



Separation of Duties in SQL Server 2014

SQL Server Technical Article

Writer: Bob Beauchemin

Technical Reviewers: Il-Sung Lee, Jack Richins, Darmadi Komo

Published: October 2013

Applies to: SQL Server 2014

Summary: The purpose of separation of duties to minimize the possibility of error and fraud, and this is accomplished by distributing privileges among users or roles, so that no one user or role has complete control over a business process or a piece of software. This whitepaper discusses SQL Server implementation specifics, including enhancements in later releases, which assist in the implementation of separation of duties, as well as proposing some mappings of sets of SQL Server privileges to often-used organizational roles.

Copyright

This document is provided “as-is”. Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

© 2013 Microsoft. All rights reserved.

Contents

Built-in Separation of Duties Features in SQL Server.....	4
Separation of DBA and OS Administrator duties	4
Permission-based administration and superuser replacement.....	4
Enumerating and classifying duties	5
Logins and users.....	6
Server roles and Database roles	7
Permission naming conventions and permission inheritance	8
Security Permission Statements	10
Bypassing the permission system	11
Superusers.....	11
Ownership chains.....	11
Module Signing and Impersonation.....	12
Implementing separation of duties in SQL Server	12
Conclusion.....	15
For more information:	15

The purpose of separation of duties to minimize the possibility of error and fraud, and this is accomplished by distributing privileges among users or roles, so that no one user or role has complete control over a business process or a piece of software. The basic premise of separation of duties can be seen in the series of checks and balances in accounting. Principal of least privilege helps to facilitate separation of duties, in that giving someone permission to perform process X may not necessarily mean that the same person should have the ability to perform process Y by default. This whitepaper explores the built-in features in SQL Server, and the improvements in later SQL Server versions, that can be used to facilitate this concept.

This whitepaper is not meant to be a comprehensive guide to separation of duties in SQL Server. Instead, it is meant to supplement two existing whitepapers, "[SQL Server Separation of Duties](#)" by Lara Rubbelke and "[Engine Separation of Duties for the Application Developer](#)" by Craig Gick and Jack Richins, which will be referenced when appropriate. Because both of those whitepapers describe SQL Server 2008 and 2008 R2, I'll also point out features that were added in releases since the whitepapers were published, namely SQL Server 2012 and 2014. In addition, the "SQL Server Separation of Duties" whitepaper describes the "[Separation of Duties Framework](#)" Codeplex project. This project can be used to facilitate implementation of separate of duties (SoD) in SQL Server. The in-depth description of setting up and using this project will not be repeated here, but I'll refer to it.

Although many existing SQL Server instances do not implement separation of duties, recent laws demanding the enforcement of the separation of duties' goals have made implementation of software

that follows these principals more important and more mainstream. There are, however, limits to how close you can get to the ultimate goal that depend on the size of an organization. For example, in a one-person group (or an N person group where N-1 people can be on vacation at the same time) it may be difficult, if not impossible, to completely enforce separation of duties. Another limitation to keep in mind is that physical access (to the instance or the file system) may be used to defeat any software-based permission system. In addition, a proper implementation of any security concept is not an ease-of-use or ease-of-administration feature. Implementation of this concept may take up-front work and make the job of the database administrator more complex.

Built-in Separation of Duties Features in SQL Server

Separation of DBA and OS Administrator duties

SQL Server has always supported the use of Windows logins, both individual Windows user logins and group logins. SQL Server can be run in Windows-only authentication mode, which means that only Windows principals can authenticate to the database at an instance-level; this is a best practice. On one hand, this facilitates the separation of duties between Windows administrators, who establish groups and add new users to the system, and DBAs who manage the instance, as well as possibly adding these existing Windows users and groups as logins and users, and assigning permissions. On the other hand, this integration can work against the separation of duties concept if SQL Server DBAs are also “identity administrators” or machine administrators and vice-versa.

SQL Server 2005 introduced WMI (Windows Management Integration) support and the SQL Server Configuration Manager graphic user interface-based utility, to allow separating the roles of Windows Administrator and DBA. DBAs must have some operating-level permissions in order to start up the instance and work with SQL Server Configuration Manager. Shutdown permission can be controlled with a SQL Server server-level permission.

