**Engine Separation of Duties for the Application Developer**

SQL Server Technical Article

**Writer:** Craig Gick, Jack Richins

**Technical Reviewer:** Raul Garcia, Il-Sung Lee

**Published:** September 2008

**Applies to:** SQL Server 2008

**Summary:** Separation of duties is an important consideration for databases and database applications. By properly defining schemas and roles, you can create a distinction between users who can manipulate data from those that administer the database. This paper discusses the topics of which application developers should be aware and provides a heuristic example to guide you in achieving separation of duties.

Introduction

Developing security as part of the application development lifecycle is critical to the future manageability of a new application. This paper details several very simple approaches a developer can take to make applications more manageable from the security perspective. We will take a look at many of the features available in Microsoft® SQL Server® 2008 and how they can aid you in creating applications where proper separation of duties is enabled by design. It is not a goal of this paper to address databases where many applications already exist, only to give developers the tools to create permission isolation for newly developed applications.

Common Issues

**The dbo Schema**

By far, one of the most important parts of creating applications with security in mind is to never use the dbo schema. Today, this schema is widely overused and creates a situation where proper separation of duties is more difficult. By creating all database objects in this schema, all applications share the same namespace as well as the same owner. This sharing means ownership chaining can be used between all objects in this schema, whether intended or not. By taking the time to isolate applications to their own schemas as well as their own owners, you can isolate the application such that ownership chaining is available only between intended objects. This isolation makes security easier to manage.

**Ownership Chaining**

Even if an application uses its own unique schemas, another common issue is the case where these schemas are all owned by the dbo database principal in the database. As is the case with the dbo schema, when all schemas are owned by the dbo principal, all objects in these schemas are chainable to all objects in the dbo schema (or additionally, any other schema owned by dbo). When you design the schemas for an application, take the time to also create an ownership role unique to the application. By creating this role, you can ensure ownership chaining works for all schemas used but also isolate the application from other schemas in the database. This way you can manage the application by using membership in a flexible database role defined for application owners instead of requiring membership in a higher level principal like dbo or db\_owner.

**Multipart Names**

As noted earlier, taking the time to create well named schemas for a new application as well as creating a unique owner role can make security management easier. Multipart names are an important part of this scenario. Your application should refer to all objects by using at least a two-part name instead of referring to all objects simply by the object name (and relying on name resolution). Doing so ensures the application can query all objects as intended and that the application works smoothly if attached to other SQL Server 2008 instances regardless of the calling context. For example, the following code sets the default schema for the principal to the application schema, assuming that all users will have the same default schema.

Transact-SQL

[[http://i.msdn.microsoft.com/Global/Images/clear.gif](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl06T-SQL');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl06T-SQL');)

-- Executing database principal has a DEFAULT\_SCHEMA = [app\_schema]

--   In this case the application developer assumes all users trying

--   to access the application table have the same default schema, which

--   is not always the case. If a database principal did not have the assumed

--   default schema, this command would fail.

SELECT \* FROM [app\_table]

The following example solves this problem by using a two-part naming convention in the SELECT statement. The user connecting to the application database has a default schema set to the location it prefers to access data from. This now is completely irrelevant to the application executing the SELECT statement.

Transact-SQL

[[http://i.msdn.microsoft.com/Global/Images/clear.gif](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl07T-SQL');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl07T-SQL');)

-- Executing database principal has a DEFAULT\_SCHEMA = [contacts]

--   In this case the executor does not have a default schema bound

--   to the assumed application. Instead the default schema is

--   set to that of the application the principal uses most.

SELECT \* FROM [app\_schema].[app\_table]

User-Schema Separation

User-schema separation is a feature that was added in SQL Server 2005. This feature separates the schemas in the database from database principals. In earlier versions, objects could only be created in schemas that shared the same name of a database principal already in existence. For example, a table could only be created in the schema [DOMAIN\username] where [DOMAIN\username] is a valid database principal. Application developers had to create applications in schemas solely defined by the names of principals. This situation had many manageability issues in that the user name could require changing, leave or change roles in the company (and no longer be valid in the database). Most application developers would simply fall back on the dbo schema for all objects where all that was needed was the ability to create schemas with meaningful names particular to the application.

