance, @RowClassification) = 0

RAISERROR('user rights not sufficient to write this data', 12, 1)

IF dbo.Dominates(@UserClearance, @AdvanceClassification) = 0

RAISERROR('user rights not sufficient to write this data', 12, 1)

--Get RLSMappingID for row\_label

exec usp\_GetRLSMappingID @RowClassification, @RLSMappingID OUTPUT

--Get KeyName and CertName for advance\_label

exec usp\_GetLabelDetails @AdvanceClassification, @KeyMappingID OUTPUT, @KeyName OUTPUT, @CertName OUTPUT, @KeyGUID OUTPUT

exec usp\_IsKeyOpen @KeyName, @KeyAlreadyOpen

EXEC('open symmetric key ' + @KeyName + ' using certificate ' + @CertName)

--Do insert, including rls\_mapping\_id and encryption of advance

INSERT INTO tblTitles

(title\_id, title, type, pub\_id, price, advance, advance\_encrypted,

advance\_label, royalty, ytd\_sales, notes, pubdate, rls\_mapping\_id)

SELECT title\_id,

title,

type,

pub\_id,

price,

EncryptByKey(@KeyGUID, CONVERT(varchar(50), advance)),

1,

advance\_label,

royalty,

ytd\_sales,

notes,

pubdate,

@RLSMappingID

FROM inserted

END TRY

BEGIN CATCH

DECLARE @ErrorMessage NVARCHAR(400);

DECLARE @ErrorSeverity INT;

DECLARE @ErrorState INT;

SELECT @ErrorMessage = ERROR\_MESSAGE();

SELECT @ErrorSeverity = ERROR\_SEVERITY();

SELECT @ErrorState = ERROR\_STATE();

IF @@TRANCOUNT > 0

ROLLBACK

RAISERROR (@ErrorMessage, @ErrorSeverity, @ErrorState);

END CATCH

--Whether successful or not, ensure any key opened by this routine is ---closed

BEGIN TRY

IF @KeyName IS NOT NULL AND @KeyAlreadyOpen = 0

EXEC('close symmetric key ' + @KeyName)

END TRY

BEGIN CATCH

--suppress error

END CATCH

GO

#### Appendix B – Symmetric Key Encryption Options

SQL Server 2005 supports several algorithms for symmetric key encryption. Supported algorithms and the associated key sizes are listed in the table.

Table 13

|  |  |
| --- | --- |
| Algorithm | Key Size |
| AES\_256 (Rijndael 256) | 256 |
| AES\_192 (Rijndael 192) | 192 |
| AES\_128 (Rijndael 128) | 128 |
| DES | 64 |
| TRIPLE\_DES | 128 |
| RC2 | 128 |
| RC4 | 40 |
| DESX | 192 |

The choice of which encryption algorithm to use is easily controlled in the SQL DDL for creating symmetric keys. Any of these algorithms can be used with the design described in this document.

#### Conclusion

For more information:

<http://www.microsoft.com/technet/prodtechnol/sql/default.mspx>

Did this paper help you? Please give us your feedback. On a scale of 1 (poor) to 5 (excellent), [how would you rate this paper?](mailto:sqlfback@microsoft.com?subject=Feedback:%20Implementing%20Row-%20and%20Cell-Level%20Security%20in%20Classified%20Databases%20Using%20SQL%20Server%202005)!href(mailto: sqlfback@microsoft.com?subject=Feedback: Implementing Row and Cell Level Security in Classified Databases Using SQL Server 2005)

1. If using Integrated Windows authentication with the SQL login mapped to a Windows user account, a Windows group can be used as well. This alternative applies to all discussions of database roles throughout this white paper. [↑](#footnote-ref-2)
2. Source: SQL Server 2005 Beta 2 Books Online [↑](#footnote-ref-3)
3. EFS is supported on Windows 2000 or later [↑](#footnote-ref-4)