SQL Server 2008 implemented the concept that “Windows Administrator is not sysadmin in SQL Server by default” out of the box. However a DBA must have some operating system authority (machine administrator) to install SQL Server. During setup, one Windows Account must be named, and this account is afforded sysadmin privilege. Best practice: The DBA account should not be used as an installation and maintenance account. An example of the separation of Windows Administrator and DBA in SQL Server 2008 is the SQL Server filestream feature, where the operating system portion of administering that feature is done through WMI (OS admin) and the database portion accomplished using `sp_configure` (DBA).

When SQL Server 2008 and above is run under Windows operating systems 2008 and above, service SID OS principals are used along with service accounts. These principals exist simply to provide each Windows service with a separate account. In addition service SID principals cannot be used to log on to the machine. In SQL Server 2008/2008 R2, service SID permissions are additive with service account permissions. In SQL Server 2012, these principals (known as “virtual service accounts”) can be used as the only service account. This separates the identity that is running SQL Server from the concept of interactive login.

Permission-based administration and superuser replacement

In SQL Server 2005, an infrastructure was put in place to be able to manage SQL Server solely with grantable permissions. Later versions add refinements to the permission and principal system, which I'll

discuss later in the paper. However, there is still some functionality that can only be accessed using built-in server and database superusers and superuser roles. This functionality consists of most DBCC operations and some system stored procedures. These (specifically DBCC) are needed in order to allow DBAs to manage SQL Server. For example, DBCC CHECKDB is the only way to perform a database consistency check. However, SQL Server 2005 also introduced impersonation and signed modules, which can be used to perform sysadmin-only activities. It is a best practice to encapsulate sysadmin-only functionality in signed modules. System stored procedures that check for sysadmin and other server and database roles are also being phased out and replaced by DDL statements that use the permission system. Therefore, prefer DDL to system stored procedures where the equivalent functionality exists. In addition, SQL Server 2005 also introduced the ability to rename the sa login account to assist in “hiding” that superuser login.

However, using granular security also runs counter to ease-of-administration and introduces complexity that can end up causing errors. Assigning permissions to every column on every table to each individual user would require inordinate resources to implement and maintain. Windows Groups, SQL Server roles, permission inheritance and permission sets make security easier to setup and maintain at the expense of granular permission administration, and sometimes granular auditability. Finally, security checks can slow down database response time slightly. I’ll talk more about the principal and permission system later in the paper.

SQL Server 2008 also introduced policy-based management, which made it possible to automate and enforce policies, including policies implementing separation of duties, over a series of instances. A built-in auditing function, with audits at both the server and database level was also introduced in SQL Server 2008.

Enumerating and classifying duties

All separation of duties implementations must begin by enumerating the specific duties to be performed. Then you can determine what different roles exist in the organization and which users/groups can perform which roles. In SQL Server we must start by separating inter-instance activity into two different levels: instance-level and database-level. We’ll start by asking questions about the existing system and use this as a framework for establishing roles. Note that the names I’ve given to these personas are “common parlance” type names; in your organization there may be specific departments that do this or you may call the role something different.

- Who installs the SQL Server software and service packs and upgrade the instance to new versions? (Installer)
- Who adds new logins to the instance? (Instance identity manager)
- Who can change login passwords or “reset” enable/disable a login? (Login admin)
- Who assigns permissions to those logins? (Instance security administrator)
- Who owns instance-level database objects (e.g. endpoints, linked servers)
- Who maintains the master and MSDB database and other “admin utility” databases? (DBA)
- Who backs up and who restores databases and checks data consistency (DBA)
- Who installs new applications and upgrades existing ones (DBA and Application DBA)
- Who adds new users to an application database? And what logins have application-level database permission (Database identity manager)
- Who assigns database and application-level permissions (Application DBA)
- Who owns databases and schemas within databases (specific login/user)

- Who insures that user data is not compromised and insures compliance with regulations (Auditors)
- Are there multiple levels of these roles, especially DBAs
 - Is there a “troubleshooter” role at an instance or application database level?
 - Do less senior DBAs have less privileges?
- What SQL Server Agent run and what identity do they use?
- Who is allowed to introduce data into the instance and extract data from the instance and what identities and permissions do they use? Examples would include:
 - Linked servers to access external data or allow access to internal data
 - BCP, BULK INSERT, and OPENROWSET(BULK)
 - SQL Server Integration Services jobs
- What outside vendor systems (e.g. job scheduling, performance monitoring) participate in the instance and what identities/permissions do they need to use and why?