This issue can be resolved by using the SQL Server 2005 (and later) data definition language (DDL) for creating users and schema. Schemas can now be treated as containers in the security model, much like folders are treated in a file system security model. The behavior of this DDL is that database principals can be created and have their own default schemas for name resolution they intend, and they are separate from the schemas applications require. The schema DDL can be used to create schemas with meaningful names, and isolated owners and the objects applications use can be contained in schemas with meaningful names particular to the required design. The following example creates such schemas, which will be built upon in later sections.

Transact-SQL

[[http://i.msdn.microsoft.com/Global/Images/clear.gif](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl09T-SQL');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl09T-SQL');)

-- Meaningful, well named schemas to contain customer data and a second

--    one that contains critical encrypted data. The ssn column is

--    varbinary because only encrypted data is intended to be inserted

--    into this column.

CREATE SCHEMA [customer\_data]

                CREATE TABLE [customer\_data].[address]

(

id int PRIMARY KEY,

address nvarchar(1024)

)

go

CREATE SCHEMA [customer\_ssn]

                CREATE TABLE [customer\_ssn].[ssn]

                                (

                                                id int REFERENCES [customer\_data].[address] ( id ),

                                                ssn varbinary ( 8000 )

                                )

go

This simple example application now has two well defined schemas: one for customer data and a second one for critical customer data. The application has complete control over these schemas, and it does not depend on a particular database principal for its naming scheme.

However, the default behavior for the DDL is that the schemas will be owned by the principal executing the command. Most often this means they will be owned by dbo or some other member of the db\_owner fixed database role. This issue can be resolved using flexible database roles.

Flexible Database Roles

Flexible database roles give you the ability to isolate permissions required for different types of users of the application. In our example, we will create several roles to demonstrate how simple it is to create this isolation.

Transact-SQL

[[http://i.msdn.microsoft.com/Global/Images/clear.gif](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl11T-SQL');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl11T-SQL');)

-- Create an owner of all schemas for this application. This allows the developer

--   to use ownership chaining as a performance enhancement when intended

--   as part of the design.

CREATE ROLE [customer\_owner];

go

-- Create an isolated user that can only exist in the database to

--    be used as the owner of cryptographic objects.

CREATE USER [customer\_owner\_user] WITHOUT LOGIN;

EXEC sp\_addrolemember [customer\_owner], [customer\_owner\_user];

go

-- Create a manager of the schema. Members of this role will be able to alter

--    and drop the objects that exist the schemas, but they should not

--    have the ability to directly access data via DML. Because they

--    can alter the definition of data, there are cases where

--    data can be inferred. For scenarios where this inference is an

--    issue, SQL Server Audit can be used to monitor the activity of this role.

CREATE ROLE [customer\_manager];

GRANT ALTER ON SCHEMA::[customer\_data] TO [customer\_manager];

GRANT ALTER ON SCHEMA::[customer\_ssn] TO [customer\_manager];

DENY SELECT, UPDATE, INSERT, DELETE, EXECUTE

ON SCHEMA::[customer\_data] TO [customer\_manager];

DENY SELECT, UPDATE, INSERT, DELETE, EXECUTE

ON SCHEMA::[customer\_ssn] TO [customer\_manager];

go

-- Create a role with the ability to select from any object in the schema

CREATE ROLE [customer\_select];

GRANT SELECT ON SCHEMA::[customer\_data] TO [customer\_select];

GRANT SELECT ON SCHEMA::[customer\_ssn] TO [customer\_select];

go

-- Create a role with the ability to insert into any object in the schema

CREATE ROLE [customer\_insert];

GRANT INSERT ON SCHEMA::[customer\_data] TO [customer\_insert];

GRANT INSERT ON SCHEMA::[customer\_ssn] TO [customer\_insert];

go

-- Create a role with the ability to select or insert from any object

--    in the schema as well as decrypt or encrypt critical data.

--    This role will be augmented in subsequent sections.

CREATE ROLE [customer\_crypto];

EXEC sp\_addrolemember [customer\_select], [customer\_crypto];

EXEC sp\_addrolemember [customer\_insert], [customer\_crypto];

go

The example now has two well formed schemas as well as an owner unique to this application. There is an owner role for this application with complete access to all data in this schema, the ability to grant permissions in this schema, and the ability to alter and drop in this schema. It should be noted that the default for role DDL is that the owner is that of the executor of the command (dbo more often than not) and can be further isolated to an overall role manager by using the AUTHORIZATION clause of the CREATE ROLE command.