Once you’ve answered these questions, you can begin to assign separate duties to roles. At that time, you can take note of where roles overlap (having multiple roles required to perform a duty is part of the definition of SoD). You can also visualize where a single individual could be a source of damage or fraud, and define your roles to ameliorate this problem. Vendors are third-party products for SQL Server would do well to go through this exercise for the purpose of allowing potential buyers to implement separation of duties and therefore making their products more marketable.

Keep these SQL Server technical requirements in mind during this exercise. To grant, revoke, or deny a permission on a database/instance object, you must have be the object owner, have CONTROL permission on the object or be a member of the securityadmin (server) or db_securityadmin (database) fixed roles. Also the separation of duties whitepaper provides an implementation of using signed modules to replace superuser-only activities. Adding a signature to a module requires ALTER permission on the module, CONTROL permission on the signing key or certificate, and in some cases, a password. SQL Server has multiple permission levels and two main verbs for dealing with permissions: GRANT and DENY. Since a DENY at any level can override a GRANT, you can assign permissions by starting with a high-level GRANT and subtracting with DENY, or starting with no permissions and doing additive granting.

It is useful as a first step to reiterate the main principal and concepts. This also helps locate idiosyncrasies in the system that cannot be overlooked without defeating the purpose of a comprehensive implementation.

Logins and users

Access to SQL Server is login-based. Logins can be Windows users, Windows groups and SQL logins. Authorization for Windows-based logins uses Active Directory or the local machine as an authenticator. SQL logins use a password as an authenticator. Logins can be disabled, but disabling a login means that the principal cannot connect to SQL Server; it does not keep it from being used by impersonation. The CONNECT SQL permission is also required to connect to SQL Server and is granted to each new Windows and SQL login by default. Because SQL Server permissions can be granted to logins and roles, but not directly to certificates and keys, some special login types exist to serve as a container for permissions. These are logins for certificates and asymmetric keys. These logins may not be used to log into the SQL Server directly, although in the case of Service Broker they can be used to allow inter-instance access to

and communicate with SQL Server through Service Broker messages using the SERVICE_BROKER endpoint.

Access to SQL Server databases is user and database role-based. In the usual case, users are mapped to logins. However, users without logins are allowed to facilitate custom permission schemes. In addition, users may be created for certificates and asymmetric keys as a container for permissions when using signed modules and for Service Broker, as with key-based logins. Some special key-based logins are pre-provisioned in the master database and key-based and SQL users are pre-provisioned in the master and MSDB databases. These should not be removed or disabled. In addition, in the MSDB database the guest account is provisioned as enabled and should not be disabled.

SQL Server 2012 introduced contained databases and user-based authentication, which blurred the line between login and users somewhat. Users in a contained database can be used as a mechanism to authenticate to SQL Server and is an alternative to login-based authentication. Contained database user-based authentication must be enabled as a configuration option and is not enabled by default. Special care must be used with contained databases because attaching a contained database can give access to the instance to that database's users. You can bring a contained database to online using RESTRICTED_USER mode to prevent this. In addition, in a contained database the ALTER ANY USER permission can be used to effectively permit access to the instance and access to other databases if the guest account is enabled in those databases.

Server roles and Database roles

A SQL Server role maps a set of principals to a set of permissions and is meant to make things easier to manage, just as Windows groups make the Windows operating system easier to manage. Roles are either built in to SQL Server (and cannot be removed) or are user-defined. Roles cannot be disabled because they are not used for authentication.

Server roles contain logins or other server roles. SQL Server has always included built-in server roles. The built-in server roles, except for sysadmin and public, should not be used in future development, and no logins should automatically be added to the sysadmin server role. The deprecated system stored procedure sp_srvrolepermission enumerates the permissions for the fixed server roles and is deprecated because it defines server role permissions in part through access to deprecated system security procedures. The permissions of fixed server roles are listed in SQL Server Books Online at <http://msdn.microsoft.com/library/ms175892.aspx>. Giving a login equivalent permissions to a fixed server role does not automatically add the login to the server role, because in many cases the engine checks only for role membership, ignoring permissions (including permissions that are denied) completely. Except for the public server role, permissions cannot be changed for the built-in server roles. SQL Server 2012 added user-defined server roles, which map principals to permissions and work as expected with denied permissions. Prefer user-defined server roles to fixed server roles.

Database roles contain users or other database roles. Like fixed server roles, permissions cannot be changed for the built-in database roles. As with user-defined server roles, permissions can be changed or denied for user-defined database roles. The permissions of fixed database roles are listed in SQL Server Books Online at <http://msdn.microsoft.com/library/ms189612.aspx>. Giving a user equivalent permissions to a fixed database role does not automatically add the user to the role, as with fixed server roles. The db_owner role has additional permissions outside the permission system (e.g. DBCC CHECKDB). Prefer flexible user-defined database roles to built-in database roles.

The MSDB utility database contains additional database roles for controlling and administering SQL Server utility functions including roles for

- Database Mail
- SQL Agent
- Data Collection
- Policy-Based Management
- Server Group Administration
- The SQL Server Utility

These SQL Server roles are needed to administer built-in features of the product and should not be removed.

Permission naming conventions and permission inheritance

The permission system includes different levels of permission as described in SQL Server Books Online here: <http://technet.microsoft.com/en-us/library/ms191291.aspx>. Learning the permission system is crucial to setting up separation of duties, especially because of the intricacies of permission inheritance.

The permission hierarchy consists of permission names and securable classes. There are 25 built-in securable classes including SERVER, DATABASE, SCHEMA, OBJECT, and additional classes. There are not specific securable classes for individual schema objects (e.g. table) except in the case of TYPE and XML SCHEMA COLLECTION. The additional securable classes that inherit from DATABASE consist of items that exist at database scope rather than schema scope (e.g. ASSEMBLY, USER).

The permission system has two types of permission inheritance: permission covering and the permission hierarchy. Permission **covering** exists at a single hierarchical level. In every case CONTROL is the highest permission and covers (implies) lower level permissions such as ALTER, VIEW DEFINITION, and TAKE OWNERSHIP. Not all permissions participate in permission covering. The permission **hierarchy** includes the SERVER, DATABASE, SCHEMA, and OBJECT levels. Granting permissions at a higher level automatically grants the equivalent permissions at lower levels of hierarchy, e.g. granting SELECT on a SCHEMA automatically grants SELECT on every OBJECT in the SCHEMA. You can see all of the permissions in the permission system as well as observe permission inheritance by issuing the statement:

```
SELECT * FROM sys.fn_builtin_permissions(DEFAULT);
```

At the top of the hierarchy is CONTROL SERVER permission. Having CONTROL SERVER automatically gives the login or server role all of the permissions in the permission system via covering and the permission hierarchy. This permission is equivalent to having CONTROL permission on all permission classes, child classes, and objects. This means that, unless there is a lower-level DENY, a principal (login or server role) with CONTROL SERVER permission can GRANT any permission in the hierarchy.

All permission classes except SERVER have a specific CONTROL permission. This means that any server principal with CONTROL SERVER or database principal with CONTROL on that database can GRANT any permission in that particular database. There is no 'CONTROL ANY DATABASE' at the server level, so no permission except CONTROL SERVER can GRANT permissions in all databases. All DATABASE permissions

inherit directly from some SERVER permission. Most other securable classes include ALTER, VIEW DEFINITION, and TAKE OWNERSHIP permissions as well.

The SQL Server books online provides code for a function called `ImplyingPermissions` at [http://technet.microsoft.com/en-us/library/ms177450\(v=SQL.105\).aspx](http://technet.microsoft.com/en-us/library/ms177450(v=SQL.105).aspx). This function takes a permission class name and a permission name and returns a resultset of permissions at a higher level of inheritance or covering that automatically give you the input permission. For example, issuing the statement:

```
SELECT * FROM dbo.ImplyingPermissions('SCHEMA', 'ALTER');
```

returns a list of the higher-level permissions that imply ALTER at the SCHEMA level, up to and including CONTROL SERVER. I've written a companion function, `dbo.ImpliedPermissions`, that takes permission and returns a resultset of all the lower-level permissions that permission implies. For example, issuing

```
SELECT * FROM dbo.ImpliedPermissions('SERVER', 'CONTROL SERVER');
```

returns the entire permission hierarchy. Here is the code for that function:

```
CREATE FUNCTION dbo.ImpliedPermissions(@class nvarchar(64), @permission_name nvarchar(64)
)
RETURNS TABLE
AS RETURN
SELECT a.*, b.height, b.RANK
FROM sys.fn_builtin_permissions("") AS a
CROSS APPLY dbo.ImplyingPermissions(a.class_desc, a.permission_name) AS b
WHERE b.CLASS = @class AND b.permname = @permission_name
GO
```