Next, an example manager role is created for the schemas. This manager role has all the actions enabled by using the ALTER permission on the schema. This means it can complete actions such as TRUNCATE TABLE, ALTER PROCEDURE, or altering objects in the schema. If you deny explicit DML operations to this role, no direct access to data should be allowed. For example, the manager role can alter the body of a stored procedure in this schema, but it does not have access to execute it. The manager role cannot use ownership chaining to access data it should not be able to view. Because this manager role can change the definition of data in the example schemas, it can infer data by using column alters or similar actions. For these scenarios, SQL Server Audit can be used to monitor the activities of the manager role members in order to guarantee they perform in accordance to company policies.

A pair of DML-based roles is created to enable members to select or insert data into the schemas. Another role manages who can insert encrypted data into the schemas as well has who can select decrypted data. The select and insert roles are an example of the type of roles you can use at the schema scope to provide duty separation. Roles can be created for any of the DML commands, and they can be used by applications for any specific DML operation required.

Note: Careful analysis of this example would reveals that members of the insert or select roles could insert or select data from either table even though the intention is to have only encrypted data in the ssn table. This can be addressed by further restriction (only granting select and insert permissions on the appropriate schema and or objects). The example was simplified for the purposes of this white paper.

Encrypted Data

In order to encrypt and or decrypt data from the example objects, a symmetric key needs to be created. In the following example, we will create this object and create isolation between those with simple select and insert abilities and those with the ability to select and insert encrypted data.

Transact-SQL

[[http://i.msdn.microsoft.com/Global/Images/clear.gif](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl13T-SQL');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl13T-SQL');)

-- Create a symmetric key protected by an asymmetric key.

--    Note that this example assumes the database already has a master key.

--    This key is owned by the owner of all customer objects required

--    by the application. The symmetric key is protected by an

--    asymmetric key that is owned solely by the owner of the

--    customer application.

CREATE ASYMMETRIC KEY [customer\_asymkey\_primary]

AUTHORIZATION [customer\_owner\_user] WITH ALGORITHM = RSA\_1024;

CREATE SYMMETRIC KEY [customer\_symkey]

AUTHORIZATION [customer\_owner\_user] WITH ALGORITHM = AES\_256

ENCRYPTION BY ASYMMETRIC KEY [customer\_asymkey\_primary];

go

-- Properly assign permissions for the cryptographic role. In order to open

--    the symmetric key, this role will need to have control permission

--    on an asymmetric key that protects it. Because control permission

--    gives the grantee full control over the object (which means it

--    can drop the object), the primary asymmetric key was

--    created earlier to prevent data loss.

CREATE ASYMMETRIC KEY [customer\_asymkey\_secondary]

AUTHORIZATION [customer\_owner\_user] WITH ALGORITHM = RSA\_1024;

OPEN SYMMETRIC KEY [customer\_symkey]

                DECRYPTION BY ASYMMETRIC KEY [customer\_asymkey\_primary];

ALTER SYMMETRIC KEY [customer\_symkey]

                ADD ENCRYPTION BY ASYMMETRIC KEY [customer\_asymken\_primary];

CLOSE SYMMETRIC KEY [customer\_symkey];

GRANT CONTROL ON ASYMMETRIC KEY::[customer\_asymkey\_secondary]

TO [customer\_crypto];

go

The cryptographic role in this example now has permission to open the symmetric key to be used for encryption or decryption. The primary asymmetric key can only be managed to the customer owner role and exists to prevent data loss. The secondary key can be used by the cryptographic role in order to open the symmetric key so that EncryptByKey or DecryptByKey can be called as needed.

The secondary asymmetric key is created because in order to open the symmetric key, you need control permission on it to decrypt it. The control permission allows you to drop the asymmetric key itself or drop its private key. If this were the only key protecting the symmetric key, it would create a situation for potential data loss. The primary key is created solely to prevent this scenario.