Note: There is currently a gap in the permission inheritance for two of the new SQL Server 2014 permissions that is reflected in the hierarchy.

Perhaps the most difficult concept to understand is the distinction between CONTROL, ALTER, CREATE, and ALTER ANY.

- CONTROL is ownership
- ALTER is ALTER/DROP plus change properties (except ownership) and it uses inheritance. ALTER SCHEMA (or ALTER DATABASE) is necessary to create objects in a particular schema.
- ALTER ANY is ALTER on all objects of that securable class plus CREATE – I refer to these as permission groups
- CREATE (at server, database, schema)

There is a specific CREATE DATABASE permission (at the server level), which also allows restoring the database but only if it does not already exist. CREATE ANY DATABASE at the server level not only allows creating a database, but also allows restoring over an existing database of the same name.

Four new permissions (on groups of securables) were introduced in SQL Server 2014. These are:

- CONNECT ANY DATABASE – at server scope

- SELECT ALL USER SECURABLES – at server scope
- IMPERSONATE ANY LOGIN – at server scope
- ALTER ANY DATABASE EVENT SESSION – at database scope

These new permissions can be used to implement separation of duties. Specifically issuing DENY of one or more of these permissions can be used to restrict the scope of permissions like CONTROL SERVER. Denying CONNECT ANY DATABASE effectively locks the login principal in question out of the server. Granting one or more of these permissions can be used to facilitate access to every database or every user securable for roles like auditing. More information on how to use these roles for separation of duties is contained in later sections of this paper.

Database level permission groups exist for the purpose of defining an overall administrator at the database level for a specific function. Because schemas are used to manage database objects, there are no object-level “ANY” permissions, with the exception of the new SELECT ALL USER SECURABLES permission. In addition, database level permission groups **almost** always have a parent of CONTROL SERVER. An example of this inheritance would be ALTER ANY DATABASE. A counter-example would be that the parent of database-level ALTER ANY USER is not ALTER ANY LOGIN.

Security Permission Statements

GRANT, REVOKE, and DENY are SQL statements used to implement the permission system. GRANT is a positive permission action and DENY is a negative permission action, REVOKE can revoke a GRANT or a DENY. All permissions in the permission system are available using GRANT and DENY. A DENY at any level for a login or user, or for any role the login or user is a member of, will override a corresponding GRANT. You cannot GRANT a permission to yourself; however, you can GRANT a permission to a server role or a role that you are a member of. You can also grant a permission to a server role (SQL Server 2012 and above) or database role and then add yourself to the role if you have the appropriate permissions on the securable and on the role.

For best separation of duties, you can separate the creation of principals from the granting of permissions to those principals. If the same principal controls principal creation and permission granting in a database and has impersonation privilege this is equivalent to granting CONTROL on the DATABASE to that principal. You can deny impersonation on all principals in SQL Server 2014 with the DENY IMPERSONATE ANY LOGIN permission. This permission also will deny the login from impersonating any users in any database as well. Note again that members of the sysadmin fixed server role cannot be denied anything, because they do not use the permission system. There is currently no equivalent “IMPERSONATE ANY USER” permission at the database level.

If possible, isolate permission-granting principals from principals that own objects if possible, although the owner of a securable can always grant permission to it. Ownership can be changed post creation – and assigned to a user without a login or a disabled, non-sysadmin login in the case of databases and other server scoped securables (e.g. endpoints). CONTROL confers ownership-like capabilities. CONTROL, ALTER, CREATE are permission-granting. Separate permissions to define logins and users from other permissions. The permission to create logins, users and corresponding roles does not include the ability to grant permissions on any non-principal securable. Separate identity administration from day-to-day operations.

Be careful of GRANT with GRANT OPTION. You can use the sys.server_permissions and sys.database_permissions metadata views to determine which principals have been granted permissions with grant option. Securable owners and principals with CONTROL can GRANT with GRANT OPTION. The CONTROL SERVER permission effectively allows grant option because it includes CONTROL permission of every securable, and principals with CONTROL on a securable can grant permissions on that securable.

Bypassing the permission system

Although the permission system is granular, intricate, and covers almost all server and database activities, there are two main constructs in SQL Server that bypass the permission system completely: superusers and ownership chaining.

Superusers

Superusers exist at a login, server role, and database level. A superuser skips all built-in permission checks and does not participate in permission-based security. The most visible built-in superusers are the SA login, the SYSADMIN server role, and the DBO database user. Although DBO is normally thought of as the owner of a database (CREATE DATABASE ... AUTHORIZATION...) DBO blurs the distinction between user and role in the all members of the sysadmin role are also DBO in every database.

Although the SA and DBO superusers are well-defined, membership in the SYSADMIN role is almost always overused. During installation of SQL Server, a SYSADMIN Windows-based login must be named. The SQL Server Agent and SQL Server service accounts are documented as requiring the SYSADMIN role and are added to this role during setup. In addition, the NT SERVICE\SQLWriter and NT SERVICE\Winmgmt Windows accounts are added to SYSADMIN role during installation. Members of the SYSADMIN role can be enumerated by using the system stored procedure sp_helpsrvrolemember.

The closest permission-based equivalent to the SYSADMIN server role is CONTROL SERVER. CONTROL SERVER differs from SYSADMIN in that:

- CONTROL SERVER does not map to DBO in every database
- Logins with CONTROL SERVER do not have a default schema in any database
- CONTROL SERVER does not permit access to DBCC or some system stored procedures

The closest permission-based equivalent to DBO is CONTROL DATABASE.

Superuser Best practices:

1. Do not assign superusers to individuals or roles.
2. Encapsulate superuser-only functionality in signed modules.
3. For separation of duties, always attempt to find a way to use the permission system rather than circumvent it using superusers.
4. Work with product vendors to eliminate requirements for superusers in their products.
5. Do not alter the built-in logins that require SYSADMIN server role.
6. Audit changes to the SYSADMIN built-in server role or prevent additions through built-in audits and policy-based management.

Ownership chains

SQL Server contains a built-in, non-removable functionality known as ownership chaining. Ownership chaining can circumvent the DENY functionality because, when a module conforms to the requirements for ownership chaining (the module owner and the owner of object references in the module are the same), permission on the objects in the module are never checked. Another way to express this is that

use of ownership chaining can have the unintended consequence of allowing access to user data through modules (stored procedures, triggers, and functions) that have explicitly been denied access.

However, ownership chaining is almost a requirement because SQL Server executes modules using the caller's identity rather than the owner's identity. Ownership chains therefore provide the ability to encapsulate access to database objects (e.g. tables) in modules (e.g. stored procedures) without giving the user of the stored procedures direct permissions on the underlying tables. It's also considered by some DBAs to be a best practice to implement user-level security this way. Another good thing about ownership chaining is that dynamic SQL breaks an ownership chain. If users are not permitted direct table access, this fact discourages use of dynamic SQL in modules. SQL Server 2005 did implement the EXECUTE AS OWNER clause in stored modules, but using this implementation loses the ability to determine the caller's identity inside the stored module without including specific "impersonation revert/re-impersonation" code that needs to be repeated in each procedure and audited specifically. Many SQL Server implementations control access to user data through a combination of the EXECUTE privilege on modules and ownership chaining with little or no user of table-level permissions.

Module Signing and Impersonation

Signed modules provide an auditable and secure mechanism to enable permission to execute specific tasks. A module may be a function, trigger, assembly, or stored procedure. Signed modules will enable organizations to maintain the proper level of security while ensuring the DBA is empowered to be responsive with minimal permissions. Signed modules are described in detail in the "SQL Server Separation of Duties" and "Engine Separation of Duties for the Application Developer" whitepaper, and the "SQL Server Separation of Duties" whitepaper also instructions on setting up and using the Separation of Duties Codeplex project. Please consult those whitepapers for more specific implementation details. Module signing requires ALTER permission on the object and CONTROL permission on the certificate or asymmetric key.