Signed Modules

So far the example in this paper has used the permissions available in SQL Server 2008 as well as the objects available in the permission model to create isolation using only Transact-SQL commands. This means members of the roles defined in the examples are restricted only by the permissions checks in the various Transact-SQL commands as to what actions they are able to take in the database. There has been no further restriction as to what commands they are able to call.

Let’s augment this example and demonstrate a case where this may not go far enough. Depending on the requirements of the application, it may be necessary to restrict actions further. In the example so far the cryptographic role is used to encrypt and decrypt data. As noted earlier, this role needs control permission on an asymmetric key, which means it can drop the asymmetric key. Let’s say the requirements have changed and separate roles need to exist for encryption and decryption. By definition, symmetric keys are used for both of these actions, so separate actions would need to be defined.

By using signed modules, you can define this type of separation and narrowly create procedures or functions that allow a specific set of actions definable by the needs of the application. This example creates two distinct roles, one that can only encrypt, and one that can only decrypt. It uses signed stored procedures, where the procedures themselves have only the permissions required to complete the actions they define. For more information about module signing, see SQL Server Books Online.

Transact-SQL

[[http://i.msdn.microsoft.com/Global/Images/clear.gif](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl15T-SQL');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl15T-SQL');)

-- Create the roles for encryption and decryption.

CREATE ROLE [customer\_encrypt] AUTHORIZATION [customer\_owner];

CREATE ROLE [customer\_decrypt] AUTHORIZATION [customer\_owner];

go

-- Define procedures that limit the actions members of these

--    roles are able to complete. These procedures are contained

--             in an isolated schema as demonstrated earlier.

CREATE SCHEMA [customer\_modules] AUTHORIZATION [customer\_owner];

go

CREATE PROCEDURE [customer\_modules].[encrypt]

( @ssn nvarchar( 11 ), @out\_data varbinary( 8000 ) OUT ) AS

BEGIN

                DECLARE @data varbinary( 8000 );

                BEGIN TRY

                                OPEN SYMMETRIC KEY [customer\_symkey]

                                                DECRYPTION BY

ASYMMETRIC KEY [customer\_ asymkey\_primary];

                                SELECT @data = EncryptByKey( Key\_GUID( 'customer\_symkey' ),

                                                                                                                @ssn );

                                CLOSE SYMMETRIC KEY [customer\_symkey];

                END TRY

                BEGIN CATCH

                                CLOSE SYMMETRIC KEY [customer\_symkey];

                    DECLARE @ErrorMessage nvarchar(4000);

                                DECLARE @ErrorSeverity int;

                                DECLARE @ErrorState int;

                                SELECT

                                                @ErrorMessage = ERROR\_MESSAGE(),

                                                @ErrorSeverity = ERROR\_SEVERITY(),

                                                @ErrorState = ERROR\_STATE();

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

                END CATCH

                SET @out\_data = @data;

END

go

CREATE PROCEDURE [customer\_modules].[decrypt]

( @data varbinary( 8000 ), @out\_ssn nvarchar( 11 ) OUT ) AS

BEGIN

                DECLARE @ssn nvarchar( 11 );

                SELECT @ssn = DecryptByKeyAutoAsymkey(

AsymKey\_ID ('customer\_asymkey\_primary'),

null, @data )

                SET @out\_ssn = @ssn;

END

go

-- Grant the modules the required permissions for execution using

--    module signatures.

ADD SIGNATURE TO OBJECT:: [customer\_modules].[encrypt]

BY ASYMMETRIC KEY [customer\_asymkey\_primary];

ADD SIGNATURE TO OBJECT:: [customer\_modules].[decrypt]

BY ASYMMETRIC KEY [customer\_asymkey\_primary];

go

CREATE USER [customer\_signing\_user]

FROM ASYMMETRIC KEY [customer\_asymkey\_primary];

GRANT CONTROL ON ASYMMETRIC KEY::[customer\_asymkey\_primary]

                TO [customer\_signing\_user];

go

-- Grant the encrypt and decrypt roles permission to execute these

--    new functions as appropriate.

GRANT EXECUTE ON [customer\_modules].[encrypt] TO [customer\_encrypt];

GRANT EXECUTE ON [customer\_modules].[decrypt] TO [customer\_decrypt];

go

An interesting note about this signed module example is that the secondary symmetric key is no longer needed because the members of the encrypt and decrypt roles do not directly have control permission on the root asymmetric key that protects the data.