SQL Server 2005 introduced impersonation both on modules (stored procedures, user-defined functions, and triggers) and in ad-hoc batches using the EXECUTE AS statement or clause. Any login or user, including superusers, can be impersonated if the principal creating the module or running the batch has impersonation permission on the principal named in the EXECUTE AS statement. Because superusers do not participate in permission checks, they can impersonate anyone. The CONTROL SERVER permission can impersonate any login or user by default. In SQL Server 2014 a new permission IMPERSONATE ANY LOGIN exists. The IMPERSONATE ANY LOGIN permission also "sifts down" through permission inheritance to the database USER level. For example, DENY IMPERSONATE ANY LOGIN to a login results in denying permission to impersonate any USER as well. Best practices include:

1. Prefer signed modules to impersonation.
2. Deny any server principal that can log in to the instance impersonation rights by default (e.g. GRANT CONTROL SERVER + DENY IMPERSONATE ANY LOGIN)

Implementing separation of duties in SQL Server

Now that we've defined the duties and understand the permission system, we can provide some examples of personas (the terms role and group are overused and overloaded) and how you'd use the permission system to implement separation of duties. Bear in mind that, depending on size structure on your specific organization, some person or group may cover multiple of these personas. You can implement personas through the user of SQL Server server and database roles.

Policies/best practices:

1. Avoid the use of superusers. Document and audit any use of the superuser identities.
2. Document your use of ownership chains or at least be aware of it. In addition, you can create “owner accounts” that are not used for authentication (disabled logins, users without logins).
3. Instance-level securables can either be owner by the login who creates them, by a specific securable-owner login or server role, or by SA. A signed module could be implemented to give control of these securables to SA.
4. User Databases should be owned by SA or by the same securable-owner login as owns other instance-level securables. This owner should be present in every instance in the organization to prevent a NULL owner problem if the database is moved to another instance.

Software installer – The software installer user can be provisioned as the login that must be given administrative access during the SQL Server installation process. During installation, this login assigned to the sysadmin superuser server role. A specific Windows user account can be designated for this purpose, and the account disabled in Active Directory when not performing SQL Server installation/maintenance activities. Simply impersonating this account will not confer the necessary OS-level permissions.

Instance identity manager – This can be designated as a SQL Server server role in SQL Server 2012 and above. It would have ALTER ANY LOGIN permission. This persona would also have ALTER ANY SERVER ROLE permission in SQL Server 2012 and above, for the purpose of assigning logins to roles. Lower level roles which require a subset of ALTER ANY LOGIN permission would be accomplished with signed modules.

Highest-level DBA – This could be a server role in SQL Server 2012 and above. DBAs need to be able to control, troubleshoot, and respond to outages in a timely manner. To accomplish this, the highest level DBA would need:

- Signed modules to accomplish sysadmin-only utility functions
- GRANT CONTROL SERVER permission
- DENY ALTER ANY LOGIN and ALTER ANY SERVER ROLE permission
- DENY IMPERSONATE ANY LOGIN in SQL Server 2014 or at least DENY IMPERSONATE LOGIN:SA in pre-SQL Server 2014 instances. There may be other specific logins where this privilege may be denied.
- DENY EXECUTE/SELECT/INSERT/UPDATE/DELETE on a per-database schema or table level DENYS to prevent access to sensitive data or all user data if desired or required. Denying the SELECT ALL USER SECURABLES permission in SQL Server 2014 would be an alternative, but would have to be combined with DENY EXECUTE because of ownership chaining.
- Controlled and audited access to a login in the SYSADMIN server role for emergencies.

Remember than CONTROL SERVER permits the role to assign permissions (except specifically to that login, on a login-by-login basis). It would also make this role able to create and control credentials for SQL Agent jobs and to install cryptographic providers.

Because DBA has CONTROL SERVER permission, to limit the proliferation of this permission, the DBA also inherits all of the persona’s roles to which a specific DENY is not issued.

- Manage and administer permissions for MSDB database.

- Be able to backup all databases
- Be able to create and restore user databases

Lower-level and more specialized DBA functions – To distinguish between levels of DBAs to prevent errors, the following permissions can be denied along with granting the CONTROL SERVER privilege.