SQL Server Audit

By partitioning your application data into multiple schemas with multiple roles for your users, you can better filter your audit record and minimize the performance impact while auditing access to sensitive data.

There are two primary scenarios where auditing makes sense:

The data is sensitive and you require an audit trail of any access to the data.

Users accessing the database are privileged, and though they cannot directly access the data, they may be able to infer or indirectly access it, and you want an audit trail to verify they have not done so.

Performance is always a concern that must be balanced with security – SQL Server Audit attempts to give you the right tools to balance your need for an audit trail with those of performance by allowing you to efficiently filter access to schema objects. Just as you can grant permission on a schema to a role, you can specify an audit of events on a schema by a role as well. For more information about security considerations and SQL Server Audit, see “Engine Separation of Duties Overview,” a white paper in progress (2008).

For example:

Transact-SQL

[[http://i.msdn.microsoft.com/Global/Images/clear.gif](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl17T-SQL');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl17T-SQL');)

USE [master];

go

CREATE SERVER AUDIT [customer\_audit]

TO FILE( FILEPATH = ’<secured directory>’, MAXSIZE = 10 MB );

go

USE [customerdb];

go

-- Audit select permission checks by the schema owners to ensure policy is not being

-- violated when this role is used. Select permission checks will occur whenever

-- data is read.

CREATE DATABASE AUDIT SPECIFICATION [customer\_audit]

FOR SERVER AUDIT [customer\_audit]

-- audit management

                ADD ( SELECT ON SCHEMA::[customer\_data]

BY customer\_owner,customer\_manager ),

-- Audit all select permission checks to Social Security Numbers (SSNs) because

-- they are sensitive.

--    This is accomplished by using the public role

--    that is defined as everyone in the database.

ADD ( SELECT ON SCHEMA::[customer\_ssn] BY public ),

-- Audit access to database objects so we can track

--    who accesses encryption keys.

ADD ( DATABASE\_OBJECT\_ACCESS\_GROUP ),

-- Audit any object changes (ALTER TABLE, ALTER CERTIFICATE, and so on).

ADD ( SCHEMA\_OBJECT\_CHANGE\_GROUP ),

ADD ( DATABASE\_OBJECT\_CHANGE\_GROUP );

go

Customer\_owner is a privileged role because it owns the schemas. Customer\_owner cannot be denied permission to select from tables in the schema it owns, but we can audit any select permission checks done using this role without impacting performance for roles designed to access the schema. The schema customer\_ssn is sensitive, so all attempts to read it should be recorded by any principal. By filtering out objects in other schemas and actions by other roles, you avoid the performance impact of generating the audit record. The performance impact of these two select audits will be minimal relative to auditing SCHEMA\_OBJECT\_ACCESS\_GROUP, which records all attempts to gain access to schema object (SELECT, INSERT, DELETE, and so on) on all schema objects by all principals.

Customer\_manager is also privileged, because by altering the objects in the schema, information can be inferred. By auditing every SCHEMA\_OBJECT\_CHANGE\_GROUP event you can verify whether these privileges are being abused to infer data by any admin role. The schema customer\_ssn is audited because it is sensitive – all access to it needs to be audited according to our self-imposed policy. The second granular audit does this.

The DATABASE\_OBJECT\_ACCESS\_GROUP and DATABASE\_OBJECT\_CHANGE\_GROUP action groups are specified to monitor access and changes to the encryption keys. These types of changes should be rare in a production environment and closely monitored due to their sensitivity. SCHEMA\_OBJECT\_CHANGE\_GROUP events should also be rare in a production environment. Auditing this action group can help detect anyone trying to infer data without directly selecting it. These events, which can include adding columns, removing columns, and modifying indexes are rare, so auditing this action group should have low performance impact.

Conclusion

By creating schemas other than dbo and managing the schemas and the roles that access them, you can develop a much more secure database application. By proper use of cell level encryption and auditing, you can better protect sensitive data and maintain an audit record of access to it and the actions of privileged users. These techniques better protect the data your application has entrusted to the database and help provide the protection and privacy users want.