- DENY ALTER ANY CERTIFICATE or DENY ALTER ANY ASYMMETRIC KEY – would prevent the creation of signed modules in the “Separation of duties framework” database [1] or in the master database.
- DENY CONNECT to specific user databases
- DENY permissions on certain instance-level securables (e.g. DENY ALTER ANY LINKED SERVER)

Instance security administrator – In order to grant and deny permissions you must have any CONTROL privilege or be the object owner. At an instance level this persona would have to have CONTROL SERVER permission. Unless you want to manage individual server-level objects by ownership or through signed modules, this role would likely be combined with the DBA role. If a specific securable-owner permission system is set up (see above under “policies/best practices”) this persona could be used as the instance security administrator.

Backup operator could be assigned to a specific login that job schedulers and third-party utilities could use to automatically backup the databases. Database consistency checks should be performed through signed modules.

Troubleshooter – If a specific troubleshooter persona is required, this can be granted access to the permissions to achieve that functionality. This may include:

- ALTER ANY EVENT SESSION/ALTER ANY DATABASE EVENT SESSION
- ALTER TRACE
- ALTER ANY SERVER EVENT NOTIFICATION/ALTER ANY DATABASE EVENT NOTIFICATION – if these are to be used for troubleshooting
- SHOWPLAN in the appropriate user database(s)

Application installer – This persona would have to have CREATE DATABASE (instance-level) permission if the application requires its own database or at least CREATE SCHEMA permission if a database must be shared between multiple applications. It can also be database owner during the installation for ease in installing database-level securables. After the installation is complete, the CREATE DATABASE permission could be revoked, the database ownership transferred to SA, and object-level securable access reverted to the schema owner. There can be a separate owner of database-level securables, or they can be owned to an account designated by the install process.

Database Identity Manager – This role would be managed by the application DBA or by an application-appointed user. In addition to management of database identities, some applications implement an application-specific security system outside of SQL Server security. These identities would need to be managed too.

Database and Schema owners - Database owners should be SA rather than a login associated with Windows user and Windows group. Schema owners should own resources, they should not be

application users as illustrated in the “Engine Separation of Duties for the Application Developer” whitepaper.

Introduce out-of-instance data – This role would be delegated to the DBA, of course, but could also be given to specific individuals. DBAs would perform linked server installation of maintenance, via manually audited requests. Outside individuals could be given ADMINISTER BULK OPERATIONS and be made members of SSIS and SQL Server Agent roles as needed. Something to keep in mind is that, if a SQL Server Agent job is owned by SA, all T-SQL jobsteps will be run as SA with the result of bypassing the permission system.

Auditing – The auditing role can be assigned CONNECT ANY DATABASE and VIEW ANY DEFINITION in the server level to facilitate audit metadata collection. If SQL Server’s built-in auditing facility is used, ALTER ANY SERVER AUDIT and ALTER ANY DATABASE AUDIT should be granted. If user table auditing is required, granting SELECT ALL USER SECURABLES at the server level is a good minimum-permission alternative to granting the equivalent permissions on every new database after it is created.

Conclusion

Consideration of separation of duties concerns have been specifically addressed since SQL Server 2005 with a granular permission system, and have kept pace with the introduction of new securable types in later versions of SQL Server. Introduction of server roles in SQL Server 2012 and introductions of new permissions in SQL Server 2005, 2008/2008 R2, 2012, and 2014 make it easier to tailor separation of duties to your organization’s specification. Signed modules provide a safe way to encapsulate duties that have not been migrated to the permission system while resorting to superuser usage as little as possible.

For more information:

<http://www.microsoft.com/sqlserver/>: SQL Server Web site

<http://technet.microsoft.com/en-us/sqlserver/>: SQL Server TechCenter

<http://msdn.microsoft.com/en-us/sqlserver/>: SQL Server DevCenter

Did this paper help you? Please give us your feedback. Tell us on a scale of 1 (poor) to 5 (excellent), how would you rate this paper and why have you given it this rating? For example:

- Are you rating it high due to having good examples, excellent screen shots, clear writing, or another reason?
- Are you rating it low due to poor examples, fuzzy screen shots, or unclear writing?

This feedback will help us improve the quality of white papers we release.

[Send feedback.](#)